UNIVERSITY OF CALIFORNIA

SANTA CRUZ

Automatic Process Selection for Load Balancing

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

William Osser

June 1992

The thesis of William Osser is
approved:

Prof. Darrell D. E. Long

Prof. Kimberly E. Taylor

Prof. Charles McDowell

Dean of Graduate Studies and Research

Contents

List of Tables

Automatic Process Selection for Load Balancing

William Osser

ABSTRACT

Distributed operating systems give users access to multiple computational engines throughout the system network. Users of one workstation are not hindered by the CPU intensive applications run on a different workstation. However, when a large number of machines in the network are idle, the efficiency of computation is decreased. Load balancing promises to alleviate this problem by sharing the workload on heavily loaded workstations with lightly loaded workstations. A threshold can be established so that only a limited number of processes are transferred to the lightly loaded sites, so that the load at those sites does not inhibit local execution of processes.

While load balancing has been studied extensively, there have been few studies on determining which processes should be relocated to increase the throughput of the system. We present an automatic process selector for load balancing that chooses processes to be relocated based on the history of its performance. We have implemented this system on the Sprite distributed operating system [1].

To my parents, neither of whom ever had any difficulty deciding which tasks should be distributed to their sons.

Acknowledgments

First and foremost, I would like to thank Professor Darrell Long. Darrell captured my interest when describing operating systems, Sprite in particular, and made sure that I did not fall too far behind my time schedule. I also want to thank Dean Long for the large amount of time he invested in reviving Sprite each time it died. Rich Golding was and still is a constant source of inspiration to me, and he gave me enough faith in myself to complete this project. Bruce Montague spent an large amount of time reminding me how to write in the active voice, and was always available for forays into the shell and kernel code. Other people in the CSL that I must mention are Michelle Abram, Bob Ellefson, Nitin Ganatra, Fred Long, and Carol Osterbrock, who put up with my endless use and abuse of several of the lab resources.

Kim Taylor and Charlie McDowell were very patient, allowing me give them a rough draft whenever I could, and making several pertinent suggestions for improvement. I want to thank my fiance Andrea Silver (soon to be Osser) for putting up with me over the last few months. I also need to thank K.B. Sriram, Max Copperman, Professor Dave Haussler, and especially Phil Long for describing the two-armed bandit problem, and pointing out how it is applicable in this system. Douglas Jones proof read several revisions of my thesis to give me a sanity check. Finally, I want to thank Fred Douglis for answering numerous questions on Sprite, and giving me the pointers I needed to find many of my referenced papers.

Chapter 1

Introduction

As the power of individual workstations increase, distributed systems are becoming more popular. Users have all of the capabilities that are provided by their workstations, and at most times, this is sufficient. However, there are cases when the power of one workstation is not sufficient to complete all of the tasks at hand. One solution to this problem is to distribute some of the tasks to an idle workstation. Studies have shown that over 65% of workstations are idle at any given time [2, 3]. Distributing processes over all of the workstations in the network balances the load at each machine so the overall time needed to complete tasks is reduced.

Automatic load balancing is a system process that distributes tasks from heavily loaded workstations to idle workstations without user intervention. This user-transparent process provides a greater amount of processing power to all users of the system. Early studies showed that automatic load balancing can be achieved in multiprocessing systems [4] (before distributed systems were commonly available). Since then, several different algorithms have been presented for load balancing of multiprocessing and distributed system. The load from one workstation can be transferred to a lightly loaded workstation through two different methods. One method is to start a process remotely and wait for it to terminate, using tools such as *rsh* [5]. Another method uses a mechanism called process migration [1].

Process migration in the Sprite distributed operating system [1] is defined as being able to halt a process during its execution, freeze its current state, move it to a different host, and then resume execution [2]. One of the advantages of process migration is that it can be used to distribute the load from heavily loaded workstations to idle ones during process execution. If an idle workstation becomes active again, any migrated processes can be sent back to their original locations.

While there are several different methods for balancing the load in distributed systems, there are few published methods of selecting which processes should be migrated for the best performance. We present a set of criteria that are used by an automatic load balancing scheme to select processes for remote execution based on past performance statistics. We have incorporated some of the ideas of previous proposals and working systems to implement an automatic process selector based on the performance history of each program. The criteria used to determine whether a process will be remotely executed is specified by either the user or the system administrator.

The rest of this document is organized as follows. In chapter 2, we examine previous research in automatic load balancing, automatic process selection, and the Sprite operating system, which we have chosen to use to implement our automatic process selector. In chapter 3, we describe the considerations and implementation of our system, and the criteria used to determine whether a process should be remotely executed. In chapter 4, we give a performance evaluation of our system. In chapter 5 we describe future work and alternate methods of implementation. In chapter 6, we present our conclusions on this project. Appendix A gives sample output from our process selector, while Appendix B gives a listing of some of the programs used to take measurements.

Chapter 2

Previous Research

Operating systems require different tools to support load balancing. Considerations include the implementation of the file system, the types of processes that will be executed on the system, and the process relocation mechanism. If a process can be relocated during execution, load balancing may need to be performed continuously. If a process is only assigned a location upon invocation, then the load balancer will only be needed at process activation. Remote execution is advantageous in a system where there are a number of long running jobs. A file system that is globally accessible provides better support for remotely executed processes than a file system in which some files cannot be accessed from any node.

Our goal is to create an automatic load balancer. This is an automated decision making process that removes as much responsibility from the users as possible while still providing a balanced load with close to optimum response time. Besides providing a mechanism to relocate processes to balance the load, we must also provide a method whereby processes are chosen automatically for remote execution. Processes that would experience performance degradation from remote execution should not be migrated, while those that would experience improved response time should be the first processes chosen for migration.

This chapter describes the requirements for an automated load balancing program that will decide which processes should be executed remotely, and which processes locally. First, we describe the requirement of the operating system to implement an initial placement policy and perform automated process selection. The different aspects of the Sprite operating system that makes it suitable for this purpose are discussed. Next, we present some of the different algorithms that have been developed for load balancing, as well as those that have been implemented in Sprite. Some methods of determining whether a process should be executed remotely are also examined.

2.1 The Sprite Operating System

There are several issues that must be considered when choosing an operating system to support automatic process selection for load balancing. The operating system must have a uniform name space for accessing files. Every process must be capable of executing on any homogeneous machine. If a process cannot be executed on every machine, then there is an additional cost that must be paid to identify such processes. There must be an inexpensive way to remotely execute a process. Additional overhead in executing a program remotely may degrade the performance of a process such that there is no benefit in executing it remotely. Finally, the performance information and resource usage of a process must be available, preferably to user processes. This information is useful for determining whether a process may be executed remotely.

The operating system that we have chosen to use is the Sprite operating system developed at University of California, Berkeley [1]. Sprite is a distributed operating system that provides a UNIX compatible interface. The name space is transparent to all of the clients [6], so all files are accessed uniformly throughout the system. Sprite provides process migration, which supports moving processes to any homogeneous machine before or during execution [2]. Sprite includes a migration daemon [7] which locates a site for remote execution and migrates a process to that site. Processes only migrate to workstations that have received no user input from the mouse or keyboard in the last thirty seconds. When input is received from either of these devices, all remote processes are evicted and sent back to the machine from which they were originally invoked. Processes migrated by the daemon are sent to machines that have been idle for the greatest amount of time.

Sprite's process migration is transparent to the user. When a process is migrated, all dirty pages are flushed to backing files. The state information of the process is then transferred to the target machine and needed pages are loaded from the backing files [2]. A process control block is kept on the machine that invokes that process for every process executed, including processes that have been migrated. This supports a mechanism whereby calls to the process can be forwarded to the machine where the process is executing. System calls are handled by the target machine with the exception of those calls that must be handled by the machine that originally invoked the process. An example of such a call is *gettimeofday*, which must be executed on its home machine since clocks are not synchronized between workstations.

Some processes can never be migrated. When a process is migrated, as much of its state is transferred as possible. State information private to the process and internal kernel state can be transferred without much difficulty. However, state information concerning certain physical devices on the home machine cannot be migrated. A program such as the X window system accesses the frame buffer, which is a physical device that cannot be transferred. Such processes are not flagged by the operating system, so any program that migrates processes needs to flag these as exceptions so that they are never migrated.

There are local devices that can be accessed remotely, as well as user and server processes that have been made available through the file system as *pseudo-devices* [8]. A pseudo-device is a process or device whose driver is a user process, instead of part of the kernel. This allows extensions to the file system without modifying the kernel. Pseudo-devices give system access to those objects for which the operating system handles I/O functions. Any access to a local device, local pipe, or a local pseudo device can be executed on a remote machine. Some of these devices are the keyboard, the tty, and the TCP and UDP servers, which supports access to NFS files instead of Sprite's own LFS file system [9].

Sprite satisfies the uniform name space requirement, as well as providing an inexpensive remote execution facility. It also provides performance and resource usage information. Since Sprite provides a UNIX compatible programming interface, certain system calls which return a process's performance statistics, such as getrusage and wait3, are available to user processes. In the current Sprite implementation, some of these statistics are simply returned as zero. However, with a small number of modifications, the needed values can be obtained.

