The Design of the Schizophrenic Workstation System

Mark Swanson
Terence Critchlow
Leigh Stoller
Robert Kessler

Center for Software Science
University of Utah
Salt Lake City, UT 84112
E-mail: {swanson,stoller,critchlo,kessler}@cs.utah.edu

Abstract

A cycle-harvesting distributed operating system is described. Building upon current kernelized operating system technology, it promises to utilize spare workstation resources for large, long-running applications, while subjecting the workstation owner/user to little inconvenience. The design is based on a novel use of the multiple-personality potential of microkernel-based operating systems. Acceptance of the system by workstation owners is encouraged by utilizing new and existing resource management mechanisms within the kernel to minimize the impact on those owners' local systems. The applications of interest may display medium-grained concurrent/parallel behavior, or they may utilize process-level parallelism, or they may simply be long-running, compute intensive, non-completion-time-critical single-threaded programs.

1 Introduction and Related Work

Networks of increasingly powerful workstations are common today in many commercial and research environments. Much of the power of these workstations is wasted. Their owners spend significant portions of their time reading mail, writing, debugging and thinking; during these times, the machines, though in use, are underutilized. Much of the time the machines are essentially idle: their owners attend meetings, engage in travel, go home in the evening. Recent monitoring of workstations in the Department of Computer Science[5] clearly demonstrates underutilization. Hoogenboom measured the mean load average of 19 faculty and staff machines to be approximately .11 (five minute load averages sampled at fifteen minute intervals for two weeks). Assuming that a machine with a load of 1.0 is fully-, but not over-utilized, 89% of the power of these machines is unused. Recently those 5 to 10 MIP machines used in the study were replaced by machines rated at 76 MIPS / 23.7 MFLOPS. Although utilization has most likely fallen, even using the conservative 89% figure, some 1285 MIPS / 400 MFLOPS are available.

In contrast to this generally low average utilization, at other times the workstation owner has computational demands that consume the entire power of his workstation and quite possibly exceed its capacity. This computationally intense usage may even interfere with the workstation's use for previously mentioned, less demanding tasks. Load sharing presents a way for the workstation owner to mitigate his own impact on his workstation's interactive performance by shedding load to other, less used workstations.

1.1 Load Sharing

Load sharing is an area of research that has sought ways to use networks of workstations more efficiently; to shift load from overburdened workstations to less utilized ones. Load sharing systems employ local and/or global scheduling algorithms [6, 27, 23] to statically determine which workstation is the best candidate for a new task, or when a newly arriving task should be transferred elsewhere. For example, the **Load Information Manager** in Utopia periodically gathers statistics about the participating workstations, using CPU average, memory availability and the number of users logged in to determine which workstation is the best candidate for a new task. Similar to load sharing systems are load balancing systems [19, 20, 4], which consider the system as a whole, attempting to remove imbalances in the system either by transferring new tasks as they arrive, or by migrating already running tasks. The goals of load sharing are similar to those of a cycle-harvesting system; we make the distinction that a cycle-harvesting system limits its load sharing to the use of otherwise unutilized resources. Some previous load sharing systems fit this definition as well. We note that load balancing systems do not fit the definition of cycle-harvesting, as their attempts to balance load can result directly in over utilization (albeit balanced) of all participating workstations.

1.2 Related Work

Many approaches to shared use of workstations have been explored, including runtime libraries, operating systems, programming languages and remote execution environments.

Distributed operating systems is an area of research that seeks to combine the power of individual workstations into larger, coordinated systems, achieving load sharing as a side effect. The various approaches that have been tried afford some level of transparency to the users. Location and reference transparency allow the user to program without knowledge of where an object is located, or whether the object is local or remote. The *port* abstraction of Chorus [18] and Mach [17] is an example of how transparency can be provided. An application simply sends a *message* to a port; the operating system is responsible for finding the receiver, and delivering it. Another example is the *RPC* mechanism of Clouds [25], which treats all services as *objects* to which invocations can either be local or remote. The goal of this type of distributed system is to create a large time-sharing system for running large applications (*possibly process level* parallel), or one on which medium-sized parallel programs can be run.

A different class of distributed system, *network operating systems*, seeks to combine networks of workstations by providing transparent access to remote resources, such as files and I/O devices, while retaining the separate operating system identities of the workstations. Locus [24], Utopia [27] and Condor [8] are examples of network operating systems that use remote access mechanisms to allow access to possibly non-local resources. In Locus, for example, remote files are accessed by converting system calls into kernel-to-kernel RPC. In Condor, system calls are handled by a *shadow* program that converts system calls into RPC. This technique requires that Condor programs be linked against a special runtime library to handle the conversion. In Utopia, a software layer implemented entirely at the user level is interposed between the application and the operating system. RPC mechanisms are used to provide global scheduling information, and remote execution.

Library-based systems such as PVM[21] and remote execution environments such as Butler[12] are another approach to load sharing. They provide the potential for wide applicability, limited by the portability of the library or environment rather than an entire operating system. For similar reasons, they are relatively simpler to implement. Their weaknesses arise from the same base as their strengths. By limiting themselves to the abstractions presented by their host operating systems, they limit their functionality and sacrifice efficiency. They often cannot properly support all direct system calls, requiring that only calls mediated by the library be used. By definition, such systems require relinking applications with the library; existing binaries simply cannot be used.

