Load Distribution on Microkernels*

Dejan S. Milojičić† Peter Giese Wolfgang Zint

University of Kaiserslautern, Informatik, Geb. 36, Postfach 3049
Erwin-Schrödingerstraße, 6750 Kaiserslautern, Germany

Abstract

The paper investigates the appropriateness of a μkernel for load distribution. We demonstrate μkernel benefits by presenting a load distribution implementation on top of the Mach μkernel. Our load distribution scheme is based on three main parts: task migration, load information management and distributed scheduling. We show that it is relatively easy to implement transparent task migration on top of a message passing μkernel, such as Mach. Changing the distributed scheduler so that it considers IPC and VM load in addition to processor load is straightforward, and it yields up to a 30% improvement in average execution time. We investigate distributed scheduling strategies in the light of the extended load information management. Finally, we compare load distribution on various OS architectures.

1 Introduction

There are many new attractive issues in distributed systems research, such as multimedia and mobile computing. However, there are also some old, but still unresolved issues that never stopped occupying the interest of many researchers. One of these issues is Load Distribution (LD). LD represents the transfer of computing requests across the nodes in distributed architectures, in order to achieve various goals, such as resource sharing, fault tolerance and improving system throughput or application response time.

The field of Load Distribution has been the focus of many interesting projects. Unfortunately, it has never been adopted as a wide used tool, despite many successful implementations, such as [1, 3, 4, 5]. One of the reasons for this may be the absence of widely used distributed operating systems, and the adequate parallel applications. The situation is being changed lately. New operating systems, such as Mach [2] and Chorus [9], are inherently distributed. New distributed applications are evolving, such as PVM [10]. In this new environment, we see new chances for LD.

Load distribution consists of: mechanisms for the transfer of load, such as Task Migration (TM); mechanisms for collecting and disseminating load information; and scheduling algorithms. In our research, we are mainly interested in the mechanisms, particularly we target the operating system aspects of load distribution. We also investigate scheduling algorithms, but only to demonstrate how well our new mechanisms behave. We do not target a traditional workstation environment, but rather clusters and massively parallel processors, running new distributed applications. We present the design and implementation of our LD mechanisms and demonstrate their effectiveness with preliminary performance measurements.

The remainder of the paper is organized in the following manner. In Section 2 we elaborate on the μkernel advantages for LD implementation. Each of the modules of our LD scheme is described in Section 3. Performance measurements are presented in Section 4. We compare LD on various architectures in Section 5. In Section 6 we mention related research. Finally, in Section 7 we give our conclusions and suggest future work.

2 Microkernels and load distribution

New operating systems tend to be small μkernels with various (usually user space) servers. On top of μkernels run emulations of various operating system personalities, such as BSD UNIX [2] and AT&T UNIX [9]. In μkernels most of the functionality of monolithic kernels is moved into user space, leaving only the basic support in the kernel, as presented in figure 1. A

*This paper was presented at the IEEE Workshop "Future Trends in Distributed Computing Systems", Lisboa, Portugal, 22-24 Spetember 1993.

†Currently on a leave from Institute "Mihajlo Pupin", Belgrade, Yugoslavia.

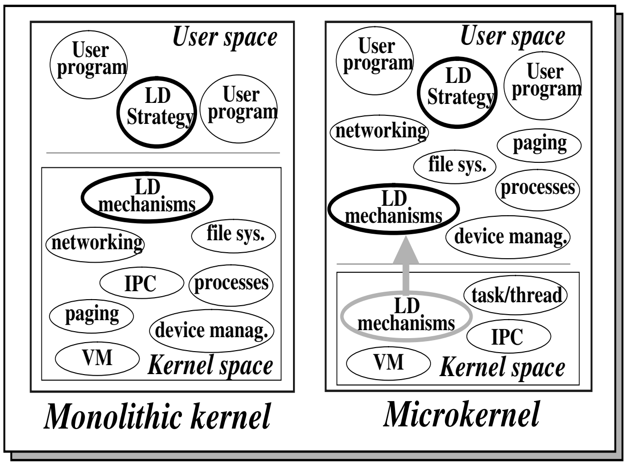

Figure 1: Monolithical v. μkernel architectures: much of the functionality is moved into user space, including the transparent LD mechanisms.