Two programs that use the migration daemon to distribute work load are *Pmake* and *mig* [2]. The former program does a parallel compilation, separating each makefile command into a different process and migrating that process to an idle host [2]. The *mig* command migrates a user-specified process to an idle host. A recent modification to the command interpreter (shell) migrates specific processes using the migration daemon [10]. These processes must be specified by the user.

A problem with the existing method is that it only migrates processes to idle clients. Once a process has been migrated to an idle site, that site is no longer idle, and thus no more processes can be migrated to the site until the first process has terminated. A true load balancing algorithm would migrate more than one process to a *user*-idle site. We have found no measurements in the literature that justify the limit of one migrated process per workstation. It may be that in a network with a large number of workstations there are always enough idle workstations for process migration. However, in a small network, the benefits of load balancing would be limited. In this case, increasing the number of migrated processes on a client would clearly be desirable have to be increased.

2.2 Load Balancing Algorithms

In order to perform load balancing, it is necessary to identify nodes to which it would be advantageous to move processes. Once a source and target site are located, a process that would normally be executed on a heavily loaded node is executed on a lightly loaded node. Distributing the load in this fashion provides potentially faster response time to processes that would otherwise be executed on the heavily loaded local site.

There are two policies describing remote process execution, *initial placement* and *process migration*. The former policy places a process at a specific site, where it will remain until execution is terminated. Under this policy, the load cannot be fully balanced, since long-lived processes are bound to the site at which they begin execution. Distributing the load using an initial placement policy is called *load sharing*. An example of such a system is Utopia [11]. The latter policy, *process migration*, allows processes to be moved to a different site during execution. In this way, processes can be moved from any site so that the load can be maintained in a balanced state. Distributing the load using process migration is called *load balancing*. Examples of operating systems that provide *process migration* include Charlotte [12], V [13], Accent [14], and Sprite [1].

Several load sharing algorithms have been developed. Most of these algorithms can be implemented as load balancing algorithms, if the targeted operating system is one that supports process migration. Load sharing is simpler, but load balancing provides more benefits. In a system with a large number of long running processes, load balancing provides a finer granularity for "smoothing" out the load across the system. Any system that implements load balancing can implement a load sharing, or initial placement, algorithm, by simply adding the constraint that processes are only migrated when they are initially invoked.

Of the algorithms discussed in this section, some have been designed with load balancing in mind and some with load sharing. Since all of these can be implemented as load balancing algorithms, we refer to them as such. Algorithms that are specific to load balancing are noted as necessary.

There are two general classifications of load balancing algorithms, centralized and distributed. The former algorithm relies on a central server to determine both when processes should be migrated to balance the load, and where to migrate the processes. A distributed algorithm lets each client make its own decision based on data that is globally available, either through broadcasting over the network or via a shared file.

The machine that invokes a process is called the *source client*. The machine that executes the process, either through *initial placement* or *process migration*, is called the *target client*. In most of the following algorithms, the source client must exceed a specified load level before it can migrate processes away from itself. This load level is called the *threshold*, and can be set either statically or dynamically, depending upon the algorithm.

2.2.1 Distributed Algorithms

A distributed algorithm leaves the responsibility of determining what action to take to each client. The client must locate a target client to which processes can be migrated, or locate a source client from which a process can be migrated. The status of each client can be kept in a shared file, or can be broadcast by each client to all other pertinent clients. Since Sprite has no concept of topographic structure, network algorithms that rely on topology, as presented in [15, 16, 17, 18, 19], will not be discussed. The algorithms presented are required to provide clients access to every site involved in process migration.

Sender-Initiated

A *Sender-Initiated* algorithm [20] is one where an overloaded source client attempts to execute a local process on a remote target client. If a shared file exists, or load information is broadcast, the source client can choose a candidate based on this information. Otherwise, the source client must poll several sites to determine the most likely host for remote execution.

The two requirements for process migration to occur in this case are:

- The source client must have exceeded the current threshold.

- A target client must be found.

If the first requirement is not met, then there is no need to balance the load any further – all clients are below the threshold. If the second requirement is not met, then all target clients (of those polled) are above the minimum threshold, and the system is too heavily loaded for any useful load balancing.

There are several different methods for choosing a site for remote execution when using a *Sender-Initiated* algorithm: *Random*, *Threshold*, *Shortest Queue*, and *Broadcast and Bidding*. In each of these methods, the load balancing mechanism is invoked whenever a process is invoked and the load at the local processor is greater than the threshold value. The load balancing mechanism attempts to find another site to execute the process. The first three methods are described by Eager *et al.* [21].

In *Random*, a target client is chosen at random and the new process is migrated to that client. There is no state information passed between the clients, so the source client has no knowledge concerning the load of the target client. If the newly arrived process places the target client above the load threshold, the load balancer is used again to find another new site for the process. Thus, it is possible for a process to migrate to several clients before a suitable execution site is found.

In *Threshold*, a target client is chosen at random and polled to see if it can accept an additional process without surpassing the threshold level. If not, the source client must "keep" the process. The source client can make several attempts to find an eligible target client. Simulations show that large number of attempts (e.g., 20) provide no significant improvement over a small number (e.g., 3 or 5) [21].

In *Shortest Queue*, also known as *Lightest Load*, several foreign sites are polled to determine their load. The site with the load farthest below the threshold, if there is one, is selected from this set to be the target client. This requires more information to be transferred between the clients than either of the previous two algorithms.

Simulations performed using these algorithms assume that the largest amount of overhead is the CPU bundling. Results show that even a simple algorithm, such as *Random*, is a significant improvement over no load balancing [21]. *Threshold* gives even better results than *Random*. However, *Shortest Queue* provides very little improvement over *Threshold*, and costs the system more overhead time to poll potential target clients. Thus, more complex algorithms will not provide any significant performance benefits. Simple algorithms that need a minimum amount of information passed between clients provided the greatest speedup.

In *Broadcast and Bidding*, the source client broadcasts to all clients in the system when it has exceeded the threshold level. Target clients that respond bid for the extra process, and the "best" bid receives the process. This bidding scheme can take a number of different forms, such as those explained by Chang [22] and Ramamitham *et al.* [23]. A simple bidding scheme is to grant the process to the first site that responds, limiting responses to those clients with a load small enough to accept additional processes. A more complex method allows for differences in bidding weight among potential target clients that would occur in a network with homogeneous architecture but heterogeneous performance capabilities. A greater weight can be assigned to sites that have greater processing power, increasing the likelyhood that a process will be migrated there.

The disadvantage of this algorithm is that a broadcast, followed by the resulting responses, increases network communication traffic. To avoid network traffic, the number of sites receiving the broadcast must be limited to a small number, chosen at random. The site that can provide the best response time is chosen as the target client in the same manner as previously described. This algorithm differs from *Shortest Queue* in that different criteria can be used to determine the best target client. It remains to be seen if this more specific algorithm would provide better results than *Threshold*.

Receiver-Initiated

A *Receiver-Initiated* algorithm [20] starts at an underloaded client. When the load of a client is below the minimum threshold level, it requests processes from a heavily loaded client. The methods used for obtaining processes are similar to those used to off load processes in *Sender-Initiated* algorithms. However, these algorithms require process migration policies to be implemented, since it is not likely that an idle processor will locate a heavily loaded client immediately preceding process execution.

There are some differences in the benefits of *Sender-Initiated* and *Receiver-Initiated* algorithms [20]. The former produces better response times overall when the system load is low, and the latter when the load is high.

If the system load is low, idle clients searching for an overloaded client will consume more resources and generate more network traffic than a few overloaded clients searching for an idle client and migrating a process. Similarly, in a heavily loaded system, several heavily loaded clients searching for a few lightly loaded hosts will be more inefficient than a few lightly loaded clients requesting processes from heavily loaded clients.

Symmetric

Simulations have shown that the combination of the *Sender-Initiated* and *Receiver-Initiated* algorithms performs better than either single algorithm [24]. These estimates are made with the assumption that network delay is the largest factor, and not CPU bundling [20]. With a small network delay, there is no significant difference between *Sender-Initiated* and *Symmetric* at low system loads. A larger network delay resulted in no significant difference until the system load was greater than 80%.

Depending on the number of clients that are polled to find a target client, network traffic may be a factor [21]. For a *Sender-Initiated* algorithm, a smaller number of queries is more beneficial in a lightly loaded system. In a heavily loaded system a greater number of queries are necessary. A *Symmetric* algorithm must limit the number queries made when the system load is low to be comparable to a *Sender-Initiated* algorithm.

One method of combining these two algorithms is to dynamically choose either the *Sender-Initiated* or *Receiver-Initiated* algorithm based on the system load. A server detects the global load and broadcasts the algorithm to be used when there is a state change. This method suffers from periodic global polls determining the system load.

Shared File

The method of determining the load on each client previously used by Sprite involved maintaining a shared file to which all clients had access. When a client's state changes from active to idle, or idle to active, it accessed the file and changed its entry. The file was locked by the client until it was finished making changes to avoid concurrent access problems. When a source client wished to find a target client, it accessed the file, found an idle site, and migrated a process to that site [2].

This was inefficient as the file system had to be invoked each time the file was accessed. For consistency reasons, no process could retain the file in the cache of the local workstation. This method was seen as more costly than a centralized program that kept track of the load average of each workstation [7].