An important characteristic of a given distributed system is whether the individual workstations retain their identity. Instead of being part of a processor pool, as in Amoeba [11], workstations remain autonomous when participating in a strictly load sharing system. Systems such as Sprite [3], Condor [8] and Butler [12] are examples of this type.

1.3 Issues in Load Sharing

In all of these approaches common difficulties arise; some of them are technical in nature, others are sociological in nature, and some fall into a gray area in between.

The technical problems relate to how well the goal of exploiting the underutilized workstations is accomplished. The questions here relate primarily to efficiency. Efficiency depends on how much of the total capacity of the networked machines can be utilized and on how much overhead achieving that level of utilization imposes. The generality, or transparency, of an approach also determines efficiency, as it determines which applications can utilize other workstations. Other technical issues include platform heterogeneity, the ability to utilize workstations of differing architectures as a single load sharing system, and portability, the applicability of the solution on homogeneous systems of various architectures or different operating systems.

Sociological issues relate to the perception that a workstation is, in some sense, owned by the person whose desk it sits on, or who is its primary user. One of the many reasons why workstations are preferred over time-shared systems is that the workstation owner does not have to share the resources of his machine. This allows him to develop an internal model for the response time of his various work tasks and thus optimize his work patterns. When others utilize the workstation, the response time model could be invalidated and result in a significant impact on the owner's work. This would likely result in the owner disallowing all external use of the machine, making it unavailable for load sharing use. Load sharing systems generally use heuristics to determine when a workstation is idle, and when a foreign process must be evicted because the owner has computing needs of his own. In Butler, eviction is accomplished by terminating the foreign process. In Sprite and Condor, the foreign process is migrated to another workstation.

Administrative and security concerns fall in the middle ground. The workstation owner's concerns go beyond responsiveness and predictability. The owner does not want the availability of his machine threatened by other users and does not want other users to gain access to his system through load sharing mechanisms who would otherwise be denied such access. Some aspects of administration and security interact with generality: dependence on the workstation's operating system for file access, for instance, might require that all workstations have identical file name spaces. Some load sharing solutions require that users wishing to take advantage of load sharing must have accounts on all machines that they might potentially utilize[12]. This can present system administration costs unless all workstations on the network are already maintained with a homogeneous group of users, which can be impractical.

1.4 Addressing the Issues

The Schizophrenic Workstation system (or just Schizo) is a distributed operating system that provides a single distributed memory multiprocessor suitable for both timeshared processing and distributed memory parallel computation. It differs from distributed systems composed of dedicated (non-autonomous) workstations in that the constituent workstations do retain an autonomous identity *in addition to* participating in the distributed system. As the design of the system is described, we will show how it addresses each of the issues outlined above for load sharing systems.

The remainder of this paper is organized as follows: Section 2 describes the operating system technology that Schizo is based upon. Section 3 presents the Schizo design in terms of this technology. Section 4 discusses how the Schizo model addresses the issues raised in this introduction. Section 5 addresses shortcomings in the model. Finally, Section 6 presents some measurements and Section 7, conclusions and some areas for further research.

2 Base Operating System Technology

The current Schizo system is based on OSF/1 Release 1.0 [15], an existing example of an *operating system server* running as a task on top of a *kernelized* OS, in this case the NORMA[1] version of the Mach 3.0 microkernel. OSF/1 is also referred to as an *operating system personality*, insofar as it implements the high level operating system facilities we commonly think of as defining an operating system, e.g., the application interface, file system and process management. Since the server, or personality, runs as a task outside the kernel, it is possible and sometimes desirable to run multiple servers (i.e. multiple personalities) on top of a single microkernel.

Given its role as a *base* for separate operating system servers, the Mach 3.0 microkernel[13] provides only a small set of abstractions:

- the task
- the thread
- virtual memory
- ports
- messages
- devices

The NORMA version of the microkernel is so called because it contains direct support for a single system image on NO Remote Memory Access multicomputers. The direct support entails changes to the microkernel such that a collection of microkernels, each running on a separate processor, present a unified port name space, a shared virtual memory system providing distributed shared memory[7], and a mechanism to specify the processor on which a new task is created. The NORMA microkernel contains additional support for determining system configuration at system startup.

Mach identifies and manipulates most resources through the use of ports. The provision of a unified port name space means that operations on ports display both referential and location transparency. Since tasks and threads are referred to in system calls by their associated port names, this transparency also applies to them.

Location and reference transparency for virtual memory relies on the unified port name space since memory *objects* are referred to via ports. It also requires the extension of the underlying paging mechanisms, since the *contents* of memory objects are referenced by virtual address. A reference to an address whose physical page is not resident on the local node is resolved via the in-kernel memory manager, resulting in a network transfer of the appropriate page.

The microkernel implements a priority scheduler for threads and provides system calls to manipulate thread priorities. The microkernel also implements ledger objects[10] to control the use of specific resources by a task or task group. Among the resources subject to ledger control are:

- number of tasks and threads;
- virtual memory usage;
- number of ports

As an operating system personality, OSF/1 presents its own set of abstractions, which are constructed on a base of Mach abstractions. For example, an OSF/1 process is comprised of a Mach task with an associated Mach thread and virtual memory objects representing its text and data. Since an OSF/1 process is contained within a Mach task, it is often the case that referring to a "task" is equivalent to referring to a "process." For the remainder of this paper, except where noted, the terms process and task are used interchangeably.