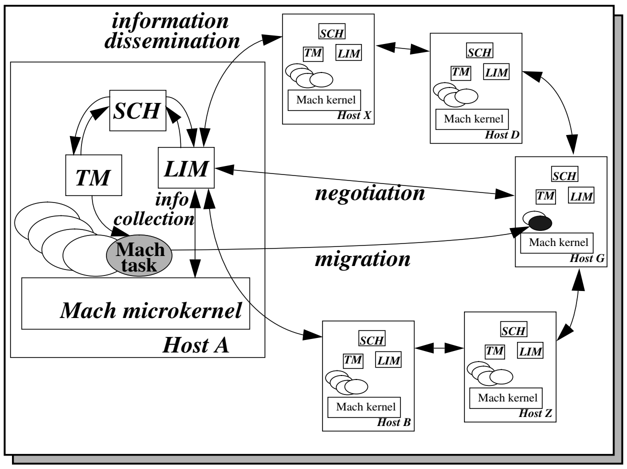

Figure 2: LD architecture: based on the information provided by the LIM module, the scheduling module instructs the TM module to initiate migrations.

description of contemporary μkernels could be found in [2]. The characteristics relevant to LD implementations are:

- Object orientedness and message passing enhance transparency. Treating the μkernel abstractions as objects, such as tasks and threads, makes the implementation significantly easier.
- UNIX semantics is moved to a higher level: LD deals with the μkernel abstractions (task/thread, ports etc.), which are more scalable and suitable to distributed environment than UNIX abstractions (processes, signals etc.).
- IPC based device access further improves TM transparency. In traditional systems, a process can leave some of its state in device drivers, rendering its transfer impossible. In Mach it is possible transparently to move the tasks that are involved in device communication, since there is a way to abort IPC activity, and restart it after the migration.
- The LD mechanisms have been traditionally supported inside monolithic kernels. For μkernels it is also possible to support these mechanisms in user space, for instance task migration [6], as presented in Figure 1.

The Mach μkernel, and in particular its NORMA version have been of significant help in our project. The NORMA support for the network transparent IPC and Distributed Shared Memory (DSM) made our efforts much easier.

3 LD scheme on top of Mach

We claim that LD should be performed on top of the μkernel, which is a common denominator for all running OS personalities. It should be implemented in user space with minimal, if any, kernel modifications. Beside the traditional information, such as processor load and paging, LD should benefit from the information on the network IPC and DSM.

What is new in our approach? Microkernels provide a unique environment which accounts for OS abstractions, such as files and various flavors of IPC, by mapping them to the basic μkernel elements: processing, memory and IPC. Accounting only on these three elements simplifies our information management, while providing more accurate insight into the system load. For example, file activity is reflected in paging, device access is based on the IPC interface, etc.

Our LD scheme has been developed in three phases. In the first phase, task migration has been provided as a main mechanism for LD. In the second phase we built Load Information Management (LIM), which is responsible for the LD information, such as the amount of the node and task processor load and communication. Finally, the distributed scheduler makes migration decisions, based on the load information, as presented in Figure 2.

3.1 Task migration

Our task migration is implemented on top of the Mach μkernel, in user space. It requires modest mod-

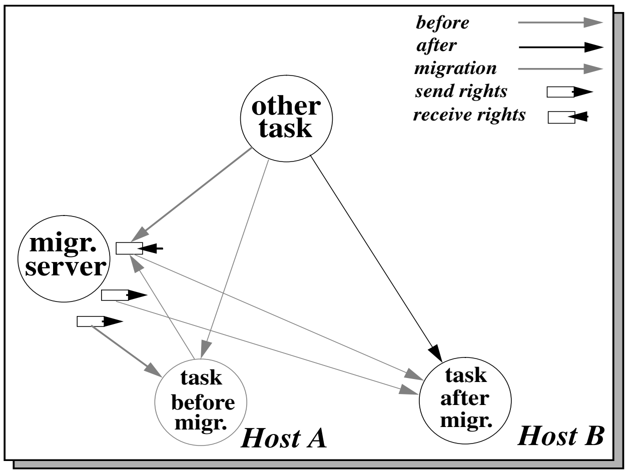

Figure 3: Interposing task kernel port: migration server has send capabilities for local and remote task instances and a receive capability for the task kernel port.

