Experiences with Load Distribution on top of the Mach Microkernel*

Dejan S. Milojičić†, Peter Giese and Wolfgang Zint

University of Kaiserslautern, Informatik, Geb. 36, Zim. 424
Erwin-Schrödingerstraße, 67663 Kaiserslautern, Germany
e-mail: [dejan,giese,zint]@informatik.uni-kl.de

Paper presented at the fourth USENIX SEDMS Symposium, San Diego, September 1993

Abstract

The paper describes our experiences in the design, implementation and use of load distribution on top of Mach. As a first step towards load distribution, we provided task migration which is our base mechanism for distributed scheduling. We compared task migration with initial placement. In order to make more accurate scheduling decisions, we instrumented Mach to account for network IPC and network paging. Processing is still the prevailing factor, but we also consider information on VM and IPC. We have conducted experiments with well-known distributed scheduling strategies in order to prove our assumptions. We are primarily interested in μkernel aspects of load distribution. We report on the lessons learned while dealing with the Mach interface and on task migration relationship to process migration and the file system.

1 Introduction

Load Distribution (LD) has always attracted the interest of the research community. Unfortunately, it has never been widely used, despite many successful implementations [Bara85, Doug91, Zajc93] and promising simulation research [Krue88, Zhou88, Krem92]. One of the reasons for the modest use of LD may be the absence of widely used distributed operating systems and adequate parallel applications. New μkernels, such as Mach [Blac92] and Chorus [Rozi92], are inherently distributed. New distributed applications are evolving, such as PVM [Sund90]. In this new environment, we expect new opportunities for LD. There are also significant improvements in hardware architectures. Massively Parallel Processors (MPP) and mobile computers may be yet another reason for a wider LD use.

In our research we mainly target operating system issues in LD. New operating systems tend to be small μkernels with various servers, usually running in user space [Golu90, Rozi92, Cher90]. On top of μkernels there are emulations of various operating system personalities, such as BSD UNIX [Golu90], AT&T UNIX V [Rozi92, Cher90], OS/2 [Phel93] and Sprite [Kupf93]. Contemporary applications communicate using two types of interface: message based and (Distributed) Shared Memory (DSM), as presented in Figure 1.

*Research is supported by DAAD, University of Kaiserslautern (Germany) and Institute "Mihajlo Pupin".

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

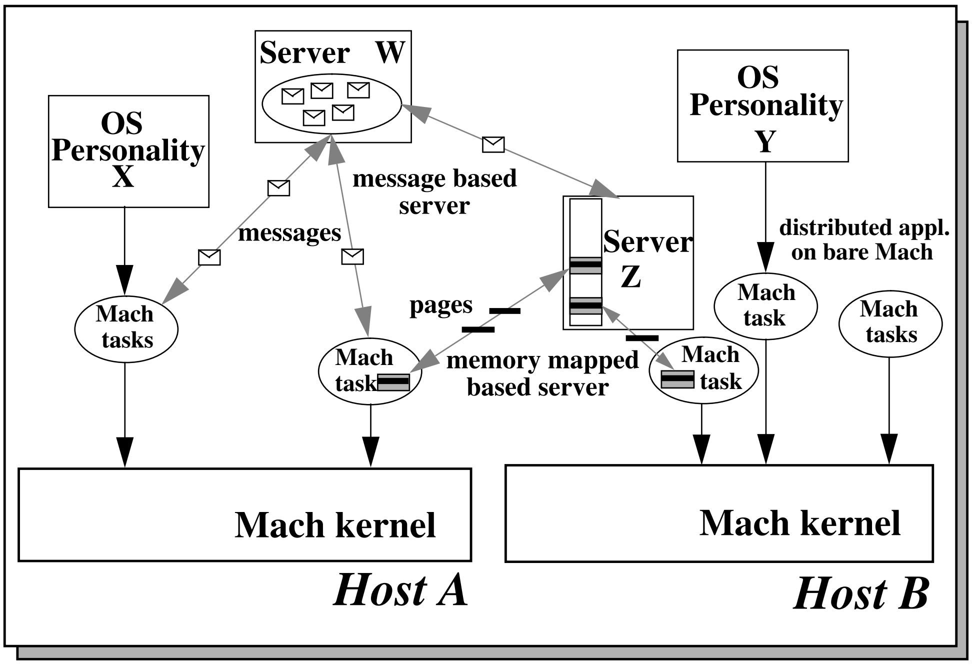

Figure 1: Common Types of NORMA Applications

Compared to earlier systems which supported process migration, μkernels also provide task migration. Task migration represents moving a task abstraction from a node to another, transparently to the task itself as well as to other tasks in a distributed system. UNIX process is mapped to a task with a single thread of control. Process migration incurs transfer of more state, such as file descriptors and enabled signals. What are the implications of this environment for LD design? We claim the following issues to be important:

- LD should be performed on top of the μkernel, which is a common denominator for all running OS personalities.
- Besides traditional information, such as processing and paging, LD should consider information on network IPC and distributed shared memory.
- LD should exploit all possible mechanisms, such as process migration and particularly Task Migration (TM). Migration has often been neglected in favor of initial placement.

In order to verify these hypothesis, we designed and implemented LD on top of the Mach μkernel. Our work consists of three phases. First we implemented task migration [Milo93] which is our main mechanism for LD. User space task migration allows us to experiment with various address space migration strategies. It provides the necessary flexibility without paying a significant performance price, and while retaining full transparency. It should be noted, though that compared to process migration we have to deal with less state. For the time being we do not migrate UNIX processes which remain on the source node.

As the second step, we exported the information necessary for LD decisions [Milo93a]. We instrumented Mach to account for network IPC and DSM.

Finally, we implemented a distributed scheduler which uses task migration as a mechanism, and takes advantage of load information in order to make more accurate scheduling decisions [Milo93a]. In this paper we report on the experiences obtained during our project. Experiences are related to load distribution, as well as to μkernel issues.

The remainder of the paper is organized in the following manner. In Section 2 we present background and previous work. In Section 3 we discuss design and implementation of our LD scheme. Lessons learned are summarized in Section 4. Finally, in Section 5 we draw conclusions and suggest future work.

2 Background and Previous Work

Due to the implementation character of the project, particularly task migration, our research is on the border between the fields of distributed scheduling and operating systems. Therefore, there is a lot of previous work in this area. We try to mention only the ones most related to our research. In this section we shall describe the Mach μkernel and mention important LD systems that influenced our work.