3 The Schizo Design

The Schizo system design employs features of the distributed microkernel-server architecture to marshal underutilized but autonomous workstations into a distributed system. Schizo is, in particular, an application of the multiple personality feature. Each workstation runs the Mach 3.0 NORMA microkernel. Each microkernel supports two personalities[1]: one is the local workstation "system"; the other is the Schizo distributed system. Only one server *task* runs on each workstation, but the server on the workstation acting as the Schizo host machine, also communicates with/utilizes

[1] This is really the simplest case. It is quite possible that the workstation owner might be running multiple "local" operating system personalities (e.g. OSF/1 and a DOS server[16]).

the kernels of all the other workstations and thereby uses those workstations to run its processes. This use of multiple personalities, one being the distributed system, is the key concept of the Schizo design and the reason why we named this project the "Schizophrenic Workstation" – the workstations will be schizophrenic in their use as either a dedicated workstation or as a node in the distributed system.

Although the NORMA microkernel provides a single unified port name space and an integrated virtual memory system, the ports and virtual memory objects of the two personalities using a workstation are logically partitionable, i.e., the tasks, threads, ports, and virtual memory objects of the local workstation system are unknown to the Schizo system. The same holds true in the other direction. The two systems are independent of each other and only interact indirectly through their use of the same microkernel and the underlying machine.

3.1 Local Scheduling

For a distributed system such as Schizo, a useful distinction between local and global scheduling can be made. The Schizo system depends on the normal Mach microkernel priority scheduler to perform local scheduling. Schizo depresses the priorities of its threads relative to the "normal" priority levels at which local tasks are expected to run. Systems that use depressed cpu priority alone have been known to display undesirable paging behavior, as the paging system is unaware of these differences in priority. We have therefore modified the kernel to provide two classes of paging service; this is treated in more depth in section 4.2.

3.2 Device Access

Devices are another area where interaction between servers might occur. While cooperative shared control of a device by multiple tasks is possible, we do not envision the Schizo system attempting to share devices with the resident server. There are a number of reasons for this:

- security – what cannot be accessed cannot be compromised (at least directly);
- reliability – multiple device controllers would require a high level of synchronization and co-operation, which is in contradiction to the premise that the two systems are ignorant of each other;
- flexibility – we envision allowing the workstation server to be whatever the "owner" chooses to run; requiring these servers to adhere to some device sharing policy/mechanism would limit the choices available to the user

The Schizo system does not require direct device access, nor are Schizo child tasks expected to do so. This is enforced by the fact that only the operating system server can provide the necessary capability, and the fact that Schizo child tasks cannot communicate with the local server to request it. This separation avoids interference with the local system, and it ensures that no dependencies (at the operating system level) on the availability of a particular workstation are introduced. The Schizo design tolerates the dynamic dependence of a child task on the workstation it is currently utilizing but does not tolerate *system* dependence on particular workstations (an exception to this is described in the next section).

The tasks *indirectly* utilize the network device(s) when they manipulate location transparent entities such as ports and virtual memory. Such accesses can result in the microkernel performing network transactions on their behalf. These network transactions, undertaken by the microkernel on behalf of the Schizo tasks, present another situation which can impact performance of local activities. We have not yet taken measurements which might indicate whether or not they present a noticeable load. If they do, we will prioritize access to the network devices so that Schizo-related messages will be delayed in favor of local workstation activity. In the event that incoming Schizo-related messages present an unacceptable load, heuristics will need to be implemented to quench them at their source.

Tasks also may indirectly utilize local disks for virtual memory paging activity. This occurs if the pager for the Schizo memory objects pages to local disk. Such activity is essentially unavoidable, but changes to the VM system (see Section 4.2) attempt to minimize performance impacts[2].

3.3 Static and Dynamic Resource Base

An exception to the avoidance of dependencies on a particular workstation is the Schizo "core." To simplify the design and to provide an assured resource base, a Schizo system must have at least one dedicated processor, on which the Schizo server task runs, accepting job requests. This single dedicated processor can be generalized to a collection of processors, in which case the dedicated core of the Schizo system is essentially an OSF/1 multicomputer or cluster[3]. On these core machines, Schizo is the only personality/server and *does* have access to (and dependence upon) devices.

The workstations used by the Schizo system (outside of its core machines) are considered to be "owned" by their primary users and not under the control of the Schizo system. It is expected that they may come and go (shutdown or crash and reboot), often without warning to the Schizo system. The Schizo system is expected to survive the disappearance of constituent workstations outside the core and to re-assimilate such systems when they reboot. Schizo tasks, however, may disappear if their host workstation disappears. In the case of controlled shutdown, the Schizo tasks will be migrated away from the workstation to some other host. The core system forms a "haven of last resort" in the event that the pool of underutilized workstations shrinks precipitously relative to Schizo's workload. An important impact of this is that when migration occurs the task must retain no residual dependencies on the workstation it is removed from. Secondly, the core system must provide sufficient resources (especially swap/paging space) to contain all of the extant tasks, ensuring their survival, although there may not be sufficient resources to ensure their progress.