A different method that uses a shared file is a modification of the *Distributed Drafting* algorithm [25]. This algorithm is a *Receiver-Initiated* algorithm that employs lightly loaded sites in finding heavily loaded sites. When a heavily loaded site is found, the idle sites "draft" some of the processes and execute them locally. The difference between this algorithm and the previous one is that the Sprite implementation is concerned with the idleness of each workstation, where as this algorithm is concerned with the load of each workstation. A client can be in three different states:

- Below-threshold
- At-threshold
- Above-threshold

When a client changes state, it accesses the shared file and updates its state information. When a below-threshold client wishes to "draft" a process, it accesses the global file and finds an above-threshold client. To insure that the information in the shared file is accurate, an extra handshake between the above-threshold client and the below-threshold client is necessary. If the former client has changed state, then the "drafting" client must reaccess the shared file to locate an above-threshold client.

One way to reduce the amount of shared file accesses is to only update a client's state when there is an extreme state transition. Whenever the state of a site changes to above or below the threshold value, if it had previously been in the opposite state, it updates its entry in the shared file. This method avoids numerous messages being sent to the server when a client is continuously switching between the middle state and one of the two extreme states.

2.2.2 Centralized Algorithms

Centralized algorithms use a process scheduler that makes decisions concerning where a process should migrate. In many cases [26, 27], process schedulers take all incoming processes and place them at a lightly loaded client. Workload and client characteristics can be used to determine the overall system load [28]. These characteristics include the number of processes waiting in the CPU queue, CPU utilization, and number of jobs active at the client. These values are translated into vectors that are used to determine the most likely candidate to receive a process.

Sprite is currently using a centralized algorithm for load balancing [7]. Instead of a shared file, a global load average daemon maintains each client's load in its own virtual memory. This avoids the multiple accesses to the file system that were incurred in the previous version. Local load average daemons, executing on each workstation, periodically send the local state to the global load average daemon. Whenever a load average daemon is started, it attempts to locate the global load average daemon. If it cannot find one, it designates itself as the global load average daemon. If multiple load average daemons are present, only one of them becomes the global daemon.

When a process requests migration, the global load average daemon is invoked, and a target client is selected from the list of idle clients. This method is a *Sender-Initiated* algorithm that relies on the central server to determine the target client. This method provides the additional benefit of monitoring how many processes each client has migrated, and supports leveling out the system wide level of migration for each client. A problem with the Sprite implementation is that it is left to the users to explicitly invoke process migration. Although not user-transparent, this does allow users aware of the migration tool to execute processes on foreign hosts.

2.2.3 Choosing an Algorithm

Studies on Sprite and other distributed environments of differing sizes have shown idle hosts are plentiful, 65-85% in a network of over 60 workstation [2], and greater than 70% in a Michigan State University study with 17 workstations [3]. If this is the case, then the system load is never high enough to merit a *Receiver-Initiated* or *Symmetric* algorithm. Depending upon the system and its network, either a centralized or distributed version of the *Sender-Initiated* algorithm, using the *Threshold* method, will provide the best load balancing with the least amount of overhead.

2.3 Which Processes Can Execute Remotely?

Once a load balancing algorithm is determined, how can we decide which processes should migrate to a target client? The goal of load balancing is to increase the throughput of each workstation so that the maximal number of processes are executed in the least amount of time. The goal of process selection for remote execution is to find those process that, when executed remotely increase the overall throughput for the entire system. We must determine which processes can be migrated and which of those processes should be migrated.

In the UNIX operating system, a process that relies on a device that is not accessible to all clients cannot be remotely executed on those sites. In Sprite, there is a transparent name space enabling access of all files to all clients [6]. Therefore, there are no file dependencies to affect process migration. However, processes on Sprite that access physical devices may not be migratable, or they may experience performance degradation if migrated to a different site. Douglis and Ousterhout [2] pose the question:

> Which processes should be migrated? Should all processes be considered candidates for migration, or only a few particularly CPU-intensive processes? How are CPU-intensive processes to be identified?

While they provide the question, the issue is left open. We must find some means to determine which processes are the best candidates for remote execution. First, we must define which processes are eligible for migration. Haĉ and Jin [29] state:

> If, in the system not loaded by any additional processes, the mean response time of a process executed locally is greater than the mean response time caused by migration of this process and relative files, then this process is called *migratable* in the sense of load balancing. Otherwise, the process is called *nonmigratable*.

Such a definition is too constrained for a system where processes are migrated to improve performance locally. We believe that if, by migrating a process, the overall mean response time for all processes in the system is improved, then the process is migratable.

The optimal method would be to know which processes, when executed remotely, would give the best overall response time. However, there is no easy way of evaluating whether a process is migratable before it is executed, nor whether it will increase the throughput of the system when migrated. There are, however, certain indications that can be noted both during and after process execution which determine whether a process is migratable. In a study on multiprocessor systems [30], Gait states that during execution, a process may be rescheduled on another processor if:

- The local resident time slice becomes exhausted;
- An idle processor intervenes to globally schedule a low priority process waiting at a processor currently executing a high-priority process; (*Receiver-Initiated* load balancing) or
- The process is swapped out to make room for newly created or higher-priority processes.

If a process is being swapped out, this reduces the cost of the process migration as page swapping has already been performed. Therefore, the memory allocation mechanism might invoke a daemon to migrate any process being swapped out. Most processes in Sprite that are migrated complete their execution on their foreign site [2]. Therefore, under normal circumstances, it is only long-lived processes that will be swapped out. Large batch jobs of low priority may be identified if a majority of their pages in virtual memory are swapped out.

One way of deciding whether a process should be executed remotely is to require the user to give this information. Since each user has some idea of whether a process should execute locally or remotely, they can provide an optimized list of processes that can be executed remotely. This is not as optimal as a fully omniscient method, since some processes might be executed remotely even though the would complete sooner locally. However, the user must constantly maintain this list for good results.

A better solution would allow the system to automatically decide whether a process can be executed remotely. If the system maintains a history of the past performance of a process, it can use a number of different metrics to decide whether a process should be executed remotely. One drawback with this approach is that the system is guessing whether the process should be executed remotely. However, every time a process is executed, the system will have more information concerning the behavior of the process. In most cases, after a relatively few number of executions, the history of a process will be mature enough to accurately predict whether the process should be executed remotely.

2.3.1 Existing Schemes

Previous work on process selection for remote execution can be divided into two different approaches. The first method relies on the user specifying processes that are candidates for remote execution, or, alternatively, which processes should never be executed remotely. When a process is invoked, the user's list is consulted to determine whether the process should be migrated.

Two systems have been implemented using this method. One is *Utopia* [11], which runs on several operating systems, including UNIX and VMS, and the other is *msh* [10], which runs on Sprite. *Utopia* distributes the load of each machine within a cluster. This cluster is defined by the system administrator depending upon their needs and network organization. Clusters have a centralized cluster server that directs process placement within the cluster. Each cluster may be executing a different operating system, but there must be a channel of communication between the clusters.

Each user maintains a list of processes that can execute remotely, along with a detailed description of what resources each process requires. There is also a global list maintained by the system administrator that is available to the users. The requirements of a process can include descriptions of the CPU or I/O needs, as well as the operating system on which it should be executed. The benefit of such a heterogeneous system is that a process that is unavailable on the local machine is made available to the user by executing it on a remote machine. The question this raises is whether the file system name space is truly transparent. A UNIX process could not be transferred to a VMS environment for execution, nor would storing it at the UNIX environment make any sense. The transparent name space could come into play between the different clusters that access the same file system.

Whenever a process on the list is executed, a search is conducted by the cluster server to locate the best site within the cluster to execute the process. In this way, a client may have its load distributed across several machines. Processes that need to be executed outside the cluster are transferred to the required cluster. Processes are executed remotely via a remote shell process (*rsh*) on the target machine.

Some UNIX based operating systems allow a remotely executed process to be evicted during its execution and continued on another machine. An example of this is Condor [31], which creates a shadow process on the source client to keep track of process state during remote execution. At certain checkpoints, the state of a process is sent back to the source client. The *Utopia* system uses an initial placement policy, since its processes cannot be moved in the midst of execution.

One restriction of *Utopia* on the operating system is that the file name space must be uniform throughout the system. Therefore, devices local to a machine, such as local boot disks containing similarly named files, cannot be part of the global file name space. Also, as noted above, some files will not be meaningful to all systems, since different operating systems may interpret files in different manners.

A command interpreter, or shell, on Sprite which employs user files to specify migratable processes is *msh*, a version of *tcsh*.[1] Each user maintains a local file of process names. If that file does not exist, a global system file is used. These files are called *export* files. The processes that are selected for remote execution depend upon the export policy chosen by the user.

Each user is allowed to specify one of four different *export policies*:

- Migrate all processes,
- Migrate only those processes that are not listed in the export file,
- Migrate only those processes that are backgrounded and are not listed in the export file, and
- Migrate only those processes that are not in the export file when the load is greater than a user supplied threshold.

Given that most processes in Sprite can be migrated, the processes listed in the export files are those that are highly interactive, such as the text editor or the command interpreter, or processes that would fail if executed remotely, such as the X server. Only one export file is used, though it would seem more beneficial to allow users to access both the local and global file.