2.1 Mach

In this subsection, we shall briefly mention a few important Mach characteristics and describe Mach NORMA version. Interested readers may consult the extensive Mach literature for more information [Blac92, Boyk93].

The Mach μkernel is well known for its portability. It is ported to various multiprocessor computers (shared memory with (non)uniform access and non shared memory, such as MPP architectures) and distributed systems. Its extensibility is proved by extensive research conducted on top of Mach, such as X-kernel protocols [Orma93] and scheduler activations [Bart93], although some experiments showed that Mach does not fit the requirements of all architectures sufficiently well [Kupf93]. There are particular Mach versions extended for real-time [Toku90] and fault-tolerance [Chen90]. Its modularity is expressed by the minimal set of supported abstractions: task and thread abstraction for processing, ports and messages for IPC and memory objects for VM.

The Mach μkernel is transparently extended to a distributed system with NORMA support for network IPC [Barr91] and distributed shared memory. NORMA IPC is an in-kernel implementation of the user space network message (*Netmsg*) server [Juli89]. While the *Netmsg* server provides functionality, the in-kernel network IPC is an attempt to improve performance. It is optimized for the short in-line (less then 128 bytes) and large out-of-line messages (one or two pages). In Mach, the out-of-line messages are logically copied, avoiding physical copying. The NORMA network IPC provides a distributed capability space, taking care of notifications for deleted capabilities and reference counting.

The NORMA distributed shared memory, called XMM (eXtended Memory Management), is an in-kernel reimplementation of the user space DSM [Fori89]. XMM extends the consistency semantics of the current pagers (e.g. inode and default pager) to a distributed environment. In order to allow the existing pagers to support multiple kernels without XMM, significant changes would be required, incurring incompatibility. XMM overcomes this problem by transparently interposing between the pager and multiple kernels.

2.2 Previous Work

One of the most complete load balancing schemes, including process migration and sophisticated load information dissemination, was developed for **MOS(IX)** system [Bara85]. MOS(IX) is one of the first μkernel-like system, since it is divided into two layers. The *precopy* technique is implemented in the **V kernel** [Thei85], for one of the first task migration implementations on a message passing μkernel. The *precopy* technique improved the freeze time (period while process is not active) but it negatively influenced the overall system performance. Significant performance improvements for address space transfer is achieved by the *copy-on-reference* technique which is introduced in **Accent** [Zaya87]. Address space is virtually mapped and pages are only transferred when they are referenced. **Sprite** process migration contributes the idea of the home node that maintains the state of the process [Doug91]. Load distribution is supported through the use of parallel make. The other interesting issues concern the *flushing* technique for the address space migration, and optimizations based on its relationship to the file system.

The relevant research in the area of distributed scheduling is the load balancing work conducted by **Ferrari and Zhou** [Zhou88]. The authors investigated load indices for various scheduling strategies. Of particular interest is the work done by **Krueger** whose PhD thesis represents an excellent overview of the issues involved in the field of load distribution [Krue88]. The author compared load balancing and load sharing, preemptive and nonpreemptive load distribution and other objectives of load distribution in distributed systems. **Cabrera** measures the typical task execution time [Cabr86]. He found that the most UNIX processes are short lived, e.g. more than 78% of the observed processes have lifetime shorter than $1s$, and 97% shorter than $4s$. Of historical importance is the work done by **Eager et al** [Eage86] which demonstrate that already a small amount of information could lead to dramatic performance improvements.

The more recent and closely related research to our work is the following. Load balancing in **Chorus** is based on processes migration but otherwise it has similar background as our work [Phil93]. Similar problems are solved: migrating capabilities, threads, etc. The **Stealth** project introduces depressing the priority of processing and VM activities of the incoming tasks [Krue91]. This requires modifications to the inode and default pager. It is opposite to what we do. We do load balancing, targeted for clusters and MPP architectures, and therefore we try to distribute types of load (processing, VM and IPC). The goal of Stealth is load sharing on autonomous workstations therefore incoming load (processing and VM) is depressed so that it does not influence the local computations. In Locus **Transparent Network Computing (TNC)** migration remains at the OS personality level, considering the μkernel layer only when necessary [Zajc93]. Interesting contributions of TNC are *Vprocs* and the work on streams migration. Extending TNC with our TM scheme would provide a complete migration solution.

3 Load Distribution Design and Implementation

Our LD scheme consists of three major elements: task migration, load information management and distributed scheduling. Each part is briefly described in Subsections 3.1, 3.2 and 3.3 respectively. More detailed descriptions can be found in [Milo93, Milo93a]. Subsection 3.4 overviews implementation history and environment.

3.1 Task Migration

Our task migration deals with the Mach task abstraction, leaving the OS personality abstraction (e.g. a UNIX process) on the source machine. This introduces a kind of home dependency, since most OS personality calls are redirected back to the source node.

We designed TM in user space. Unfortunately, some modifications to the kernel are necessary. These are however minor (200 lines of code) and originate from a Mach limitation with regard to the task and thread kernel ports. The kernel ports represent kernel objects. User tasks control kernel objects by means of sending a message to the corresponding send capability for the kernel port. The message is intercepted by the kernel which recognizes that it is directed to a kernel object and instead of queuing the message, the appropriate kernel procedure is invoked on behalf of the object. Kernel ports do not have the corresponding receive capabilities, since they are owned by the kernel. With the current Mach interface therefore it is not possible to migrate the task and thread kernel ports from within user space. We provide two additional calls for interposing the task and thread kernel ports. The interpose calls take as an argument the interpose port. The interpose port replaces the original task kernel port, which is returned as an output parameter of the interpose call. After the interpose call, only the caller has the send capability for the migrated task, and the communication directed towards the task ends up in the original task kernel port that now resides in the capability space of the calling task. After migration, the original kernel port is interposed back, and the collected messages are restarted in order to invoke the appropriate actions on behalf of the new migrated task instance.

Our TM scheme is transparent, we can migrate at any point of time. If a thread in the migrated task executes a system call, it is aborted and brought to a clean point where it does not contain any kernel state. This is supported by the Mach system call *thread_abort*. Typical examples are sending or receiving messages, or page faults. In each case, action is either awaited to finish, if it would end in the guaranteed short time, or interrupted otherwise. If it was interrupted, the system call is restarted after migration, which is handled by the system call library.

There are no limitations to the system calls that a task may issue, which is not true for most other user space implementations, such as Condor [Litz92]. Like other user-space implementations, our TM scheme is easy to extend and modify. Like in-kernel implementations, we do not sacrifice performance, transparency and functionality. Therefore, we manage to combine most of the good characteristics of both kernel and user space implementations.