3.4 Global Scheduling

As a distributed system, Schizo must make decisions about task placement. In the current design, this activity is handled centrally, and rather naively, within the Schizo core. The central server regularly polls the participating workstations to gather load information. When a new Unix process is to be forked, or a process (task) needs to be migrated, a location is chosen for it based on these criteria:

- a workstation already running a Schizo task will not be selected;
- a workstation with a load above a specified threshold will not be selected;
- from the remaining workstations, the one with the lowest load, below a maximum threshold, will be selected;
- if no qualifying workstation is found, the task is created on the core system.

Workstations that are currently running Schizo tasks are ignored in favor of those that are not yet doing so based on the assumption that Schizo tasks are compute intensive and will make maximum progress if they need not compete with other Schizo tasks. In the event that Schizo determines that it has insufficient resources to ensure the new task's survival in the event of a decrease of participating workstations, it will refuse to accept the task.

The major attractiveness of this approach lies in its relative simplicity. The global scheduler is currently limited in function to identifying a candidate workstation for creation of a new process or migration of an existing one. It also detects situations in which a process must be migrated because of an increase in the load on its host node which prevents it from making progress. As a centralized mechanism, the scalability of this approach may be questionable, although some previous research[23] indicates that it will be adequate for the numbers of processors we envision, in the tens or (rarely) low hundreds of machines.

[2] Replacing the virtual memory object's default pager with one that paged to another device, e.g., the network, could avoid all use of the local disk. This ability is not included in the current design.

[3] In fact, to utilize file systems on a multiple-node core system, the OSF/1 AD variant must be used, which provides a distributed file system service.

3.5 Design Summary

The overriding theme of the design of Schizo is to maintain separation of the local workstation owner's system from the distributed load sharing system. This separation is accomplished by implementing these two systems as separate servers running on top of the virtual machine presented by the Mach microkernel(s). Where the microkernel does not enforce adequate separation, Schizo adheres to appropriate *conventions*, such as eschewing access to the workstations devices, to enhance the separation. Existing and new mechanisms within the microkernel are used to bias the microkernel's allocation of resources in favor of the workstation owner's system.

4 Evaluation of the Schizo Design

How does our Schizo design address the problems of load sharing detailed in the introduction? We shall examine in turn each of the areas identified there.

4.1 Efficiency

In terms of efficiency on an individual workstation basis, Schizo can achieve the same level of utilization of that workstation as a good batch or background task mechanism can achieve.[4] Whatever processor and physical memory resources are not required by local tasks, are given to the Schizo tasks resident on the workstation.

This is in contrast to the approach of some load sharing systems[8] that, in an attempt to be unobtrusive, cease activity on a given workstation for some fixed amount of time after any "local" activity. Such a system does well in exploiting idle workstations, but fails to utilize the unused cycles of an underutilized workstation. On the other hand, Schizo's prioritized use of both processor and physical memory allow it to exploit these underutilized machines without significantly impacting the responsiveness of local activities.

The other component of utilization is generality; i.e., what are the limitations imposed on processes that Schizo distributes to the participating workstations? The only strict limitation is that the process may not perform direct device access. All operating system services are available to any Schizo process regardless of the process' actual location and will function as if the process were resident on the same processor as the Schizo server task. This is not to say that a process that performs large numbers of system calls, especially file system operations, will enjoy the same level of performance as if it were co-located with the operating system task. Additional latency will be introduced by the required network transactions. Conversely, the operating system task may be able to process system calls more quickly in a Schizo system if the user computations have been successfully distributed to other workstations, leaving more resources for the operating system task on its processor.

4.2 Sociology

There is potential for sociological impact on the workstation owner both when Schizo tasks are resident and when they are not. In the latter case, it is important that the observable overhead imposed by the Schizo system is small. The system calls of local tasks are still serviced by a locally resident operating system server task. We anticipate that support of location transparency within the microkernel will result in only slightly increased service times for system calls. The other cost is an occasional microkernel response to load gathering queries from the Schizo system. Schizo requires no active participation beyond these responses, from the workstation, in terms of global scheduling.

When a Schizo task is present, the sociological impact is potentially even greater. Therefore, a major goal of the design requires that the presence of Schizo processes should be unobtrusive to the workstation owner. In a standard Mach 3.0 based system, multiple operating system servers and their child tasks are generally treated equivalently by the kernel. However, Schizo's child tasks

[4] In fact, the design of an effective mechanism for unobtrusive background tasks is a by-product of this research.

should be treated differently than those of the workstation owner's system, in that the latter should receive preference in any competition for resources. This is accomplished via prioritization of CPU usage and the virtual memory system.

Operating system servers are free to manipulate the priorities of the threads in their child tasks. As mentioned earlier, the thread priorities for all Schizo tasks running on "owned" workstations are depressed to ensure that the workstation's tasks receive preference for cpu resources. While this ensures that Schizo tasks run only when there are no local tasks that wish to run, this does not address the potentially adverse effects of the paging behavior that results when a large Schizo task is allowed to run.

The effects of paging behavior have been addressed by extending the microkernel to recognize two classes of memory objects. Whenever the Schizo personality creates processes on a two-personality workstation, it specifies to the microkernel that the task allocated should have an attribute specifying that all memory objects it creates be of low priority. Processes of the local workstation system generally do not have this attribute. When a page must be replaced due to a memory shortage, processes with the low priority attribute are considered first when determining which page to evict. This approach to prioritized physical memory usage was pioneered (on an earlier version of Mach) in the Stealth distributed scheduler[6].