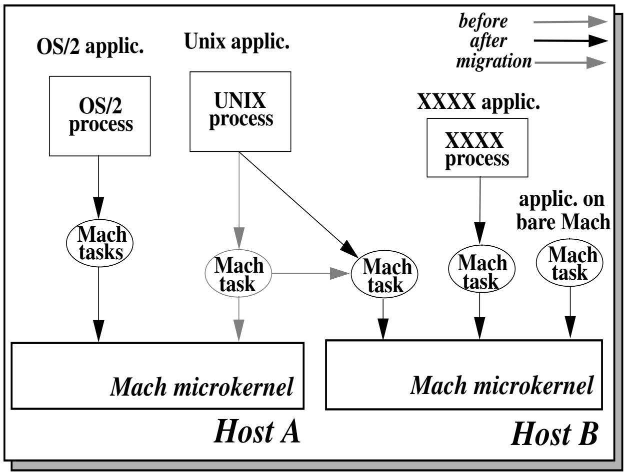

Figure 4: Task migration design: the Mach task is migrated, while the UNIX process remains on the original node, all UNIX related calls are redirected back.

ifications to the μkernel. User space task migration allows us to experiment with various address space transfer strategies [6]. It provides flexibility without significantly paying performance price, and retaining full transparency. Transparency is supported by the network IPC, DSM and interposing task and thread kernel ports. All messages sent to the task capabilities are transparently forwarded to the new node, as supported by the network IPC. After migration the task's address space is accessed copy-on-reference, or it is mapped shared, as supported by DSM.

Interposing the task and thread kernel ports is the only additional support that we introduced. Kernel ports represent kernel objects. Having a capability for the kernel ports allows control on behalf of related objects. When a message is sent to a kernel port, instead of queuing the message, the appropriate kernel procedure is activated on behalf of the kernel object. While it is possible to have the send capability for the kernel port, it is not possible to obtain the receive capability (since the kernel port belongs to the kernel), thereby preventing user space transparent migration. In order to overcome this limitation, we introduced two new system calls for the interposition of the task and thread kernel ports. The original kernel ports are interposed for the new ports. After the interposition only the TM module has the send capability for the new task and thread kernel ports, while the original kernel ports now reside in the capability space of the TM module. The messages sent by other tasks to the original task/thread kernel port are now queued in the message queue, since it resides in the TM module capability space. After migrating all of the task state, the kernel port is interposed back on the destination node and all messages are restarted. Port interposition is described in Figure 3.

The rest of the TM algorithm is relatively simple, we migrate various task state, such as the suspend count, capabilities, thread states, etc. Performing optimizations is more complex. Details on the design, implementation and performance of our TM can be found in [6].

The Mach microkernel was an important factor for the development of task migration. We were able to implement most of the task migration in user space without significantly sacrificing performance and retaining transparency. However, only the task abstraction is migrated, while the OS abstraction remains on the original node, as presented in Figure 4. This solution may suffice for some applications and for some it may not. The crucial factor is the ratio of the OS activities (system calls) v. μkernel activities. If the ratio is high, leaving the OS abstraction on the source node may be too costly. Therefore, it would be also necessary to migrate the process abstraction. This also conforms to fault tolerance requirements. We plan to provide some form of primitive OS abstraction (process) migration.

3.2 Load information management

Load Information Management (LIM) is concerned with extending the Mach μkernel to provide more information on the network IPC and DSM. We assume that, beside the standard information on processor load, the information on IPC and VM are important for realistic assessment of modern distributed applications, which mainly communicate by (distributed) shared memory and (network) IPC. While some information is provided, such as processor load and paging, other information isn't supported at all, e.g. information on the network IPC and DSM. Therefore, we instrumented the kernel to collect such information and implemented routines that provide it to the LD scheduler. Unfortunately, some of the information, such as the local IPC, turned out to be hard to account for without making any significant modifications to the existing kernel.

We instrumented the μkernel to collect information on the node and task level. For the node level we account for the number and size of messages across the network and for the number of pageins and pageouts. Our scheme for information collection is an attempt to tradeoff costs and minimization of the kernel modifications for the completeness of the collected information. Therefore, we do not collect all possible information per task, but only information that is simple and cheap enough to collect. We collect the information on the number of sent messages, and the number of the network pageins. We do not collect information on the number of messages and amount of data received by the task, nor we age this information. This would require more changes to the current kernel and would incur more space and time costs. For example, the correct place to store information on the output IPC traffic would be inside the send capabilities, and neither in the port structures, nor in the task structure, as we currently do. This would, however, require new fields in the capability structure. In order to account for received messages it would be necessary to have a back pointer to the task capability space, which currently does not exist.