Our TM scheme significantly benefits from the Mach network IPC and DSM. The DSM support is a prerequisite when a part of the task address space is shared or needs some kind of consistency (not provided by the default pager), as in the case of the mapped files. We implemented two kinds of TM servers, a Simple Migration Server (SMS) and an Optimized Migration Server (OMS). OMS supports various address space transfer strategies, such as *copy-on-reference, precopy, flushing* and *eager copy*, in a similar way to implementations in systems such as Accent, V kernel, Sprite and MOS(IX). However, in OMS these strategies are supported in user space (unless there is a need for DSM, where we rely on NORMA DSM). SMS was recently reimplemented in the kernel. The integration of SMS into the kernel is discussed in Section 4. Only SMS is used for LD experiments, due to its simplicity and robustness. OMS is too complex and the in-kernel migration is not mature enough. A more detailed description of SMS and OMS could be found in [Milo93].

3.2 Load Information Management

Our load information management scheme is similar to other implementations, such as TNC [Zajc93] and MOS [Bara85]. It consists of information collection, dissemination and negotiation. It differs from the other schemes in the level at which it is performed, and in the kind of the information it is based on. Beside processing, which is regarded as the prime factor for distributed scheduling, we also consider information on VM and IPC.

We instrumented Mach to collect information on the network paging and network IPC. Information is collected at the node and task level. At the node level we account for the number and the size of network messages, pagein/pageout requests and the number of in/out migrations.

Our scheme for information collection is an attempt to tradeoff the completeness of the collected information for the costs and minimum modifications to the underlying kernel. Therefore, we do not collect all possible information on the task level but only the amount that is simple and cheap enough to collect. We account for the number of remote messages sent from the task and the number of remote pageins. It is too costly to account for the received messages in the current Mach implementation because there is no back pointer from the Mach port to the task. Therefore, the only opportunity is to store the information in ports and then to loop through the entire task capability space, which may be time-expensive if we assume searching of many tasks, and space-expensive, since we need to reserve space in each port instead of only once in the task. It is impossible and inappropriate to account for remote pageouts on behalf of a task. It is impossible because there are no back pointers from the page to the task. It is inappropriate because, if a few tasks share the same page, all of them will be accounted for pageout, although in reality only one of them might have accessed the page, in some sense bearing the responsibility for pageout.

We currently use only three nodes for measurements, and therefore we disseminate information with a circulating token for the strategies that rely on periodic information exchange. Due to the small number of nodes, there is no significant overhead in the periodic circulation of the token, however as soon as we start using more nodes, we shall switch to the dissemination scheme as used for MOS(IX) [Bara85].

3.3 Distributed Scheduling

Our distributed scheduler (LD server) is a user level program running on every node in the LD cluster. The nodes can join or leave the cluster at any time. The LD server is a Mach application which communicates using Mach IPC. In order to allow for the comparison of the different LD strategies, the LD Server is highly parameterized. We can specify the following input parameters to the scheduler:

- The type of the strategy: **no LD** is the case when no load distribution is performed; **random** strategy is activated if the local load exceeds a threshold, in which case the tasks are distributed randomly onto other nodes in the cluster without considering the load of the node we migrate to; **sender initiated** strategy polls a specified number of nodes in order to find a suitable one; in **receiver initiated** strategy, nodes that have lower load than threshold try to find an overloaded node; the **symmetrical** strategy is a combination of the sender and receiver initiated strategies; etc.

- The level of considered information: no information at all; only processing; information on processing, network IPC and XMM.

- Strategy specific parameters, such as thresholds, frequency of load collection and dissemination, server priority, accumulated task user time before being considered for TM, etc.

The LD server periodically inspects the load on the local node using the load information management interface. If the local load crosses a threshold value, the LD server acts according to the specified strategy. Task migration is our basic mechanism for LD. Based on the specified criteria a task is selected and if appropriate migrated to a suitable node. Depending on the underlying strategy, negotiation takes place before migration in order to find a destination node or to verify its suitability and willingness to accept the migrated task.

3.4 Implementation History and Environment

We started our LD project in the fall of 1991. We migrated a task for the first time in May 1992. Migration was stable after a few months of improvement and with more robust versions of the kernel. Load information management was finished by the end of 1992. We conducted load distribution experiments since the beginning of 1993. New strategies are continuously added for new experiments.

The underlying environment consists of 3 PCs interconnected via Ethernet. Each PC contains a 33MHz i80486 processor with 8 MB RAM and a 400MB SCSI disk.

For the implementation and experiments we used Mach NORMA versions 7, 12, 13 and 14, and the UNIX server UX28. Currently we are moving to the OSF/1 environment. The routines concerning kernel modifications fit in a file of 200 lines of C code, most of which are comments, debugging code and assertions. The SMS server has around 600 lines of code, while the LD server and load information management part consist of around 1800 lines.

For most of our measurements we used the artificial load and our computing environment as a testbed. At the very beginning we intended to use real applications, however it was quite hard to find any appropriate distributed application. PVM was being ported to Mach but was still not available. Most other applications required porting to Mach. UNIX applications are not suitable since we do not support process migration. Therefore, we implemented an Artificial Load Task (ALT). ALT is an attempt to provide a simple and reproducible behavior of processing, IPC and VM. Load is specified by the parameters for the processing, (network) IPC and VM (XMM) access. Processing is specified by the amount of CPU user time, IPC as the number of messages, and XMM, as the number of network pageins. VM access and IPC are equally distributed over processing time, and their amount per unit of time is constant, same for all ALTs. For each node in the cluster the load is separately specified. If not otherwise noted, the mean interarrival time of the ALTs on each node is computed by dividing the mean user-time of the tasks (given by the hyperexponential distribution) by the load for the specific node. The ALT interarrival times are drawn from an exponential distribution with the above computed mean interarrival time. The node of the server which ALT communicates with is randomly chosen and remains the same throughout the task's lifetime. In the presented experiments, tasks are migrated at most once. Tasks can not be migrated before they accumulate $800ms$ of user time.

4 Lessons Learned

Our research project is a practical one. During its life, we dealt with various μkernel and load distribution issues. This section summarizes the lessons we have learned while developing and using LD on top of Mach. Throughout the section, we refer specifically to our TM scheme and to the Mach μkernel, however similar reasoning could be applied to other μkernels as well. We would like to discuss the following observations:

1. Task migration is easy to implement and insulate.
2. User space and in-kernel TM are similar regarding performance and implementation, they differ in maintainability, interface and kernel integrity.
3. Task migration is not necessarily inferior to initial placement.
4. Task migration is not always enough.
5. Network IPC is powerful but also complex to implement and optimize.
6. Information on IPC and VM improves scheduling decisions.