When the shell is initiated, process names are read in from the export file and inserted into a hash table. Whenever the shell initiates a process, the hash table is searched to see if there is an entry that matches the process name. If no match is found, depending upon the export policy, the shell may send a request to the global migration daemon to execute the process remotely. The

[1]tcsh is a version of the C shell with file name completion and command line editing.

load average daemon searches for an idle client upon which the process can be executed.

One of the problems with this method of process selection for remote execution is the involvement of the user. When processes are executed for the first time, the user must update his list so that the process executes locally or remotely as appropriate. If a process changes characteristics the user must update his export file. If the list is not properly maintained, the benefits of this system are lost.

Another method for selecting processes to execute remotely is to estimate from past performance whether the process should migrate. There are two problems that must be considered when using past performance as a guide to the migratability of a process. What characteristics should a process exhibit to be eligible for remote execution, and what data is available from previous executions of the process?

In *History*, proposed by Svennson [32], the amount of CPU time necessary to complete execution is used to determine whether a process should execute remotely. This approach is based on the assumption that the greatest cost associated with executing a process remotely is the amount of CPU time necessary to send the process to the target client. Upon completion, the amount of CPU time used by the process is recorded, and averaged in with all previously recorded times.

One variable in this system is the percentage of process eligible for remote execution. All processes are entered into a list, ordered by their execution time. In this manner, a certain percentage of processes, based on the time of execution, can be selected. Svennson defines a *filter factor*, which has a value between zero and one. All processes in the list below the percentage defined by the filter factor are executed locally while those above it are scheduled for remote execution. In a network with a large number of long running processes, the filter factor may be low. Alternatively, if the system administrator knew that the majority of processes executed were short lived, and therefore, not worth the trouble of invoking remote execution, the filter factor could be set to a high value.

Svennson states that some factors have no significant effect on the performance of *History*. We agree that the factors he describes, such as which user, number of input parameters, and the time of day of execution, are relatively insignificant. Svennson's simulations also used a distributed file system, where processes that execute remotely have the same file access time as processes executed locally. However, he does not take into account the cost of transferring information to or from the source client. This can occur whenever a process has interaction with user devices, such as the keyboard or display, or when system calls that must be sent back to the home machine occur, such as IPC.

2.3.2 Deciding Which Processes Can Execute Remotely

The optimal solution is to know exactly which processes will benefit from remote execution each time those processes are executed. A slightly less optimal method is to rely on the user to determine which processes are migratable, and to maintain a database of this information. However, since most users are not likely to know which processes may benefit from remote execution, or even which processes they may be using during any given day, the optimal method is unobtainable. Keeping track of system statistics is useful, but if CPU time is the only statistic recorded, there is no accounting for the amount of information that must be transferred between the source and target clients.

Such information may determine whether a process will benefit from remote execution no matter how long it takes to complete execution.

Combining the two methods would allow users to specify which processes should be executed locally, while letting the system automatically determine which processes should be executed remotely based upon their past performance. The more frequently a process is executed, the more accurate the average measurements of its performance will be. There are two exceptions: process that use different resources based on input parameters, and processes that, when recompiled, change characteristics drastically. With just these exceptions, these measurements would be as accurate as a user maintained file that is always up to date.

Chapter 3

Adding Automatic Load Balancing to Sprite

The existing load balancing policy in Sprite is a centralized version of the *Sender-Initiated* algorithm. The global load average daemon maintains a list of idle workstations ordered by the amount of time that they have been idle. Whenever an idle site is needed, the global daemon is invoked, and the site that has been idle the longest is selected as the target client. Using *msh*, the processes that are chosen for remote execution are those not listed in the export file.

We have modified the *msh* shell so that historical information is used to determine whether a process should be executed remotely. The additional information that is required to make a decision is the number of system calls that must be handled on the source client, and the number of accesses to devices local to the source client. For example, in Condor [31], the number of checkpoints that need to be sent to the home machine is used as a measurement to determine whether a process should be executed remotely. This information can be obtained by tracking the number of times these specific calls are made.

By combining different techniques, we believe that we can provide a better decision making process for determining which processes should be executed remotely. In addition to being able to state which processes should be executed locally, we can use historical information to determine which processes will be the best candidates for remote execution. Another benefit is that there is no requirement for constant user involvement.

For the sake of simplicity, we have chosen to implement an initial placement policy instead of using the full capabilities of process migration. By using an *initial placement* policy we lose some of the advantages of load balancing but we decrease the overhead of process migration. The minimum amount of time necessary to migrate a null process on a SPARCstation 1 (4/60) running Sprite is 76 milliseconds [33]. This cost increases for each page and file block that must be written back to the file system, as well as for all open file descriptors that must be migrated with the process [33]. By migrating processes only upon invocation, we limit the cost of migration to a minimum. We discuss extending this work to include process migration in §5.3.

Implementing automatic load balancing on Sprite requires both a load balancing policy to decide when and where load should be distributed, and a policy to decide which specific processes should be migrated. The existing load average daemon on Sprite interacts with local daemons to keep track of the load of each site, and whether it is idle. Whenever a request to migrate a process is received from a client, the global load average daemon finds an idle site, if one exists, for remote execution. This is a *Sender-Initiated* algorithm with a central server determining the best target client based on load information. The current method used to decide which processes should be migrated relies on the user keeping track of all processes that should not be migrated. What is needed is a method which removes the responsibility of selecting processes from the user, but provides results better than the current method. We have modified both the kernel and the user shell in order to implement such a system.[1]

Several modifications to the shell were required. We modified *msh* [10] so that it maintains a database of the past performance of processes based on data that is returned to the shell from the kernel. The shell decides whether a process should be executed remotely based on both this history and a user supplied migration policy (see §3.3) similar to the existing *msh* export policies.

Compared to the modifications to the shell, only a few modifications to the kernel were necessary to implement our process selector on Sprite. The only information needed from the kernel is resource usage. Any process that accesses a local device or makes a system call that must be executed on the source client must have that access recorded. The kernel must keep track of these values for the current process and all of the current process's children.

All of this statistical information must be propagated up through the kernel so that it is available to user processes. Two different statistics are recorded under different circumstances. System call usage is recorded whether the process is executed locally or migrated, since these calls can be trapped and recorded no matter where they are executed. Local device usage is only recorded when the process is executing locally, since the device is no longer *local* when the process is executing remotely. The only responsibility of the kernel is to make this information available to the shell. We let the shell take care of the entire decision making process.

[1]All changes were made according to the *Sprite Engineering Manual* [34].

3.1 Modifying the Shell

Since *msh* [10] was available to us and included several of the functions that were needed, we started with this version of the shell as our foundation. We want to allow for the user and the system to specify processes that should never be migrated, so we have kept export files in our system. However, since all processes listed in the export files are *never* migrated, we now call these files the *restrain* files.

Instead of checking only one restrain file, both a global and local file are used to specify which processes must be executed locally. The global file is provided by the system administrator and contains a standard list of processes, commonly executed by the users of the system, that should not be migrated. The local file is provided so that experienced users can provide their own list of processes that cannot be migrated. Another file, called the *history* file, has been added that maintains the history of all processes that are not in the restrain files. This file, like the local restrain file, is located in the user's home directory.

When the shell is initiated, the process names from the restrain files are read in first, and tagged as being nonmigratable. The entries in the history file are then loaded and inserted into the same hash table as were the entries from the restrain files. Precedence is given to entries in the restrain files. The only way a process can exist in both files is if the process has just been added to one of the restrain files. In this case, when the entries in the history file are entered into the hash table, the history of the superfluous entry is discarded.

Each entry in the hash table that is tagged as migratable uses three different fields to store information: CPU time, system calls, and local device usage. The CPU time is used to determine whether a process executes long enough to merit the cost of migration. As stated, the time necessary migrate a null process on a SPARCstation 1 workstation is 76 milliseconds. A process that does not execute for at least 1 second will experience a minimum 8% increase in execution time. A process that does not execute for more than 100 milliseconds could have its execution time more than doubled if it is migrated. The CPU time is also used when determining if the number of local system calls or local device usage will degrade the process's performance.

System calls that must be handled by the source client can limit the benefits of remote execution. If a large number of calls must be sent to the source client, the CPU time must be proportionately larger to offset the time needed for these remote procedure calls. Condor [31] uses a similar measurement to determine the benefits of remote execution. For any process, they compare the amount of CPU time with the number of checkpoints that are sent to the source client. The result is the *leverage* of the process. A large leverage indicates that the number of checkpoints are small compared to the amount of CPU time consumed. Processes with a small leverage should not be executed remotely. A checkpoint involves a great deal more overhead than the local system calls executed on Sprite, since it is used to keep track of process state information during execution. These checkpoints are used by the system to relocate a process during execution and start the process on another workstation.

There are two types of local system calls that are used in our implementation. The first is comprised of any call to the file system which accesses a device or service local to the source client. These devices and services are accessible through the file system using the pseudo-file mechanism in Sprite [9]. Any device mounted as a pseudo-file can be accessed as a file, with normal read and write operations. Any access to a local device is recorded by the kernel, but it is only relevant when the process is executing locally, since a process executing remotely has a different set of devices that are *local*. We refer to the number of accesses to local devices as the *local device usage*.