4.3 Security

Beyond the promise of not significantly impacting workstation performance is prevention of denial of service. Load sharing use of a workstation should not increase the vulnerability of the workstation. In particular for Schizo, its tasks running on a workstation should not be able to compromise the microkernel (an indirect threat) or the other tasks running on that workstation (a direct threat). The independence of the servers, as described above, addresses direct negative interaction between Schizo tasks and those of the local system, including the local operating system server itself. Indirect denial of service might result from exhaustion of microkernel resources, e.g., by creation of a task large enough to consume all of the available paging space.

The microkernel's ledger system[5] will be used to prevent this kind of resource exhaustion by limiting the total quantity of resources (e.g. ports, memory objects, paging space, etc.) that Schizo and its tasks can consume on each workstation. This use of ledgers will serve an additional function in ensuring the survival of Schizo tasks in the event that its resource base erodes seriously. As previously mentioned (Section 3.3), the core system reserves sufficient resources to reabsorb all tasks placed on other workstations. The limits provided via ledgers will serve not only to restrain use of workstation resources but also to ensure that the system will be able to find the resources necessary to migrate the task back to the core system if that becomes necessary.

5 Making Schizo from OSF/1

As it comes "out of the box," the combination of a Mach 3.0 NORMA microkernel and an OSF/1 server could be used to build a distributed, multiple personality system much as we have described. What distinguishes Schizo from such a rudimentary system can be summarized as follows:

- unobtrusiveness;
- security;
- robustness;
- global scheduling;
- task migration

[5]The current version of the NORMA microkernel does not implement the ledger system. However, it is part of the OSF Draft Proposed Specification[14], and is expected to be implemented in the future.

The design of Schizo has addressed several of these issues, including scheduling, unobtrusiveness, and security. Robustness is being addressed empirically, as an implementation issue. The NORMA extensions were implemented for a multicomputer environment, in which loss of the system as result of loss of a node was deemed acceptable. With each crash of a participating node, we learn more about inter-node dependencies and how to deal with them.

5.1 Prototype Implementation

Our initial implementation of Schizo relies on a server task running on the core system. This server registers itself with the name-server and waits to accept requests to start processes on available workstations. The requests are presented via a frontend program that is responsible for contacting the Schizo server. In a future version of the system the Schizo server's functionality could be bundled more tightly with the OS server. The current organization eases the development process, both in terms of system building and in debugging.

A Schizo program is started by contacting the Schizo server, and handing it a *program invocation* (i.e. the program name and arguments). During the creation of the new process, Schizo selects a node on which to create the underlying Mach task. Schizo then replaces the exception port of the new task with one that it holds the receive right for. This enables it to filter and react to the exceptions the task may encounter. The new task also receives an additional send right to a *fork request* port, the receive right of which is also held by the Schizo server. This enables Schizo to intercept fork system calls so that it can find a suitable node for the new task, making it relatively easy to write *process level* parallel programs. The OSF/1 emulator has been modified to use this new send right instead of the normal one for all fork operations.

In order to facilitate the migration of Schizo tasks away from a machine that is being shutdown, the microkernel has been extended to issue a new exception to all remaining tasks when it receives a shutdown message from a server (usually the workstation OS). The Schizo server receives this exception and migrates those tasks to some other node before replying to the exception, thereby allowing the microkernel/node in question to actually shutdown. The task migration mechanism used is one developed at the University of Kaiserslautern[9], which provides both eager and lazy memory object copying. When the migration is the result of a node shutdown, lazy copying is usually fatal, as the node goes down before the memory object can be faulted to the new node.

Orderly shutdowns are one situation in which task migration must occur, but another arises when the node on which a task has been placed becomes utilized by workstation tasks to the point that the Schizo task cannot make progress. In such an event, the task is moved to a new underutilized node, if one is available, or to the Schizo core if a suitable node cannot be found. Such migrations are done lazily, with pages sent across the network as they are touched. The advantage of this approach is that the overhead of migrating the memory objects is spread out over time, rather than in a large burst that is noticeable to the workstation owner.

The Schizo prototype also includes an implementation of the Stealth Prioritized Virtual Memory System that partitions memory objects into 2 classes; low priority and normal priority. The standard microkernel paging algorithm uses a set of three queues holding active, inactive and free pages[2]. The implementation of the two classes of memory objects entails providing each class with its own set of paging queues. At creation time, an object is assigned to a class based on the priority of the task requesting its creation. One set of queues receives preferential treatment and is considered to possess higher priority in use of free pages. In addition, pagein activity for lower-priority objects is not initiated if pagein for a high priority memory object is in process[6]. A more detailed description and performance results can be found in [22].

[6] It is not clear at this time whether this additional mechanism is necessary, since pagein activity runs in the faulting thread's context and is thus run at a depressed priority. Experiments are being conducted now to determine its effectiveness empirically.

Architecture	Fork Time	Exec Time
HP730	.02	.033
I486	.27	.29

Figure 1: Times for (remote) fork and exec system calls. Times are in seconds.

Architecture	Fault Time
HP700	6480
I486	15240

Figure 2: Time to fault a page across the network. Pagesize is 4096 bytes. Times are in microseconds.

6 Measurements

A series of experiments have been performed and their results will be described in this section. By means of these experiments, we hope to:

- characterize the kind of applications that are appropriate for the Schizo system;
- demonstrate the functional capabilities of the system;
- quantify the effects of remote execution and migration on the applications;
- quantify the effects of the guest applications on the workstation tasks.