The presented measurements have been conducted on our testbed of three computers. Although it may seem that having only three computers limits the generality of our conclusions, we do not believe that the rather small configuration has the significant impact in this case. Our scheme is designed to be scalable.

However, the poor performance and functionality of network IPC may mean that some improvements that we have achieved may vanish for the better network IPC implementation. We believe though that like the newer and faster processors have not eliminated the need for load distribution, similarly the better implementation of network IPC would not render the communication optimization useless.

Task Migration is Easy to Implement and Insulate

By implementing two user space task migration servers, as well as in-kernel task migration, we have demonstrated that there are no significant complexities involved in the TM design and implementation. The changes to Mach for a user-space implementation are minor. The main effort for moving user space TM into the kernel consisted of changing the interface.

The Mach object orientedness allows for much easier design. Transparently accessing the Mach objects across the network simplifies the effort of controlling (extracting, inserting and accessing) various objects in the migrated task.

Insulating process/task migration from the other modules in the system may be hard to achieve. Douglis reports that it is hard to insulate process migration from the other modules in Sprite, due to the significant dependencies [Doug91]. Compared to other process/task migration experiences, we had no problems with insulating task migration from the other modules. We make a potentially unfair comparison between process and task migration because there are differences that may affect our conclusions. We do plan to upgrade our scheme with process migration as well, and only then we could firmly prove some of our claims. However, there are also enough reasons that we may already predict a lot of behavior, since most of the state is already contained within the task. For instance, OSF/1 AD task contains most of the process state that is residing within the emulator. Although the OSF/1 environment is moving towards the emulator-free task [Pati93], in which case all of the process state will reside in process manager, similar activities would be required to extract and migrate the required state.

While designing and implementing TM, the NORMA interface (network IPC and XMM) has been of the significant help. Similar development history exists for other systems. For example, process migration in Sprite [Doug91] relies on the file system implementation [Welc90], particularly on migration of the opened I/O streams. Load distribution for the V kernel [Stum88] is based on the related work of task migration [Thei85]. Similarly, we make use of NORMA IPC and XMM for the implementation of our TM and LD. The simplicity of our task migration stems from the NORMA support.

The Mach NORMA version provides a message based and a memory mapped interface, compared to the traditional UNIX-like *open-close-read-write-ioctl* interface. Douglis compares the UNIX interface with the message based interface and derives the conclusion that although on the surface it seems that the message based interface simplifies state management problems, there is not really too much difference between using the two types of interface from the complexity point of view [Doug91]. He admits, though that the message based systems make migration somewhat easier.

Another Sprite implementor argues that message based interface is a general tool, providing powerful functionality at the lower level [Welc93]. The UNIX-like interface, however being at a higher level, provides for various optimizations for the particular implementation, e.g. of a file system. Sprite, a representative of the UNIX-like interface, proved this opinion to be correct.

Our findings are in line with both Sprite implementors. We argue that our Mach task migration was easy to implement but performance optimizations are limited to improving NORMA functionality.

User Space v. In-kernel Task Migration

After two versions of migration servers we implemented task migration in the kernel. The in-kernel task migration is achieved by moving SMS into the kernel. This required switching to the appropriate in-kernel interface and we also applied some optimizations. The performance is better compared to the user space SMS server, and it is in the range of the performance of OMS. The performance improvement is not the consequence of running in the kernel space but rather due to the various optimizations. The dominant costs for migration are the network messages. Since SMS is deliberately unoptimized and relies only on the LD server on one node, it involves a new message for each state transfer, incurring high costs. This was, however the matter of choice, and is certainly not inherent to user space implementation. In OMS, for example, there are servers on each node which cooperate in packing and unpacking of the task state in messages, thereby improving performance. For the in-kernel migration, the kernels play the role of the servers, and therefore allow for the optimization by packing more state into one message. OMS is not moved to the kernel since its optimizations are too complex (e.g. packing more capabilities into a message) which would incur too much complexity for the in-kernel implementation. Besides, the overhead of initial costs is not significant, therefore the tradeoff of simplicity for performance is reasonable.

Most of the SMS in-kernel reimplementation complexity involves optimization, e.g. if there are several send capabilities for a port, only one send capability and the reference count are migrated, instead of migrating each send capability; the thread states are combined into an array instead of migrating each state separately, etc. The implementation effort was the

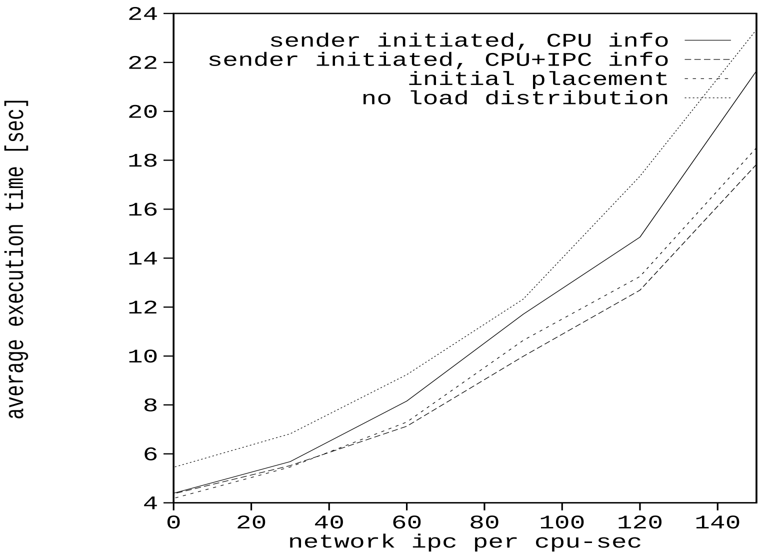

Figure 2: Average Execution Time as a Function of the Amount of IPC for Initial Placement and Task Migration.

matter of hours and debugging the matter of days. The ease of in-kernel implementation further demonstrates modularity of the Mach μkernel, although it also owes a lot to the expertise achieved during user-space development.

The in-kernel TM is developed due to the interpose calls. In a way, interpose calls represent an attack on the kernel integrity, and certainly break the current Mach interface. Therefore, we do not believe that these calls could ever progress into the standard Mach distribution, contrary to our intentions to provide TM to a wider community. Performing kernel port interposition as a part of the in-kernel migration does not raise concerns.