If a task communicates with other tasks on more than one underloaded node, it is up to the LD server to make decision what to do. A possible scenario is to migrate to the node with which the task communicated most. Currently, in the task kernel structure we collect information on communication with up to four hosts. Further information could be either discarded or the space could be enlarged dynamically. Figure 5 presents accounting for the network IPC. Accounting for XMM is done in a similar fashion.

Information dissemination is currently provided by periodically circulating the load information for all nodes in the LD cluster. Since we only have three com-

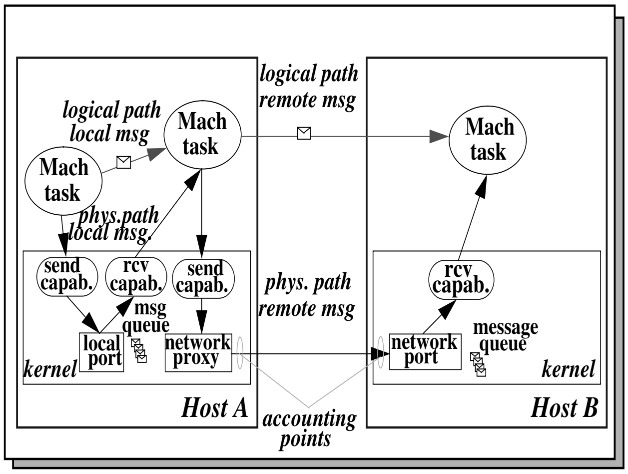

Figure 5: IPC paths and accounting: additional code for accounting on the network IPC is added to out- and incoming part of the network IPC layer.

puters, providing a sophisticated scheme doesn't bring any improvement. However, as soon as we switch to more nodes, we shall use Barak's scheme for information dissemination [1]. During negotiation we inspect the load of the destination node, if paging space is free, etc.

3.3 Distributed scheduling

The main goal of our work on distributed scheduling is not to invent new distributed algorithms, but rather to observe the benefits of the extended load information on various strategies. Most of the existing strategies only consider processor load. This is due to the complexity of collecting the appropriate information on the variety of OS abstractions, such as files, networking, devices and kinds of IPC. We set our priority to observe the μkernel influence on various distributed strategies. We achieve this by introducing three different levels of information. Either no information, only the processor load information, or processor load enhanced by the information on communication. Then we repeat experiments with well known strategies and various levels of considered information.

The distributed scheduling part of our scheme consists of a LD server that runs as a user space task on every node that belongs to the LD cluster. Nodes can leave or join the cluster at any time. In order to allow for the comparison of different LD strategies, the LD server is highly parametrized. It can be dynamically instructed to change its strategy and parameters. We

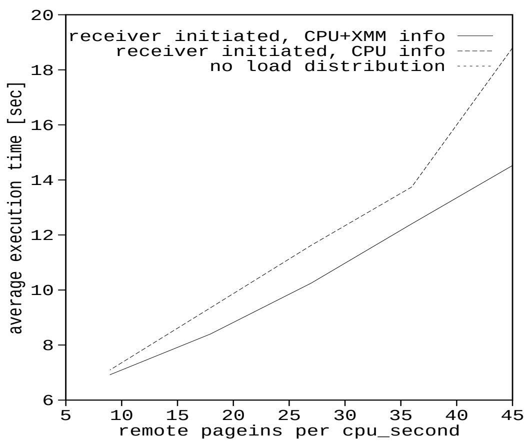

Figure 6: Average execution time as a function of the amount of DSM access.

support the following strategies (default values for parameters are given in braces):

- Random: the LD server periodically checks if the local load is higher than HIGH_THRESH (3 tasks in the run queue), and if true it migrates the longest running task to a randomly chosen node.
- Sender initiated: the LD server periodically checks the local load and if the local load is higher than HIGH_THRESH (2), it polls POLL_LIMIT (2) remote nodes. If for a polled node the load is lower or equal to the local load minus DIFF_THRESH (2), then the longest running task is migrated there.
- Receiver initiated: the LD server periodically checks the local load. If the local load is lower than IDLE_THRESH (2), then POLL_LIMIT (2) nodes are polled. From the first polled node with the local load higher or equal to the receiver load+DIFF_THRESHOLD (2) the longest running task is migrated to the receiving node.
- Symmetrical: combines the receiver and sender initiated strategies. The parameter values are POLL_LIMIT (2) and HIGH_THRESH (3).