After describing the experimental testbed, each of the items above will be treated.

6.1 Experimental Testbed Configuration

Currently, the Schizo system implementation is a prototype and has not been deployed on personal workstations. Experiments are performed using two clusters of dedicated machines. One cluster consists of 4 HP 730 workstations with 32 MB, ethernet interconnect, and local disks on SCSI disk controllers. The other cluster consists of 4 PCs with I486 DX/50 processors, 16 MB, ethernet interconnect, and local disks on SCSI controllers. All of the machines are running NMK13.16 and OSF 1.0.4 single server. To minimize external network traffic effects, the two clusters of machines coexist on a single ethernet subnet with only one other host which serves as a gateway to the outside world.

6.2 Characterizing Appropriate Applications

We can quantify two characteristics of appropriate applications: minimum practical size, where size is a function of compute time and program test size, and by kind and frequency of systems calls it employs.

Figure 1 displays the times for `fork` and `exec` system calls done remotely (i.e., the process calling `fork` is on the same node as the OSF/1 server, but the child is to be created on a different node).[7] The paging activity involved is minimal, as the child's pages are copied on demand (lazily) and it immediately exec's a one-page program. The exec'ed program simply exits. Figure 2 shows the time to fault a page of memory across the network. The test program ensures that the pages in question are already memory resident by touching them. It then forks, with the child assigned to

[7] `Fork` and `exec` may be expected to improve in the OSF/1 AD server, which has some provisions for distributed process management.

System Type	HP700		I486	
System Call	Local	Remote	Local	Remote
nop	100	1240	348	3170
getpid	13	13	20	20
read	812	83170	1120	116830
write	496	49900	659	50020
stat	269	1450	769	3710
sbrk	22	22	36	39
chown	273	1380	752	breaks
create	555	133430	1376	166950
signal	375	245680	912	300330

Figure 3: Remote and local system call timings on HP700 and I486 clusters. Times are in microseconds.

Test case	Local	Remote
null.c	1	8
cpio.c	4	32

Figure 4: Remote and local C compile times for an empty file an a 1400-line program on the I486. The times are in seconds.

a different node. After the fork, the pages are touched again by the child. Taken together, we can postulate that "running" a program on another node will (on the I486) cost at least .56 seconds plus approximately 15 milliseconds per page of text. This constitutes a (by no means maximal) lower bound on the compute time that a Schizo application would need to consume to be considered "practical."

Figure 3 lists a selection of system calls and the elapsed time for each as a local or remote call. The `getpid` and `stat` calls are handled by the emulator in the calling process' context; thus their times are essentially the same in both cases. `Nop` is an "empty" call of no arguments with no associated actions; it is used simply to measure communication and context switch time involved in a system call. `Stat` takes roughly 5 times longer in the remote case and `read` and `write` are each two orders of magnitude slower in the remote case. Coupling this with the rather large times for remote `fork` and `exec`, an application such as **make** is currently an inappropriate application for Schizo. See Figure 4 for a comparison of local and remote compilation times

One can more generally state that any program performing large amounts of reading and writing (especially small reads and writes) and making frequent use of signals (e.g., emacs) is an inappropriate application.

Having ruled out certain classes of applications, we turned to those that fit the description in our introduction: long-running, compute intensive, single-threaded programs.[8] An example of such an application is a ray tracer implemented in Scheme[26]. Once it completes an initialization phase, it computes intensively; and the length of the computation is easily adjustable by specifying the desired number of pixels. Test scenarios could be developed with short (several minute) computations and then much longer runs of the same program and test scenario were easily achieved. Finally, the program is a "real" application, rather than a synthetic benchmark. Figure 5 lists the local and remote elapsed times for a small and moderate sized sample runs of the ray tracer. We have not yet discovered an explanation for the (apparently anomalous) speedup achieved by executing the program remotely. Though relatively large, the program's virtual memory size did not exceed the

[8] We had the additional qualifier "non-completion-time-critical"; this is more descriptive of the user's expectations than of the program.

Test size in pixels	Local	Remote
1156	12.9	13.3
13000	96.8	88.5

Figure 5: Remote and local ray trace times on the I486. Times are in minutes.

available real memory and little paging occurred after the relatively short initialization phase. We have used this program in later tests described in Sections 6.4 and 6.5.

6.3 Demonstrating Functionality

The desired functional behavior for Schizo encompasses the following:

- the ability to run programs on participating nodes, with automatic node selection by Schizo, and without requiring any modification, recompilation, or relinking of the application;
- the containment of any child processes within Schizo's purview;
- the migration of processes as necessitated by increases in load on host machines;
- the migration of a process as necessitated by the orderly shutdown of the host machine on which it resides.

As just one example, Utah Common Lisp (UCL) was started via Schizo. It ran a simple loop[9], which allocated and discarded objects on the heap, coincidentally causing periodic garbage collections. With the help of some artificial load generated on the host workstations, the UCL program moved from node to node making and collecting garbage. The ability to perform most of the functions listed above (except for handling shutdown), is implicit in the experiments outlined in the next two sections.