There are advantages and disadvantages for both the user space and in-kernel task migration. User space migration provides for easier experiments with various address space migration strategies. For example, in OMS we support *flushing, precopy, copy-on-reference*, and other strategies. On the other hand, since in some cases DSM is needed, it is better to use the in-kernel address space transfer or to have some other kind of DSM support, such as the one described in [Fori89]. From the design point of view, we do not regard the difference between the in-kernel and user space migration as a significant one. Having both implementations available we shall collect some more data and observe the (dis)advantages of each version.

Task Migration is not Necessarily Inferior to Initial Placement

There has been analytical, practical and simulation work that demonstrated that process migration is not suitable for LD [Eage88, Doug91, Shiv92]. Most researchers would start their reports by stating that process migration, despite being a useful tool, still involves the significant complexities for the implementation and incurs sufficient costs that render it useless for LD. One of the researchers who considers process migration advantages [Krue88] is mostly worried about the operating system complexities involved with process migration.

We try to prove that the above observations are not always valid for our task migration.

We showed that our task migration implementations do not involve any significant complexity, on the contrary, we provide both user space and in-kernel task migration facilities without significant effort. With regard to the costs, since our task migration supports *copy-on-reference* data transfer scheme, the initial migration costs are comparable with the costs of initial placement. Run time costs for message redirection can be neglected. The only significant costs may be due to the lazy copying of pages. It is true that an inconvenient access pattern could result in higher costs for TM. On the other hand, only referenced pages are transferred, and we may potentially predict the task behavior based on the size of its address space and its resident set size.

On the positive side, task migration provides advantages due to the additional information a running task can provide, such as the information on who the task communicates with, and the task execution time. Using TM is an implicit way to filter out short-running tasks not suitable for migration. Besides, migration is the only appropriate mechanism for receiver initiated strategy, a preferred strategy for systems with higher load [Krue88].

We conducted an experiment to compare task migration with initial placement. Once started, a task can provide more information on its behavior. For example, we can find out who and where a task communicates with, and potentially migrate the task to the appropriate location. Initial placement does not offer such an advantage, since we know nothing about the task behavior in advance. Instead of initial placement, which would involve accessing the code pages remotely, we use remote invocation (sending just parameters, while the text segment is not migrated but resides on each node), which represents the lower bound cost for initial placement. Figure 2 presents the average execution time of the tasks as a function of the amount of IPC they perform per second of the user-time. The tasks are started on all three nodes as described in subsection 3.4. Nodes are unequally loaded with workload of 0.8, 0.5 and 0.1. Initial placement is event driven. When a task is created, it is initially placed on an underloaded node. In the case of migration, tasks can be migrated if they execute longer than $800ms$. Load distribution is started each second, and if appropriate, a task is migrated.

We can notice that the sender initiated algorithm (based on TM) does not have much worse performance than initial placement, while the sender initiated algorithm that also considers information on IPC slightly outperforms initial placement for more intensive IPC traffic. All three cases are better then the case when no LD is performed.

The reasons why we do not obtain even better performance improvements are twofold. The first reason has to do with NORMA IPC. In the overloaded host, NORMA IPC is a bottleneck leading to unacceptable migration times. Whereas on the underloaded host (from NORMA IPC point of view) migration lasts few hundred ms, on the overloaded host it goes up to few s, which increases the task average execution time and decreases the benefits obtained by considering IPC. The second reason deals with the small number of nodes that we are currently using. In the case of initial placement, there is a high probability that a suitable node with respect to IPC traffic is selected. With more hosts, the probability would be much lower, and the benefits of having the correct information on communication in the case of task migration will be more expressed.

Task Migration is not Always Enough

Extending the environment running on Mach to a distributed system is not painless. There are many issues that require significant modifications in order to satisfy functionality and performance requirements. Some of them include the file system, process migration, emulator, etc. Our task migration scheme is a Mach level mechanism. In the case that the user applications need a UNIX interface, TM is not enough. We observed that the performance penalties due to home dependency could be significant enough to render task migration useless (for experimental results see [Milo93]). In such cases it is necessary to provide the OS personality abstraction migration. In other cases, though task migration can still suffice as a lone migration mechanism. In MPP, for example, it may be inappropriate to have an OS personality server on each node (due to paging, memory consumption, etc.). In such cases, providing the Mach interfaces may be enough, possibly upgrading it with a library which would emulate the UNIX-like VM. Many applications conform to this requirement, e.g. simulations or numerical computations which generally do not have the need for file access (except for the initiation and termination phases), or other UNIX interfaces.

There are a few possibilities to combine task and process migration. Once we select a task to be migrated, we can inform the OS personality to migrate its related abstraction, e.g. after the task is migrated it may be appropriate also to migrate the OS personality abstraction. The reverse strategy is also possible, an OS personality decides to migrate its process abstraction, consults the load information module, and then performs task and process migration. Finally, a process abstraction can remain on the source host if performance requirements and task behavior allow for that.

Files are more related to the process than to the task abstraction, but we are certainly concerned with the file influence on task migration. In the current TM scheme, files are transparently supported by using DSM. Since files are mapped into the task address space, after task migration they are mapped as shared, retaining the necessary consistency. Therefore, we can still benefit within the UNIX environment, although only the task, and not the process abstraction, is migrated. Functionally this solution suffices but it is unacceptable from the performance point of view. There are two related activities in this area. In the OSF/1 AD operating system a significant effort is invested in Distributed File System (DFS) support [Zajc93]. Although targeted for NORMA architectures in general, its main application is foreseen for MPP architectures. Another file system is being developed particularly for clusters [Roga93]. Since most of the functionality in either of the two DFS implementations is primarily based on XMM and NORMA IPC, we do not see any limitations on the process/task migration. Similarly to the existing implementation, files will be supported by remapping memory mapped areas from one node to another (thereby retaining consistency), and by migrating the capabilities that represent opened files.

Compared to the Sprite experience, where migrating I/O streams (files, devices, etc.) is a primary source of the complexity, migrating Mach tasks with the opened files is mainly supported by NORMA IPC. However, performance is still concern. While in Sprite there is an optimized interaction between the file system and process migration, the question is still open of how well does the current Mach interface (XMM and NORMA IPC) match the needs of DFS. It should be thoroughly tested and measured in order to verify all the performance issues related to DFS activities, and related to the interaction with the task/process migration. Our current efforts to provide at least a partial process migration which would interact with the cluster DFS [Roga93] are in line with this investigation.