The following parameters can be specified for the mentioned strategies:

- Period: the time between load collection (1sec).
- Priority: the Mach priority of the LD server task.
- IPC/XMM: Specifies whether IPC/XMM should be considered when selecting a task for migration. If IPC/XMM are considered, then in the above strategies not the longest running task is selected for migration, but a task running longer than MIN_LIFE_TIME (800ms) with the most IPC/XMM per second of CPU time.
- MIN_LIFE_TIME (800ms): the time a task has to run before it becomes a candidate for migration.
- If remigrations are allowed (default is no).

As mentioned earlier, we still consider processor load as a prime factor for the scheduling decisions. However, we also consider the task communication as a criteria for the selection of the destination node. If there are several candidates for migration, a task is selected according to the migration criteria. Once that the appropriate mechanisms are provided, it is easy to implement adequate scheduling strategy, or even to provide a variety of them and then select the appropriate one according to the application behavior.

4 Performance measurements

Our goal is not to implement any innovative scheduling algorithms, but rather to investigate issues that are inherent to the underlying μkernel environment. In particular, we are interested in showing that the information on the network IPC and DSM, which are the main type of interface in the contemporary μkernels, could improve the LD decisions. Therefore, we conducted experiments with the artificial mix of processor load, network IPC and network paging in order to improve our LD algorithms. The measurements are intended to demonstrate the functionality of our scheme. More detailed measurements, with real applications and more computers will be performed later.

Figure 6 presents the task average execution time as a function of the DSM paging for two LD scheduling strategies. A number of tasks are started on one node and distributed using the sender and receiver initiated strategy. Each task communicates with a random server through DSM. Servers are executed on all nodes. Amount of DSM is expressed as the number of the page faults across the network. The measurements are repeated with or without considering information on DSM. Performance improvement varies with the amount of performed XMM communication, and goes up to the 30%. Similar results are obtained using the network IPC instead of DSM for communication [7].

	functionality	transparency	extensibility	performance	scalability
User space, UX I. (Condor)	poor	poor	excellent	poor	poor
Monolith. UX I. (Sprite)	good	good	poor	excellent	fair
Message based I. (Accent)	good	good	poor	good	good
μkernels (V Kernel)	good	good	fair	good	good
On top of μkernels (Mach)	excellent	good	excellent	good	good

Table 1: Comparison of the LD mechanisms on various architectures.

This is not a surprise, since DSM relies on the network IPC. The figure demonstrates the advantage of considering information on communication for the distributed scheduling. It should be noted, though, that the current network IPC and DSM are unoptimized and that the future implementation may influence the benefits of our approach. However, we believe that similar to the processing power, the network IPC also has some bandwidth, and no matter how much we improve it, there will always be some application that needs more of it. Therefore, considering communication in the distributed scheduling would benefit even for the highly optimized network IPC.

The cost of LD (task migration, load information management and distributed scheduling) has been measured to be approximately 1% of the overall CPU time during the experiment. All measurements have been performed on three 33MHz i80486 based PCs, connected via ethernet. We used Mach NORMA 13 version and UX28 server.

5 Comparison of LD architectures

In order to compare various LD implementations we define few classes of LD and few comparison criteria. We classify LD implementations into: **user space implementations on monolithic kernels with UNIX interface**, such as Condor [5]; **in-kernel implementations for monolithic systems with UNIX interface**, such as Sprite [3]; **in-kernel implementations on message based operating systems**, such as Accent [12]; **in-kernel implementations for μkernels**, such as the V kernel [11]; and **user space implementations on top of μkernels**, such as Mach [6]. The following comparison criteria are used. **Functionality** describes if there are any restrictions on the LD mechanism, e.g. system calls that are not supported. **Transparency** represents if all processes are migratable at any time, if there is a need to recompile or relink the applications, etc. **Extensibility** describes the level at which it is easy to extend/modify the LD scheme, e.g., how much is the LD mechanism insulated from the rest of the OS; **Performance** is represented by the ratio of the local and remote execution times. **Scalability** concerns the LD mechanism, however, it mainly depends on the scalability of the underlying operating system. Architectures are rated by marking them "poor", "fair", "good" and "excellent".

A user space implementation of the LD mechanism is an extensible solution with a limited transparency and functionality and poor performance and scalability. It is suitable only for long running tasks that do not issue various OS calls.