The second type of local system call is comprised of all calls that must be sent back to the source client that do not involve the file system. These system calls can be recorded whether the process is executing locally or remotely, since the call can be trapped regardless of execution location and recorded before it is executed. In comparison, checkpointing in Condor can only be recorded when the process is executing remotely, since checkpointing is not used for local execution. We refer to the number of system calls that must be handled by the source client as the *local system call usage*.

When a process is executed, the shell forks off a child process and waits until that process completes. Either during or upon completion, some information from the kernel is available to user processes. The getrusage system call returns the kernel statistics of an executing process. A user process can use this call to obtain a partial history of a child process. The wait3 system call returns the exit code of a child process, as well as the resource usage returned by getrusage.

The only time the performance of a process is recorded is when a normal exit code is received. The history of processes that are interrupted or exit with an abnormal code is discarded. The impact of this policy is that all programs must exit properly (e.g. programs written in **C** must exit with a value of 0). A future implementation may include a method of recording information from abnormally terminated processes.

We have augmented getrusage and wait3 so that both return the number of local system calls and local device usage. The CPU time was already returned. Currently, only wait3 is used for examining the performance of a process. When the child exits, the shell uses wait3 to obtain kernel statistics. These statistics are averaged in with the previous statistics to provide a sample mean of the performance of the process. These statistics are not recorded for those processes that are marked as nonmigratable.

Finally, we must make these statistics available to other invocations of the shell. In our implementation, the statistics are written to the history file whenever the shell is terminated. An addition we would like to add is to periodically save the information. Shells read in the previous statistics whenever they are initiated. Statistics are not shared between concurrently running shells.

We had to take special care when writing statistics in order to avoid overwriting new data. When two or more processes try to write to the history file, there must be some locking mechanism to prevent concurrent access. When the history file is opened, we use an advisory lock to determine if another process is using the file. If it is in use, the process waits until the file is unlocked. Since access to the file is protected by this lock, there is no problem with concurrent access.

Any values accumulated by one shell are not available to any other shell that is executing concurrently. If the same process is executed in different shells, the only way for its history to coalesce into one record is to merge all of the history in each shell together. Since overwriting the existing history file may overwrite statistics recorded by another shell, we had to find a method whereby all of the statistics could be gathered together. Each shell has a record of which statistics were previously recorded and which statistics have been recorded during the current execution. When the shell gains access to the history file, the statistics in the file are averaged in with the statistics that have been recorded during the current execution. The updated statistics are then written back to the history file. When a process writes to the history file, the recorded statistics for the current execution are set to zero. These values are no longer needed since they are now accounted for by the previously recorded statistics.

In the current implementation, there are two different methods for updating the history file. One method implemented is to exit the shell. Since this is a costly way of updating the history file, we have also included a programmer's toolkit which allows the informed user to write the statistics to the file (see appendix A). A reasonable solution would be to update the history file periodically. We plan to include this change in a future implementation, but due to time constraints, we were unable to add it to our current version. The proposed method is to set a timer based on an environment variable so that the shell updates the history file at each time interval.

3.2 Kernel Modification

Our design goal for modifying Sprite's kernel was to keep changes to a minimum. Any changes we made had the possibility of affecting the entire system, and debugging the entire kernel would be an enormous task. Also, we did not want to degrade the performance of the kernel. Large additions of code could slow down execution. The only changes necessary were adding the local system calls usage and the local device usage fields to the process control block, making certain that these fields were incremented at the appropriate times, and returning these values to user processes.

Every system call has two functions that it uses depending upon whether it is executing remotely or locally. In most cases, both of the functions are the same. However, those system calls that must be returned to the source client call a different function if the process is executing remotely. This function uses a kernel-to-kernel remote procedure call (RPC) to execute the needed function on the source client. Each time one of these system calls is made, the local system call usage is incremented. Since these calls are trapped whether they are executed remotely or locally, the local system calls are always recorded.

The only local system calls that are not recorded in the manner described above are calls that access devices, pipes, or pseudo-devices local to the source client. A pseudo-device is an extension to the file system that allows user processes and services to be accessed remotely as it were a device [9]. These system calls must be trapped by the file system. This allows processes to access them as if they were streams, using normal read and write operations. When a read or write operation is performed, if the stream accessed is a local device, pipe, or pseudo-file, local device usage is incremented.

The only problem with trapping this call in the file system is that this data is meaningless when the process is executing remotely. The previously local streams are now accessed as remote streams; in our implementation, the kernel cannot distinguish remote calls to different hosts. Since we cannot know whether there are any other remote streams being accessed by the current process, we must discard this statistic whenever the process is migrated.

Returning the statistics to user processes requires a number of small modifications to the kernel. When a process control block is first created, the additional fields must to be initialized to zero like the rest of the structure. When a child process completes execution, the values of the local system call usage and local device usage fields have to be added to the parent's fields. When either getrusage or wait3 are used, the parent process that made the call must locate the child's process control block. The pertinent data must be transferred to a different memory location so that the security of the kernel is not compromised. Once all of the has been done, the system call returns the data from the kernel to the parent process.

3.3 Migration Policies

We have built three different metrics that reflect the past performance of a process. Users can use any combination of one or more metrics, as well as using no metric at all. These metrics allow experienced users to design a migration policy for their needs. A system administrator can set the default policy based on the requirements and resources of the system. Different policies can have different effects on the processes that are migrated.

The three different metrics are CPU usage (*Min_Time*), the ratio of CPU usage to local system call usage (*Sys_Call*), and the ratio of local device usage to CPU usage (*Dev_Ratio*). In addition to these metrics, there are two additional policies which can be used: migrate no processes, and migrate all processes not listed in the *restrain* files. The reason for the different metrics is to give options to systems with different needs.

If the cost of a local system call is high, then the *Sys_Call* metric should be used. For example, every time a process uses the gettimeofday system call, the call must be sent to the source client via kernel-to-kernel RPC. A large number of these calls will degrade the completion time of a process. If the CPU usage is large compared to the number of system calls, then the kernel-to-kernel RPC overhead will be negligible. Therefore, a ratio is used to determine whether the number of local system calls will affect the performance of a process executed remotely. The number of local system calls is recorded whether the process is executing locally or remotely. Each time a process exits normally, the number of system calls made is averaged in with the previous value. Thus, if the number of local system calls varies between executions, it may be noted after execution is complete whether any benefits were received by migrating the process.

When this policy is used, a process is eligible for remote execution if the ratio of the average CPU time to the number of local system calls is greater than the *Sys_Ratio* threshold. We have observed that all processes make a minimum of two system calls that must be executed locally. In our implementation, we account for these two calls. In other words, if the number of local system calls is two, we make a comparison as if the number were zero, in which case, the process would pass this metric.

If the cost of accessing devices local to the source client is high, then the *Dev_Ratio* metric should be used. In Sprite, this metric is not as great a factor as the *Sys_Ratio* metric. All reads and writes to local streams can be buffered so that the actual time spent accessing them is minimal. Also, the statistics returned for local device usage are only valid when the process executes locally and exits normally.

If the process is executed remotely, there is no method by which we can be certain if any benefit was gained from remote execution using the *Dev_Ratio* metric.[2] The local device usage may be different for the remote execution based on different input parameters.

When this policy is used, a process is eligible for remote execution if the ratio of the average number of accesses to local devices to the average CPU time is

[2]This problem can be translated into the *Two-Armed Bandit Problem* [35, 36], where one arm has a positive or negative value based on the gain or loss from remote execution, and the other arm has a value of zero.

less than the *Dev_Ratio* threshold. If the number of local devices accessed by the program is zero, then the process passes this metric. This can occur in any program that has no I/O associated with local devices.

The final metric is execution duration for the process. A process that executes for a short amount of time will suffer tremendously from process migration. The execution time of a short-lived process is greatly affected by the overhead of migration, while a long-lived process does not experience any significant penalty. The metric used to prevent short-lived processes from migrating is *Min_Time*. Any process that has an average execution time greater than the *Min_Time* threshold is eligible for remote execution.

This last metric is similar to the method used in *History* [32]. The difference is that *History* only migrates the top percentage of processes, whereas our system migrates all processes that execute longer than the specified threshold. We feel that our method is more robust as the processes migrated will not depend upon the number of short-lived or long-lived processes that have been previously executed.

3.4 Other Considerations

Processes that are eligible for remote execution are not always migrated to another site, as an idle site may not be available. However, the shell does not know whether a process executed remotely, locally, or was evicted from a target client during execution and sent back to its source client. The local device usage statistics are disregarded if the metrics used indicated that the process could have migrated.

If a process is eligible for remote execution, it may never be executed locally.

The only problem that this creates is that the performance of such a process may change so that it is no longer eligible for migration. This can happen whenever a process is modified so that it accesses local devices more frequently than before. Such a change may cause unwanted performance degradation until the process executes enough times to change the average characteristics to the proper values. At this point, we only have a few ideas on how to fix this problem. One of them is to keep track of the date of creation of the file, and dispose of the process's history whenever it the date has been changed. However, since local device usage plays a small role in Sprite, we do not believe that it is a serious concern at this time.