Network IPC is Powerful but also Complex to Implement and Optimize

Our experience with NORMA IPC shows that it has neither been thoroughly optimized, nor fully debugged. While using it we encountered bugs and performance drawbacks. As a message based interface, it is more comfortable compared to RPC, in particular from the

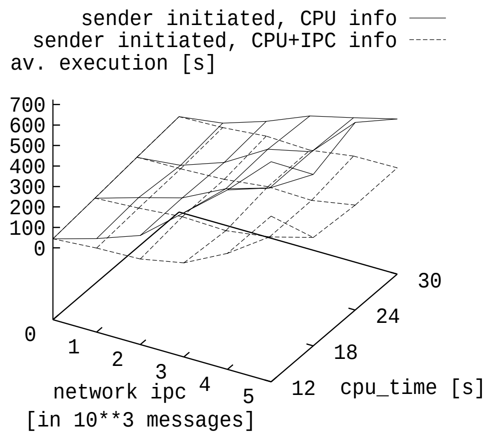

Figure 3. Average Execution Time as a Function of the Amount of IPC and Processing.

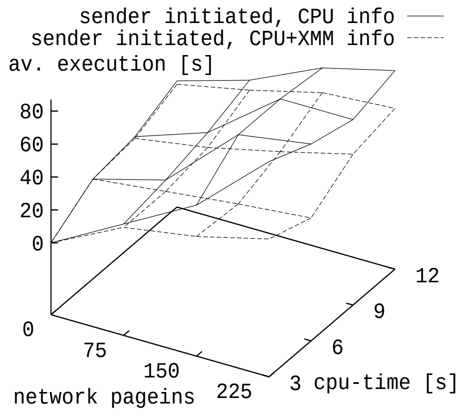

Figure 4. Average Execution Time as a Function of the Amount of XMM and Processing.

aspects of transparency and scalability however its design and implementation are fairly complex. Once that it is correctly supported, there is a big question of providing the adequate performance. We noticed a few performance drawbacks. For example, in the case that a send-once capability (a special case of send capability used for only one message transmission) stems from a migrated receive capability, the existing implementation requires an additional message to the source node for this send-once capability unless there is already a proxy port and the correct node has been set. The need for the additional message is quite a frequent case, resulting in a double cost for the send-once capability transfer.

We also observed a problem related to stressing IPC activity. TM can be significantly delayed when IPC is stressed. Delays can be up to a few seconds, which is unacceptable. This raises the question of a prioritized IPC. Since NORMA IPC has not been optimized (currently it runs in a stop-and-wait mode, there is no sliding window, no piggy-backing, etc.), the behavior might be somewhat exaggerated, however even with an optimized implementation, delays would probably be observable.

Prioritizing Mach processing and paging is done in Stealth project [Krue91]. The migrated-in tasks have depressed priority in order to reduce their influence on the local tasks. Similar reasoning could be applied to IPC.

The presented problems and drawbacks are intended to criticize neither message based interface nor its current implementation, but rather to show the examples of the complexities involved in providing a correct and well-performing message based interface. Despite our objections, we significantly benefited from the existing NORMA IPC and our task would have been much harder without it.

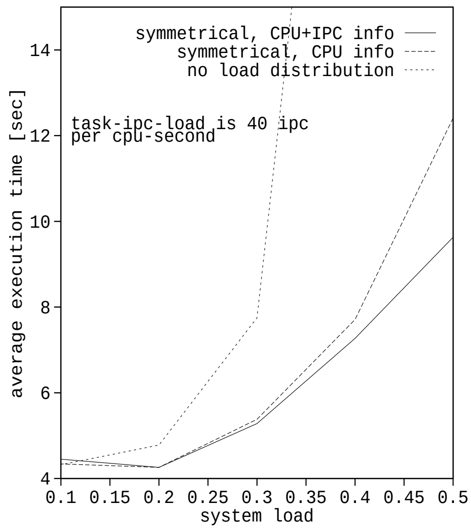

Figure 5. Average Task Execution Time for Symmetrical Strategy v. System Load.

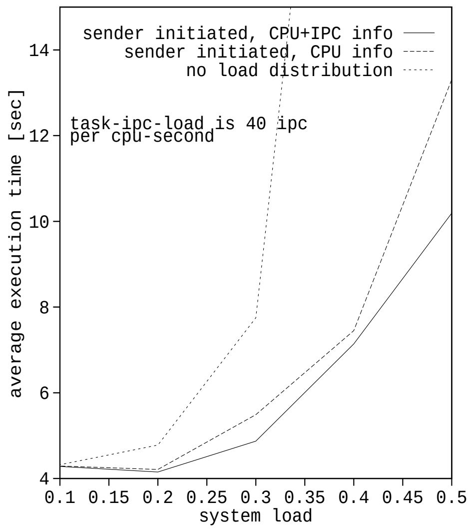

Figure 6. Average Execution Time for Sender Initiated Strategy v. System Load.

Information on IPC and VM Improves Scheduling Decisions

Traditionally, processing load has been the main source for load distribution decisions. Other factors, such as files, device access, virtual memory, networking etc., have been considered, but rarely used in practice. In μkernels, all operating system personality abstractions, such as files, flavors of IPC and device access, are mapped to μkernel abstractions, e.g. in Mach: tasks/threads, memory objects and IPC objects. This provides for a unified accounting for operating system resources. In particular it is relevant to extending μkernel abstractions to distributed environments, namely, network IPC and DSM. Accounting only on three elements (processing, VM and IPC) simplifies our information management, while providing a more accurate insight into the system load.

In order to demonstrate the potential benefits of the additional information, we conducted some preliminary experiments with and without considering network IPC and XMM.