The in-kernel LD mechanism implementations for monolithic kernels provide for excellent performance because they are tightly coupled with the file system implementation, this however, influences extensibility, since process migration is hard to insulate from other parts of kernel. The communication is based on the optimized remote procedure call, which further improves performance, but somewhat affects scalability.

The message based systems improve scalability over monolithic kernels, however, paying the price in performance. A message based interface provides for a powerful, low level tool, which is hard to optimize. The in-kernel implementation poses a problem to extensibility.

LD implementations on μkernels improve extensibility, since they act on μkernel abstractions which are easier to deal with due to their simplicity. They retain good performance and functionality like other in-kernel implementations.

Finally, the LD mechanisms on top of μkernels provide for excellent extensibility like other user space implementations, however, they also have good performance and transparency like in-kernel implementations. Since the LD mechanism is supported in user space, it is relatively easy to support various address space transfer strategies [6], improving its functionality.

The comparison is summarized in Table 1. A description of each implementation and a detailed reasoning for rating are beyond the scope of the paper.

6 Related research

Load balancing on **Chorus** [8] provides for process migrations, but with a lot of μkernel issues, such as migrating capabilities, threads, and thread state. The **Stealth project** [4] depresses the processor and VM priorities of incoming tasks so that they don't interfere with the local tasks. This requires modifications to the inode and default pager. The **TNC** project [13] supports UNIX process migration, considering the μkernel level just as necessary, for example, thread state, VM and necessary capabilities are migrated, but neither the whole IPC space, nor the kernel ports for the task and threads.

7 Conclusion

We showed that the Mach μkernel is a suitable environment for a LD implementation, due to the ease of the TM implementation and the adequate load information management. While systems, such as Sprite, MOS(IX) and Condor, already achieved some level of adoption in local academic circles, a μkernel LD implementation should widen the LD functionality, improve portability and be of influence to a wider LD adoption. We demonstrated it by a prototype implementation, circumventing the problems that were obstacles to previous implementations, such as the simple user space task migration implementation and effective load information management.

We plan to use our LD scheme in a larger environment than the current one. Using three computers and simple artificial load was enough to demonstrate the functionality of our scheme, but it is certainly not enough for more detailed performance measurements. We also plan to switch to real applications. Finally, we are currently investigating process migration as an extension to our scheme for the cases where task migration doesn't meet performance or fault tolerance requirements.

Acknowledgements

We would like to thank Mike Kupfer for a significant help in reviewing the paper.

References

[1] A. Barak and A. Shiloh. A Distributed Load-Balancing Policy for a Multicomputer. *Software-Practice and Experience,* 15(9):901–913, September 1985.

[2] D. Black, et al. Microkernel Operating System Architecture and Mach. *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures,* pages 11–30, April 1992.

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

[5] M. Litzkow, M. Livny, and M. Mutka. Condor - a Hunter of Idle Workstations. *Proceedings of the 8th International Conference on Distributed Computing Systems,* pages 104–111, Jun 1988.

[6] D. Milojičić, W. Zint, A. Dangel, and P. Giese. Task Migration on the Top of the Mach Microkernel. *Proceedings of the Third USENIX Mach Symposium,* pages 273–290, April 1993.

[7] D. Milojičić, P. Giese and W. Zint. Experience with Load Distribution on Top of the Mach Microkernel. *Submitted for the Third USENIX SEDMS Symposium,* September 1993.

[8] L. Philippe. Contribution á l'Étude et la Réalisation d'un Système d'Exploitation à Image Unique pour Multicalculateur. *Technical Report 308, PhD Thesis, Université de Franche-comté,* 1993.

[9] M. Rozier. Overview of the Chorus Distributed Operating System. *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures,* pages 39–70, April 27-28, 1992.

[10] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience,* pages 315–339, December 1990.

[11] M. Theimer, K. Lantz, and D. Cheriton. Preemptable Remote Execution Facilities for the V System. *Proceedings of the 10th ACM Symposium on OS Principles,* pages 2–12, 1985.

[12] E. Zayas, Attacking the Process Migration Bottleneck. *Proceedings of the 11th Symposium on Operating Systems Principles,* pages 13–24, November 1987.

[13] R. Zajcew, et al. An OSF/1 UNIX for Massively Parallel Multicomputers. *Proceedings of the Winter USENIX Conference,* pages 449–468, January 1993.