6.4 Performance Effects on the Guest Application

To prove useful, the Schizo system must ensure that programs entrusted to it make reasonable progress when resources are available within the system. The location at which these resources are available is expected to shift as workstation owners come and go and as they shift from one task to another. We wish to make a stronger statement than a simple assurance of progress; the user should be able to form a reasonable estimate of completion time for his program. This estimate would include assumptions about the availability of resources and expected "normal" time to completion for the task. We assume that the user can observe the load on the workstation base before forming his expectations, or that he has a model, perhaps from prior experience, of the load to expect.

There are other effects on runtime that are artifacts of running on a distributed system. We addressed some of these earlier, in particular system call and paging times. When migration is taken into account, it may effect runtime both directly, in terms of the immediate cost of migration and the compute time lost while migrating, and in increased paging costs due to load on the machine the task is migrated *from*. We assume that the system has selected a lightly loaded destination for the task.

Figure 6 shows the results of an experiment intended to quantify this latter cost. An artificial load program was constructed that repeatedly loops, imposing a high compute load for a given number of seconds (the "load" column in the figure) and then becoming quiescent (sleeping) for some other specified number of seconds.

[9] Of course, in Common Lisp, a simple loop can touch hundreds of text and data pages.

Load Period	Migration Interval	Elapsed time in minutes	Migrations
30	180	20.36	6
30	240	18.46	6
30	300	16.66	4
0	150	16.88	6
0	210	15.81	4

Figure 6: Effects of source host load on a migrated task.

The experiment is run with three nodes, one serving as the Schizo core and two as participating "workstations". The load program is started on the two workstations, 180 degrees out of phase (i.e. viewing the loaded/idle states as a clock). The test program (the ray tracer described earlier) is started via Schizo, which places it on whichever workstation is currently idle. When the load program on that machine enters its load imposition phase, the ray tracer is migrated to the other machine. The time to completion and number of migrations for varying migration intervals is reported. As we can see, the ray tracer, which took approximately 13 minutes as a normal, local job, is significantly degraded when the node it migrates from is very busy. We attribute this to time spent fetching pages from the busy machine. As the migration interval grows, more time is available to work with the pages once they are faulted over. This conclusion is borne out by the last two cases, where no load was imposed on the source machine; the degradation is much less.[10]

We have not yet formed a model for the expected degradation, but these experiments will serve as a starting point for developing one.

6.5 Performance Effects on the Host Workstation

The performance effects of Schizo tasks on the host workstations and their tasks has already been identified as an important consideration. We attempt to mitigate these effects in a couple of ways:

1. by depressing cpu priority and constraining competition for physical memory;
2. by migrating a task to another host when the current host becomes too busy.

We shall first examine some results that demonstrate how effective the first strategy is in minimizing the impact of guest tasks. We shall then explore what, if any, lingering effects a migrated job has on the source node.

6.5.1 Limiting CPU and VM Consumption

An experiment was conducted in which a script of "typical" user's activities, including editing and sleeping, was run as a foreground task. As an example of a Schizo guest task, a Cholesky factorization was run as a background task to determine its effects on the time required to complete the script of user activities.

Figure 7 displays the results. A baseline time was established corresponding to a private workstation with no Schizo tasks. Then a "worst case" time was obtained by running the background task with no controls, i.e., as another foreground task. An appreciable degradation occurs in this case. Next the background task is run with its cpu priority depressed. The degradation experienced by the foreground task is lessened, but it is still significant. Then the background task was run with normal cpu priority and depressed VM priority. This approach led to even better results, from the point of view of the foreground task, than depressing cpu priority. Finally, depressing both cpu and VM priority led to a degradation that was barely perceptible.

[10] Since no load was imposed on the source machine, the migrations were artificially induced by signals to the Schizo server.

Background Activity	Foreground task Elapsed time in seconds	% Increase in Time
None	133	–
Cholesky, no control	151	13.5
Cholesky, depressed cpu	145	9.0
Cholesky, depressed VM	141	6.0
Cholesky, depressed cpu and VM	136	2.2

Figure 7: Effects of background task on foreground activity.

	Counter value after 30 seconds elapsed time	% of baseline
Baseline (no Schizo tasks)	245624169	–
Schizo task faulting pages	162747867	66.2

Figure 8: Amount of "work" performed in 30 seconds.

6.5.2 Residual Effects

We saw in Section 6.4 that the lazy faulting of pages that occurs after a migration has a noticeable effect on the progress of the migrated job. We modified the experiment slightly by adding a counter in the busy loop of the artificial load program, as a measure of the "work" it performed in the given number of seconds. Figure 8 shows the amount of work performed by the load program in an idle Schizo system (no migrating tasks) and the amount performed immediately after a migration occurs. The impact of the migration on the workstation is significant. It is also intuitively reasonable. Page faults, including those caused by the migrated process, are serviced by kernel threads, which run at a higher priority than user threads, There may also be a second order effect if the pages for the migrated task have been paged out to disk and need to be paged in to satisfy the faults. We have not attempted to isolate these effects, if present. These results lend support to our intuition that some additional work on depressing Schizo activity will be required to meet our goal of non-intrusiveness.

7 Conclusions and Further Research

The prototype that we have constructed has demonstrated that Schizo can distribute load to participating workstations without seriously impairing their performance as autonomous systems. It can automatically shift load as a consequence of changes in resource availability, and it can "rescue" processes from workstations undergoing planned shutdowns.

Other goals of the design have not yet been realized in the prototype, particularly resource control by means of ledgers. Deployment on users' desks awaits further work on robustness and on performance of the NORMA microkernel base.