Figures 3 and 4 present the average execution time of 12 experimental tasks (ALT) started on one node every 500 milliseconds and distributed on 3 nodes. Average execution time is presented as a function of processing and communication the tasks perform. The amount of communication is expressed as the number of network messages/pageins that each task communicates with a server on a randomly chosen node. Messages are sent in pairs (16 bytes and 256 bytes), and pages are 4KB size. The amount of processing is expressed as a function of the cpu-time the tasks consume (user-time). The figures demonstrate the advantage of considering the additional information. The upper surfaces represent LD without considering additional information while the lower one does. There is an obvious advantage of considering it. The benefit depends on the amount of communication that an application performs. For example, improvement can be over 100% if there is a significant amount of communication with the remote server. There are two more interesting details. Peaks in the surfaces demonstrate cases when there is a small amount of processing while IPC/XMM activity is stressed. NORMA IPC becomes a bottleneck in such cases. The

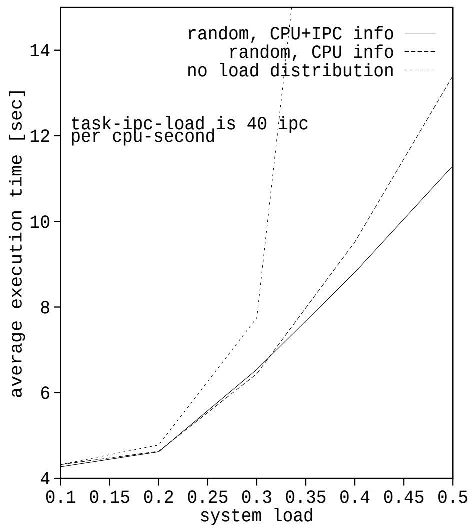

Figure 7. Average Task Execution Time for Random Strategy v. System Load.

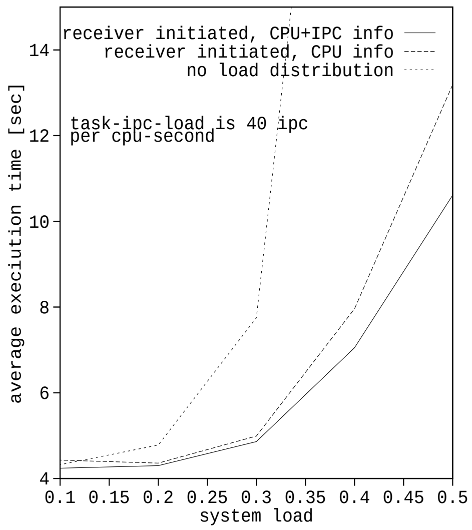

Figure 8. Average Execution Time for Receiver Initiated Strategy v. System Load.

surfaces intersect for the zero IPC/XMM activity, demonstrating that our scheme incurs no significant overhead in considering additional information.

It should be pointed out that our goal is not to migrate each client towards the server. An obvious example is a file server, where we certainly would not allow migration of each client to the file server. In such cases the LD server would not be started on the file server node, preventing any task to be migrated towards it. We are rather targeting new distributed applications, with a variety of clients and servers. Under some circumstances we shall decide to migrate a task. If it also happens that the server node is a convenient destination for migration, than we can take advantage of our knowledge about DSM and network IPC information.

This experiment demonstrates that in some cases LD decisions could be improved by using information on network IPC and XMM. Performance improvements are gained in two ways. The obvious advantage comes from the fact that moving a client towards the server improves the average task execution time. The less obvious advantage is the consequence of the limited IPC bandwidth which could, similar to processor, become a bottleneck. Distributing load from the client node to other nodes improves performance because communication is parallelized, although it still ends up with the same server node, but now in parallel with few client nodes.

There is no intent to introduce any complex or perfect load information management. We propose only simple and straightforward extensions of existing schemes. Out of the task candidates for migration, and the host candidates for destination, we select those that also satisfy the communication criteria. Necessary information is obtained either locally (for task selection) or during negotiation (for node selection). Neither poses significant overhead. Accounting information for large number of machines only influences the amount of information that is reasonable to store within the particular structures. Therefore, either the limited number of nodes can be considered (on the first-come-first-served policy), or the space for storing information can be added dynamically.

In order to inspect how our LD scheme performs for various strategies, we repeated the experiments with a few well know strategies, as presented in Figures 5, 6, 7, and 8. The figures show average execution time of ALTs as a function of load for the symmetrical, sender initiated, random and receiver initiated strategy. The parameters are the same as described in Section 3.4, except for workload which varies from 0.1 to 1.5 on one node and it is 0 on two other nodes. The performance improvement is observed for all strategies.

5 Conclusion

Our research consists of the design and implementation of load distribution on top of the Mach μkernel. It is achieved in the following three phases: task migration, instrumenting Mach to provide information on network IPC and XMM, and distributed scheduling. The conducted research allowed us to reach some experiences, as presented in Section 4. Once more we would like to point out the benefits of using additional load information, and the comparison between task migration and initial placement.

We demonstrated that considering additional information on network IPC and XMM for distributed scheduling can improve the average execution time of the tasks. We demonstrated improvements for various scheduling strategies. We also revisited the usefulness of task migration for LD compared to initial placement.

We showed instances when the information provided by running task leads to better scheduling decisions and decreases average execution time. Our research is not intended for the underutilized workstation environments, but rather for new MPP and cluster architectures and new distributed applications. We believe that such a new environment should make use of all possible mechanisms for load distribution.

We foresee the following work as a prospect for our future research:

- Providing at least a primitive form of process migration and experimenting with its relationship to a new distributed file system.
- Working with real applications would most probably be performed with the PVM port on Mach and choosing suitable PVM applications.
- Repeating the conducted experiments in a larger installation with more computers than the current environment.

Availability

All referenced programs or utilities are available upon request. We also hope to provide our in-kernel task migration to OSF or CMU in order to merge it into the source code tree.

Acknowledgements

We would like to thank CMU and OSF for providing us with Mach, as well as for the continuous support. The OSF/1 source code license is provided to us as a part of our collaboration with the Grenoble OSF Research Institute. The following is an alphabetically ordered list of the paper reviewers, whose significant help is appreciated. David Black, Henry Chang, Orly Kremien, Reinhard Lüling, Laurent Philippe, Bryan Rosenburg, Nikola

Šerbedžija, Brent Welch and Roman Zajcew. Particularly we are indebted to Alan Langerman for his profound review and sharp criticism. Our non-native English has been ironed by Simon Patience. Special thanks are to program committee for the extensive comments which improved our article in many aspects. Prof. Nehmer provided support and made the whole project live.

References

[Bara85] Barak, A. and Shiloh, A. (September 1985) *A Distributed Load-Balancing Policy for a Multicomputer.* Software-Practice and Experience, 15:901–913.

[Barr91] Barrera, J. (November 1991) *A Fast Mach Network IPC Implementation.* Proceedings of the Second USENIX Mach Symposium, pages 1–12.

[Bart93] Barton-Davis, P., McNamee, D., Vaswani, R., and Lazowska, E. (April 1993) *Adding Scheduler Activations to Mach 3.0.* Proceedings of the third USENIX Mach Symposium, pages 119–136.