Chapter 4

Measurements

We use several different types of programs to measure the effectiveness of our implementation. Some are CPU intensive, while others are file intensive, and the remaining few are examples of processes that should never be migrated. Our network consists of three Sun SLCs (4/20) and one Sun IPC (4/40) which is the file server. This is much smaller than the Sprite network at University of California, Berkeley. All programs were executed on Sun SLC's (4/20), ten times in rapid succession and measured for their CPU usage as well as their overall time. Migration to the IPC was disabled so that differences in computational power were not a factor. The file system is distributed throughout Sprite, so we did not want file access to be part of our measurements Each program was executed once before the measurements were taken, so that any files needed by the programs would already be resident in local file cache. No other user processes were executed on the machines while data was gathered.

Table 4 gives a comparison of both the average CPU time and the average execution time needed by the process. The latter measurement is the time from invocation to completion. The final two columns give a percentage speed up based on the ratio of the time for the process when migrated to the time when the process was executed locally. In all cases, the speed up for the average CPU

Table 4.1: Comparison of local execution versus remote execution

	Local		Migrated		Speed Up	
Process	Average CPU time	Average Exec Time	Average CPU time	Average Exec Time	CPU	Real
LaTeX	5.89	6.34	5.73	7.28	2.8%	-12.9%
compression	3.01	3.05	2.91	3.01	3.4%	1.3%
decode	3.23	3.26	3.28	3.31	-1.5%	-1.5%
consw	14.83	14.90	14.65	14.83	1.2%	0.5%
ls	0.070	0.090	0.130	0.210	-46.1%	-57.1%
cat	0.054	0.064	0.086	0.116	-37.2%	-44.8%
gettime	0.218	0.260	0.924	4.07	-76.4%	-93.6%

usage was greater than the total execution time. This can be attributed to the overhead locating an idle client to execute the process on, and the actual process migration. The main point of this table is to indicate which processes should never be migrated. Those processes that experience a speed up that is slightly greater than or less than zero are good candidates. Processes that with a large negative speed up should never be migrated.

The file that was compiled by LaTeXwas small (30 Kbyte). The programs compression and decode are arithmetic encoding algorithms. The former took a 16 Kbyte file and compressed it using Laplacian Estimation [37]. The latter used the compressed file and generated the original file from it. The program consw uses a large number of reads and writes to local devices to determine the context switch time (given in Appendix B.1), and ls and cat are the standard UNIX utilities. Finally, gettime is a program that calls gettimeofday two thousand times.

As predicted, CPU intensive programs, such as compression and decode, completed execution in roughly the same amount of time. A more exhaustive test should show that there is no difference between local and remote execution.

The discrepancy in total execution time for $\LaTeX$ is not apparent. One observation that was made concerning this program is that there were 7 local system calls made during execution, rather than the 2 to 4 calls that most programs make. By table 4 (explained below), the average time for a migrated gettimeofday system call is 2 milliseconds greater than when it is executed locally. This does not explain the additional time taken when the process is migrated.

Both compression and decode passed all metrics that were put to them, since the average number of system calls was 2, the average number of local devices used was 0 – the programs have no output – and the average CPU time was high (the program executes in excess of three minutes when the input is 512 K-bytes).

Another anomaly was the execution time of consw. When executed locally, over $60,000$ accesses were made to a local pipe. However, when executed remotely, there was no loss of execution time. We attribute this to the fact that the program opens up a pipe local to the remote machine when migrated. If a process must communicate through a pipe to another process (IPC), then we would expect an increase in execution time when the process is migrated to communicate with the (now) remote pipe. The *Dev_Usage* metric was not a factor here since the devices local to the source client were not accessed from the target client. A possible detection of this type of occurrence may be to measure the local device usage when the process executes remotely.

The ls and cat system utilities are short programs that take much longer to execute remotely. The overhead to migrate these processes is much greater than the amount of time that the processes actually execute. In this case, using the CPU metric would cause these processes to be executed locally and, therefore, not degrade their performance time.

Table 4.2: Comparison of gettimeofday calls executed locally and remotely.

Location of Execution	Time for 1 call	Time for 1,000 calls	Average time for call
local	30 msec	140 msec	110 μsec
remote	94 msec	2,250 msec	2,160 μsec

An example of a program that should always be executed locally is one that makes excessive local system calls. Our gettime program is such an example. To measure the average amount of time taken by a gettimeofday call, we took the average execution time of a program that called gettimeofday one time. Using this as the overhead of executing the instruction, we executed another process which called gettimeofday 1001 times. Subtacting the difference in times and dividing the result by 1000 gave us the average time for this system call. Each program was executed twenty times to obtain the average value.

By table 4, we see that the average time taken to execute a gettimeofday system call when executing locally is 110 microseconds, whereas the time needed to execute it remotely is over 2 milliseconds. We have not measured the other system calls that must be executed locally, but if the time of execution for each call is comparable to the time for gettimeofday, then any program that uses local system calls to this extent must be forced to stay local to avoid increased execution time of this magnitude.

A more complete test would measure the overall throughput on a system running just *msh*, or just *History*, and comparing that with the throughput of our system. We do not have a large group of users on our Sprite network, nor do we have a large number of workstations running Sprite. This test would need to be run in an actual user environment, such as the Sprite network at University of California, Berkeley. This test would show whether users were maintaining their export files adequately, in the case of *msh*, and whether the filter factor in *History* was set to the proper value.

One problem that has plagued development of this project from the start is the unreliability of the Sprite log file system (LFS) [38]. When we started our final series of tests, we began to experience failures in the file system. The source code for the kernel was not always available because of the file system failures, and therefore, recompilations took an excessive amount of time. Because of this, we have not been able to take proper measurements on the effects of local device usage and the number of local system calls.

Even without these two metrics, we believe that our automatic process selector is an improvement over *msh* and *History*. Short-lived processes which would be migrated in *msh* are forced to execute locally if the *CPU_time* metric is used. Long-lived processes that cannot be migrated because they access private local devices, or when the user is aware of the excessive number of local system calls and device access, can be forced to execute locally instead of remotely, as they would in *History*. It remains to be seen how much of an advantage the other two metrics are.

Chapter 5

Future Research

As with any project, several ideas have come to light that may improve the performance of our system but have not been implemented. Some of these did not seem to provide additional benefit, while others were not implemented due to lack of time. In this chapter, we describe some of these ideas and our intuition as to whether their implementation would be beneficial.

5.1 Enhancing Process Identification

Currently, a process record stored in the history file has no notion of the process's path. Processes that have the same name, but are located in different directories are recorded as if they were the same process. Adding the path name to the process entry would provide better process identification. An additional benefit of this modification is that different history files could be merged together to provide a more complete history.

If the creation date of a program is stored in the history file, then when a program is modified the shell could disregard the old statistics and start anew. In this way, the history would be more adaptive to a program's changing needs, since a modified program may have completely different resource requirements.

A drawback of this implementation is that minor modifications to a program that do not affect the resource requirements or execution time would cause the previous history to be lost.

5.2 Rigorous Statistics Acquisition

By gathering more data, we may be able to make a better decision as to whether a process should migrate, or as to what location it should migrate. Trade-offs between the amount of space needed for the statistics and the benefits gained must be considered, as well as the increased level of complexity and potential performance degradation.

Modifying the file system so that it keeps track of every single device accessed by a process would give a clearer indication of the optimum site of remote execution. Because Sprite's file system is distributed over the network [6], migrating a process to the file server of the needed file domain could provide better execution time. We could determine which file server received the greatest number of accesses and, therefore, which client would be the best site for remote execution.

We chose not to implement this for two reasons. First, maintaining an array of all device accesses when the number of clients in the system is large (> 100) would increase the size of the history file to an unmanageable degree (from 40 bytes per process entry to 400 bytes per process entry). Using our current design, this array would also be required by the kernel, increasing the size of the kernel structures enormously. Another disadvantage would be overutilization of those workstations that host the file system, while other workstations would be underutilized. We believe that the overall benefit of this modification would be small.

By examining the file system calls carefully, it may be possible to determine which hosts are being most utilized. While this may give a good indication of where to migrate the process, it may also be used to determine what the local device usage is when a process is executing remotely. Such knowledge would increase the accuracy of our automated process selector. No longer would the system have to guess whether a benefit accrued from remote execution based on the *Dev_Ratio*. The statistics could be limited to the number of accesses to devices local to the source client (as it is now), or could be as extensive as an array of all clients with all device accesses. The latter implementation, as noted above, would use an excessive amount space. It remains to be seen whether this modification can be added to the kernel.

5.3 Load Balancing

We have implemented a load sharing program with automatic process selection. Because we are using an initial placement policy, we do not keep track of a process during its execution, nor are processes relocated during execution except when they are evicted from a foreign site. A load balancing system would relocate processes during execution to other clients if the local client became heavily loaded. This would include moving already migrated processes.