We have identified several areas in which further research is warranted. One is the addition of a checkpointing mechanism to protect Schizo processes against network and node failure. Currently, Schizo processes that are running on a node that crashes, are aborted and all local state is lost. For long running application programs, this approach is unreasonable. Users of Schizo should be confident that a long running job submitted to the system will survive node failure, continuing to run as long as the core system remains alive.

The ability to handle heterogeneous nodes is another fruitful area of research. A first approach would be to allow the memory of nodes to be shared between different architectures. Since a page is ultimately just a collection of bits, paging to a node of a different type should be feasible and might allow applications that were paging to disk to page to other nodes. The next step of allowing forking and migration of tasks across different architectures will require significant operating system and also compiler work (for example, at well defined points, it is possible that a task could snapshot its state and transform that data to another architecture and restart).

Another area we wish to explore is extending the Schizo model of one distributed system and many autonomous workstations to one where all the workstations host a set of "overlapping" distributed systems. Rather than a single core system where jobs are submitted, jobs may be submitted on any of the participating nodes. The advantage of this approach is that there is no longer a dependency on a core system; any user on any participating node can submit a job to Schizo. The disadvantage is that the scheduling and migration facilities are now decentralized, and thus are potentially more complex in construction.

References

[1] J. S. Barrera. A fast Mach network IPC implementation. In *Proceedings of the Second Usenix Mach Symposium*, pages 1–12, 1991.

[2] D. Black, J. Carter, R. MacDonald, J. Van Sciver, P. Wang, S. Mangalat, and E. Sheinbrood. OSF/1 virtual memory improvements. In *Proceedings of the Second Usenix Mach Symposium*, pages 87–104, 1991.

[3] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software-Practice and Experience*, 21(8):757–785, August 1991.

[4] C. Gao, J.W.S. Liu, and M. Railey. Load balancing algorithms in homogeneous distributed systems. In *Proceedings of the International Conference on Parallel Processing*, pages 302–306, 1984.

[5] P. Hoogenboom. System performance advisor: An expert system for Unix system performance management. Master's thesis, Dept. Computer Science, University of Utah, Salt Lake City, Utah, 1992.

[6] P. Krueger and Chawla R. The Stealth distributed scheduler. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 336–343, May 1991.

[7] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, Sep 1986.

[8] M. Litzkow, M. Livny, and M. Mutka. Experience with the Condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Oct 1990.

[9] D. Milojicic, W. Zint, A. Dangel, and P. Giese. Task migration on top of the Mach microkernel. In *Proceedings of the Third Usenix Mach Symposium*, 1993. Submitted.

[10] D. W. Mitchell. Mach resource control in OSF/1. In *Proceedings of the Second Usenix Mach Symposium*, pages 123–130, 1991.

[11] S.J. Mullender, G. Rossum, A.S. Tanenbaum, R. Renesse, and H. van Staveren. Amoeba - a distributed operating system for the 1990s. *IEEE Computer Magazine*, pages 44–52, May 1990.

[12] D. A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12, 1987.

[13] Open Systems Foundation and Carnegie Mellon University., Cambridge MA. *MACH 3 Kernel Interface*, Jan 1992.

[14] Open Systems Foundation and Carnegie Mellon University., Cambridge MA. *MACH 3 Kernel Interface: Draft Proposed Specification*, July 1992.

[15] *Guide to OSF/1: A Technical Synopsis*. O'Reilly & Associates, Inc., 1991.

[16] R. Rashid, G. Malan, D. Golub, and R. Baron. DOS as a Mach 3.0 application. In *Proceedings of the Second Usenix Mach Symposium*, pages 27–40, 1991.

[17] R. F. Rashid. Threads of a new system. (mach, a multiprocessor operating system). *UNIX Review*, 4(8):37–49, August 1986.

[18] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS distributed operating systems. Tech Report 90-25, Chorus Systemes, April 1990.

[19] M. Schaar, K. Efe, L. Delcambre, and L. N. Bhuyan. Load balancing with network cooperation. In *Proceedings of the 11th International Conference on Distributed Computing Systems*. IEEE, May 1991.

[20] N.G. Shivaratri and P. Krueger. Two adaptive location policies for global scheduling algorithms. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 502–509, 1990.

[21] V. S. Sunderam. PVM: A framework for parallel distributed computing. In *Concurrency: Practice & Experience*, pages 315–339, Dec 1990.

[22] M. Swanson and T. Critchlow. A background task facility for MACH-based systems. Technical report, University of Utah Computer Science Department, Apr 1993.

[23] M. Theimer and K. Lantz. Finding idle machines in a workstation-based distributed system. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 112–122, Jun 1988.

[24] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.

[25] C. J. Wilkenloh and U. Ramachandran. The Clouds experience: Building an object-based distributed operating system. In *USENIX, Distributed and Multiprocessor Workshop*, pages 333–347, October 1989.

[26] Benny Yih. Ray tracing in concurrent scheme. Technical Report CSS Opnote 91-02, Center for Software Sciences, Department of Computer Science, University of Utah, Salt Lake City, Utah 84112, 1991.

[27] S. Zhou, J. Wang, X. Zheng, and P. Delisle. UTOPIA: A load sharing facility for large heterogeneous distributed compter systems. Technical Report CSRI-257, Computer Systems Research Institute, University of Toronto, April 1992.