[Blac92] Black, D., Golub, D., Julin, D., Rashid, R., Draves, R., Dean, R., Forin, A., Barrera, J., Tokuda, H., Malan, G., and Bohman, D. (April 1992) *Microkernel Operating System Architecture and Mach.* Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pages 11–30.

[Boyk93] Boykin, J., Kirschen, D., Langerman, A., and LoVerso, S. (1993) *Programming under Mach.* Addison-Wesley.

[Cabr86] Cabrera, L. (June 1986) *The Influence of Workload on Load Balancing Strategies.* Proceedings of the Winter USENIX Conference, pages 446–458.

[Chen90] Chen, R. (October 1990) *Building a Fault-Tolerant System Based on Mach.* Proceedings of the First USENIX Mach Workshop, pages 157–168.

[Cher90] Cheriton, D. (June 1990) *Binary Emulation of UNIX Using the V Kernel.* Summer USENIX Conference, pages 73–86.

[Doug91] Douglis, F. and Ousterhout, J. (August 1991) *Transparent Process Migration: Design Alternatives and the Sprite Implementation.* Software-Practice and Experience, 21:757–785.

[Eage86] Eager, D., Lazowska, E., and Zahorjan, J. (May 1986) *Dynamic Load Sharing in Homogeneous Distributed Systems.* IEEE Transactions on Software Engineering, 12:662–675.

[Eage88] Eager, D., Lazowska, E., and Zahorjan, J. (May 1988) *The Limited Performance Benefits of Migrating Active Processes for Load Sharing.* Performance Evaluation, 6:63–72.

[Fori89] Forin, A., Barrera, J., and Sanzi, R. (January 1989) *The Shared Memory Server.* Proceedings of the Winter USENIX Conference, pages 229–243.

[Golu90] Golub, D., Dean, R., Forin, A., and Rashid, R. (June 1990) *UNIX as an Application Program.* Proceedings of the Summer USENIX Conference, pages 87–95.

[Juli89] Julin, D. (September 1989) *Network Server Design.* Technical Report, Mach Networking Group, Carnegie Mellon University.

[Krem92] Kremien, O. and Kramer, J. (November 1992) *Methodical Analysis of Adaptive Load Sharing Algorithms.* IEEE Transactions on Parallel and Distributed Systems, 3:747–760.

[Krue88] Krueger, P. E. (June 1988) *Distributed Scheduling for a Changing Environment.* Technical Report TR-780, PhD Thesis, CS Department, University of Wisconsin-Madison.

[Krue91] KRUEGER, P. and CHAWLA, R. (June 1991) *The Stealth Distributed Scheduler.* Proceedings of the 11th International Conference on Distributed Computing Systems, pages 336–343.

[Kupf93] KUPFER, M. (April 1993) *Sprite on Mach.* Proceedings of the third USENIX Mach Symposium, pages 307–322.

[Litz92] LITZKOW, M. and SOLOMON, M. (January 1992) *Supporting Checkpointing and Process Migration outside the UNIX Kernel.* Proceedings of the USENIX Winter Conference, pages 283–290.

[Milo93] MILOJICIC, D., ZINT, W., DANGEL, A., and GIESE, P. (April 1993) *Task Migration on the top of the Mach Microkernel.* Proceedings of the third USENIX Mach Symposium, pages 273–290.

[Milo93a] MILOJICIC, D., GIESE, P., and ZINT, W. (September 1993) *Load Distribution on Microkernels.* Proceedings of the Fourth IEEE Workshop on Future Trends in Distributed Computing.

[Orma93] ORMAN, H., MENZE, E., O'MALEY, S., and PETERSON, L. (April 1993) *A Fast and General Implementation of Mach IPC in a Network.* Proceedings of the third USENIX Mach Symposium, pages 75–88.

[Pati93] PATIENCE, S. (April 1993) *Redirecting System Calls in Mach 3.0: An Alternative to the Emulator.* Proceedings of the third USENIX Mach Symposium, pages 57–74.

[Phel93] PHELAN, J. M. and ARENDT, J. W. (April 1993) *An OS/2 Personality on Mach.* Proceedings of the third USENIX Mach Symposium, pages 191–202.

[Phil93] PHILIPPE, L. (1993) *Contribution à l'étude et la réalisation d'un système d'exploitation à image unique pour multicalculateur.* Technical Report 308, Phd Thesis, Université de Franche-comté.

[Roga93] ROGADO, J. (June 1993) *A Prototype File System for the Cluster OS.* Presented at Grenoble OSF RI Workshop on Microkernel Technology for Distributed Systems.

[Rozi92] ROZIER, M. (April 1992) *Chorus (Overview of the Chorus Distributed Operating System).* USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pages 39–70.

[Shiv92] SHIVARATRI, N., KRUEGER, P., and SINGHAL, M. (December 1992) *Load Distributing for Locally Distributed Systems.* IEEE Computer, pages 33–44.

[Stum88] STUMM, M. (1988) *The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster.* Proceedings of the Second Conference on Computer Workstations, pages 12–22.

[Sund90] SUNDERAM, V. S. (December 1990) *PVM: A Framework for Parallel Distributed Computing.* Concurrency: Practice and Experience, pages 315–339.

[Thei85] THEIMER, M., LANTZ, K., and CHERITON, D. (December 1985) *Preemptable Remote Execution Facilities for the V System.* Proceedings of the 10th ACM Symposium on OS Principles, pages 2–12.

[Toku90] TOKUDA, H., ET AL. (October 1990) *Real-Time Mach: Towards a Predictable Real-Time System.* Proceedings of the First USENIX Mach Workshop, pages 73–82.

[Welc90] WELCH, B. (April 1990) *Naming, State Management and User-Level Extensions in the Sprite Distributed File System.* Technical Report UCB/CSD 90/567, PhD Thesis, CSD (EECS), University of California, Berkeley.

[Welc93] WELCH, B. Personal communication 1993.

[Zajc93] ZAJCEW, R., ET AL. (January 1993) *An OSF/1 UNIX for Massively Parallel Multicomputers.* Proceedings of the Winter USENIX Conference, pages 449–468.

[Zaya87] ZAYAS, E. (November 1987) *Attacking the Process Migration Bottleneck.* Proceedings of the 11th Symposium on Operating Systems Principles, pages 13–24.

[Zhou88] ZHOU, S. and FERRARI, D. (September 1988) *A Trace-Driven Simulation Study of Dynamic Load Balancing.* IEEE Transactions on Software Engineering, 14:1327–1341.