Load balancing would entail increasing the number of migrated processes that can be executed on a client. The current limit of two foreign process per workstation, one of low priority (backgrounded) and one of higher priority, is too low. The difficulty of having a large number of foreign processes on a workstation is Sprite's concept of user-owned workstations. Whenever a user accesses the workstation so that it is no longer idle, all foreign processes are evicted and sent to their home machine. Even though the global load average daemon could be invoked to find suitable hosts for these processes, the system will never be balanced so long as a user may refuse migrated processes. Because one of the tenets of Sprite is that each user has the power of at least one workstation, complete load balancing will have to be a secondary consideration.

Another component of the load balancing algorithm used in Sprite that has not been implemented is the *threshold* value. There must be some minimum load at a workstation before processes are migrated. Setting this value at zero causes an excessive number of processes to be migrated where the benefits of such migration are limited. By only migrating processes when the load on the local workstation is high, we can improve the chances that this migration will improve the overall response time of the system.

Another modification necessary to implement load balancing is migrating processes during execution. To accomplish this feat, a supervisor process must monitor the load on each workstation and know which process should be migrated in the case of an excessive load. The supervisor could choose the process at random, but it would be more useful if some information to make this decision were available. If the supervisor has access to the history file generated by the shell, then the information used would be the same as that used for the initial placement.

The supervisor process could be implemented as a local daemon that runs on each workstation, which is active whenever the load surpasses a specified threshold. However, there is a problem in deciding which history file should be used. Since each user has their own private history file, the daemon should not be bound to a specific user's history. One possible solution is to load each user's history file into memory, and augment each table by the user's id number. When the supervisor needs to determine if a process should be migrated, the user id is used to access the correct history, and then the past statistics of the process are accessible. Another solution is to merge all history files into one history file usable by the daemons. This would require that each entry in a user's history file has a path name associated with it. In both of these methods, the supervisor would need to update its history periodically from each user's history file.

A third solution that involves less file access is to use getrusage. The supervisor process could note the current statistics of all executing processes using this system call and determine from that information which process is the best candidate for migration. This method would be less accurate than the other two methods, as the execution pattern of a process may change during its execution.

5.4 Increased Testing

We need to test our automatic process selector in a more active environment. The limitations of our environment have contributed to our limited results. A more exhaustive sequence of tests needs to be performed on an environment with numerous users and clients. Measurements would indicate the overall performance of our system compared to *msh* and *History*. Also, the benefits of the measuring and using the number of local system calls and number of accesses to local devices needs to be determined. As it stands, we could not determine any correlation between the amount of local device use and the execution time when the process was executed remotely. We hope that in the future, we can test the performance of our system on a larger community of Sprite users, such as the group at University of California, Berkeley.

5.5 Other Modifications

Currently, the statistics of processes that terminate with an abnormal exit code are not recorded. In some cases, this might be beneficial, such as when a process is interrupted. However, there are processes that do not have normal termination methods, and thus, they must be interrupted. Such processes will never be chosen for process migration since no history is recorded for them. If a method for determining the cause of termination could be implemented, it may be possible to have a special class of processes that have their history recorded when they are interrupted.

The one modification that should be made before this system is put into general usage is history file recording. Since most users of Sprite have their own workstation, it is unreasonable to expect that they will exit the shell each time that they end a session. More likely, the user will leave the shell running continuously. Since the goal of this system is to remove users from the decision making process, requiring them to write back the history file is stopping short of our designs. We need to add a periodic update that forces a write back to the history file. The period for this update could range from thirty seconds to ten minutes.

Another question that arises is the impact of the modifications to the kernel. Though the modifications were kept simple to avoid adding overhead time, a comparison should be made between execution time of processes under the old kernel, and execution time of processes running our kernel. Also, we must consider what effect the advisory lock may play in slowing down the shell in the case of multiple accesses to the history file, though we expect that this would be small.

Chapter 6

Conclusion

We have implemented an automatic process selector that can be used in conjunction with an automatic load balancer to determine which processes are the best candidates for remote execution. The decision to migrate a process is based on its past performance. Several metrics have been implemented to allow for several different criteria when deciding which processes should be migrated. Although our automatic process selector has not been completely debugged, it still provides results that are theoretically better than *msh* and *History*, two other methods used to select processes for remote execution. More rigorous testing would provide a more definitive answer. Another benefit of our system is that it reduces the amount of user intervention necessary to maintain proper statistics, as in *msh*.

There are definite advantages to working with a well organized kernel. Modifying the Sprite kernel was a pleasure, as the arrangement of the modules and the documentation of the code was straightforward. On the contrary, modifying the shell was more complex and most of our debugging time was spent in the shell, rather than the kernel. If enough time were permitted, a new shell could be built based around the process selector. A communication channel could be set up between all invocations of the shell on the local machine and a load balancing daemon so that the daemon would have access to the statistical records of the shell. An informed decision could be made by the load balancing daemon as to which process should be migrated to balance the load without reducing the throughput of the system.

A limitation of our system is that it relies on an *initial placement* policy instead of utilizing the full capabilities of *process migration*. A true load balancer cannot be implemented until we add an additional program which selects processes to migrate during their execution. The benefits of such an implementation remain to be seen, though Eager *et al.* present evidence that the benefits of process migration are small under normal circumstances [39].

Our system as it is can be used on distributed operating systems that do not have process migration, and only assumes that any process can be executed remotely. We feel that having a general purpose program is highly useful and may be incorporated in future operating systems in addition to Sprite.

References

[1] J. K. Ousterhout *et al.*, "The Sprite network operating system," *IEEE Computer*, vol. 21, pp. 23–36, February 1988.

[2] F. Douglis and J. Ousterhout, "Transparent process migration for personal workstations," Tech. Rep. UCB/CSD 89/540, University of California Berkeley, California, November 1989.

[3] M. Mutka, "Estimating capacity for sharing in a privately owned workstation environment," *IEEE Transactions on Software Engineering*, vol. 18, pp. 319–328, April 1992.

[4] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Transactions on Software Engineering*, vol. 32, pp. 85–93, January 1977.

[5] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[6] B. B. Welch and J. K. Ousterhout, "Prefix tables: A simple mechanism for locating files in a distributed file system," in *Proc. of the 6th International Conference on Distributed Computing Systems*, pp. 184–189, May 1986.

[7] F. Douglis and J. Ousterhout, "Transparent process migration: Design alternatives and the Sprite implementation," *Software–Practice & Experience*, vol. 21, pp. 757–785, Aug. 1991.

[8] B. B. Welch and J. K. Ousterhout, "Pseudo-devices: User-level extensions to the Sprite file system," in *Proc. of the 1988 Summer USENIX Conf.*, pp. 184–189, June 1988.

[9] B. B. Welch and J. K. Ousterhout, "Pseudo-file-systems," Tech. Rep. UCB/CSD 89/499, Univ. California Berkeley, April 1989.

[10] A. Ho and F. Meyer, "A migration shell for Sprite." Submitted for CS 262 at University of California, Berkeley, May 1990.

[11] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: A load sharing system

for large, heterogeneous distributed computer systems," Tech. Rep. CSRI-257, University of Toronto, November 1991.

[12] Y. Artsy and R. Finkel, "Designing a process migration facility: The charlotte experience," *IEEE Computers*, vol. 22, pp. 47–56, September 1989.

[13] D. R. Cheriton, "The V distributed system," *Communications of the ACM*, vol. 31, pp. 314–333, March 1988.

[14] E. Zayas, "Attacking the process migration bottleneck," in *Proc. of the 1986 fall joint computer conference*, pp. 1–23, IEEE, May 1986.

[15] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 8, pp. 279–301, July 1989.

[16] F. C. H. Lin and R. M. Keller, "The gradient model load balancing method," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 32–38, January 1987.

[17] T. L. Casavant and J. G. Kuhl, "Analysis of three dynamic distributed load balancing strategies with varying global information requirements," *Proc. of the 7th International Conference on Distributed Computing Systems*, pp. 185–192, August 1987.

[18] S. H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Transactions On Software Engineering*, vol. SE-7, pp. 583–589, November 1981.

[19] S. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan, "Analysis of a graph coloring based distributed load balancing algorithm," *Journal of Parallel and Distributed Computng*, pp. 160–166, October 1990.

[20] D. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance Evaluation*, vol. 6, pp. 53–68, March 1986.

[21] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 662–675, May 1986.

[22] C.-K. Chang, "Bidding against competitors," *IEEE Transactions on Software Engineering*, vol. 16, pp. 100–104, January 1990.

[23] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Transactions on Computers*, vol. 38, pp. 1110–1123, August 1989.

[24] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "Analysis of the effects of delays on load sharing," *IEEE Transactions on Computers*, vol. 38,

pp. 1513–1525, November 1989.

[25] L. M. Ni, C.-W. Xu, and T. B. Gendreau, "A distributed drafting algorithm for load balancing," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1153–1161, October 1985.

[26] F. Bonomi and A. Kumar, "Adaptive optimal load balancing in a nonhomogeneous multiserversystem with a central job scheduler," *IEEE Transactions On Computers*, vol. 39, pp. 1232–1250, October 1990.

[27] A. Thomasian, "A performance study of dynamic load balancing in distributed systems," *IEEE*, pp. 178–184, August 1987.

[28] A. Haĉ and T. J. Johnson, "Sensitivity study of the load balancing algorithm in a distributed system," *Journal of Parallel and Distributed Computing*, pp. 85–89, October 1990.

[29] A. Haĉ and X. Jin, "Dynamic load balancing in a distributed system using a sender-initiated algorithm," in *Proc. of the 7th International Conference on Distributed Computing Systems*, pp. 170–177, September 1987.

[30] J. Gait, "Scheduling and process migration in partitioned multiprocessors," *Journal of Parallel and Distributed Computing*, pp. 274–279, August 1990.

[31] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor – a hunter of idle workstations," in *Proceedings of ACM Computer Network Performance Symposium*, pp. 104–111, June 1988.

[32] A. Svensson, "History, an intelligent load sharing filter," in *Proc. of the 10th International Conference on Distributed Computing Systems*, pp. 546–553, 1990.

[33] F. Douglis, *Transparent Process Migration in the Sprite Operating System*. PhD dissertation, University of California Berkeley, California, September 1990.

[34] J. K. Ousterhout, "Sprite engineering manual." Programming and documentation conventions for the Sprite Operating System.

[35] D. A. Berry and B. Fristedt, *Bandit Problems – Sequential Allocation of Experiments*. Chapman and Hall, 1985.

[36] J. C. Gittins, *Multi-Armed Bandit Allocation Indices*. John Wiley & Sons, 1989.

[37] J. Glen G. Langdon, "An introduction to arithmetic coding," *IBM Journal of Research and Development*, vol. 28, pp. 135–149, Mar. 1984.

[38] F. Douglis and J. Ousterhout, "Beating the I/O bottleneck: A case for log-structured files systems," Tech. Rep. UCB/CSD 88/467, Univ. California Berkeley, October 1988.

[39] D. L. Eager, E. D. Lazowska, and J. Zahorjan, “The limited performance benefits of migrating active processes for load sharing,” in *Proc. of ACM SIGMETRICS 1988*, May 1988.

Appendix A

History Toolkit

We have modified a built in command so that experienced users can access the historical data from the shell. In *msh* [10], the export command either printed out all entries loaded from the export file, or listed the current policy and threshold values. The following is a description of the different options available using the export command.

`Export --help`: Prints out the current export policy, the metrics *Min_Time*, *Dev_Use*, and *Sys_Ratio* used in the for the different export policies, a description of the policies, and descriptions of the different options. The following is a sample message printed by invoking export with the *–help* option.

```
EXPORT HELP     current policy:  5
Min_Time:  100, Dev_Ratio:  500, Sys_Ratio:   50

Policies:
 000 (0): Don’t Migrate anything
 001 (1): Migrate if CPU > Min Time
 010 (2): Migrate if Dev_Use/CPU < Dev Ratio
 100 (4): Migrate if CPU/Sys_Ratio > Sys_Ratio
1000 (8): Migrate everything

Process statistics are used to determine whether processes
are migrated.  These statistics are kept in .tcsh.history
Options:
```

```
-help:   Print this message
-print:  Print all processes with and without statistics
-write:  Update history file with new performance statistics
-stat:   List the statistics for all processes with statistics
-stat {process_name}:  List statistics for specific process
-rm {process_name}:    Remove the named process from the
                       history file and current statistics
-read:   Read in the statistics from the history file,
         losing all new performance statistics
```

The *–read* and *–write* commands will be outdated when the periodic history file updates are implemented. From experience, it has been found that the *–print* and *–stat* options are rather unwieldy, since a large number of processes tend to accumulate in the history after just a few hours of use. This prompted the addition of a *–stat* option with a process name specifier. The *–rm* option is used to remove a bogus process name from the history file. Otherwise, if the user is aware that the performance of a process has changed drastically and doesn't want to retain the old statistics, this option can also be used to discard the old statistics.

In this example, the current policy is to migrate processes execute for longer than one second (policy 1 with Min_Time equal to 100) and if the ratio of the CPU time to the number of local system calls is greater than 50.

The following is a sample output of the *–print* option:

```
(  0)    finger  CPU=  13, S/C=  3, Times=  4, D/U= 11, Rec'd=  4
(  3)  shutdown  Not Migratable
(  5)    select  Not Migratable
(  8)    update  CPU= 154, S/C=  2, Times=  2, D/U=120, Rec'd=  1
(  9)    xclock  Not Migratable
   .
   .
   .
[ 96]+       vi  Not Migratable
[ 96]+       ls  CPU=   4, S/C=  3, Times=101, D/U=  0, Rec'd=101
```

```
( 96)        rm  CPU=   4, S/C=  2, Times= 16, D/U=  0, Rec'd=  2
( 99)   loadavg  Not Migratable
(100)        ps  Not Migratable
(101)  hostname  Not Migratable
(102)        mx  Not Migratable
(104) spritemon  Not Migratable
(105)        su  Not Migratable
(109)        tx  Not Migratable
(114)     Flock  CPU=   3, S/C=  2, Times= 11, D/U=  2, Rec'd= 11
(116)   Xsprite  Not Migratable
(117)    migcmd  Not Migratable
(122)  xwebster  Not Migratable
(124)   whereis  CPU=  36, S/C=  2, Times=  2, D/U=  1, Rec'd=  1
```

The numbers in the left hand columns are the hash values used to sort the data. This is an artifact from *msh*. The entries are ordered by their hash values. Any process that is in one of the restrain files is tagged as *Not Migratable*. The other files have abbreviated statistics given (described below). We have not tested our system for aesthetical value, so it remains to be determined if giving the statistics in this form is appreciated by the general user populace.

The statistics give, in order, are the CPU time in tens of milliseconds, the number of local system calls, the number of times these two statistics have been recorded, the number of times a local device was accessed, and the number of times that this statistic has been recorded.

The following is a sample output of the *–stat* option:

```
                     times                                  process
cpu time  Sys Calls  executed  Dev Use   Dev Rec’d   name
      13          3         4       11          4     finger
     154          2         2      120          1     update
      12          2         1       19          1     xsetroot
       4          2         1        0          1     chmod
       6          2         6       12          6     prefix
       4          2         2        0          1     mkdir
     512          4         6       41          3     latex
      28          3         2       20          2     mail
      34          3         1       71          1     chsh
     120          2         2        0          2     gcc
      12          2         1       17          1     talk
      48          2         7        1          4     grep
      70          2         3      100          1     doit
       6          1        64       31         64     more
       6          2         4        2          3     cat
    1702          2         8        2          2     compression
     156          4         1       25          1     man
       8          2         2        0          1     stat
      68          3         2        1          1     tar
      36          5         3       87          3     ftp
       4          2         8        1          2     pwd
      26          4         2        3          2     who
      24          4         2        3          2     if
      10          2         1        5          1     co
     105          2         4        0          1     cp
     122          2         1        0          1     uncompress
      68          4        24       10         24     rup
     141          2        25       30         25     du
       4          3       101        0        101     ls
       4          2        16        0          2     rm
       3          2        11        2         11     Flock
      36          2         2        1          1     whereis
```

The measurements presented here are a less abbreviated form of the statistics presented by the *–print* option.

Some processes can be moved from the history file to the *restrain* file.

For example, while this program was being tested, stty would be called every time that the shell was invoked. This generated an average CPU time of 80 milliseconds, 2 system calls (the minimum for a program), and no local device usage. It was clear that such a program should never be migrated, so it was added to the *restrain* file. From statistics printed above, and a knowledge of the programs involved, it is clear that chmod, mkdir, cat, and more should be added to the *restrain* file.

Appendix B

Sample Programs

B.1 Consw.c

```
# include    <stdio.h>
# include    <sys/types.h>
# include    <sys/times.h>

# define    LOOPS    10000

/*
** Find the context switch time.
**
** DDEL
** Thu Dec 13 14:42:49 PST 1990
*/
void main() {

    int    hither[2], yonder[2];
    int    i;
    char    puck;

    struct tms    now, then;
    int        overhead;

    (void) pipe(hither);
    (void) pipe(yonder);

    (void) times(&now);
    for (i = 0; i < LOOPS; i += 1) {
        (void) write(yonder[1], &puck, sizeof(char));
        (void) read (yonder[0], &puck, sizeof(char));
    }

    (void) times(&then);

    overhead = then.tms_stime - now.tms_stime;

    printf("Overhead per loop = %lf, ", (double) overhead /
                                           (60 * LOOPS));
    fflush(stdout);

    if (fork() == 0)
        for (i = 0; i < LOOPS; i += 1) {
            (void) read (yonder[0], &puck, sizeof(char));
            (void) write(hither[1], &puck, sizeof(char));
        }
    else {
        (void) times(&now);

        for (i = 0; i < LOOPS; i += 1) {
            (void) write(yonder[1], &puck, sizeof(char));
            (void) read (hither[0], &puck, sizeof(char));
        }

        (void) times(&then);

        printf("elapsed = %lf, context switch = %lf\n",
        (double) (then.tms_stime - now.tms_stime) / 60,
        (double) (then.tms_stime - now.tms_stime - overhead)
        / (LOOPS * 60));
    }

    exit(0);
}
```

B.2 gettime.c

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
/*
** Get time of day 2000 times
** Access local devices excessively
**
** William Osser
*/
main()
{
    int loops;
    struct timeval time1;

    for (loops =0;loops < 2000; loops++)
    {
        gettimeofday(&time1);
    }
    exit(0);
}
```