Task Migration on the top of the Mach Microkernel*

Dejan S. Milojičić†, Wolfgang Zint,
Andreas Dangel and Peter Giese

University of Kaiserslautern, Informatik, Geb. 36, Zim. 436
Erwin-Schrödingerstraße, 6750 Kaiserslautern, Germany
tel (49 631) 205 3293 fax (49 631) 205 3558
e-mail: [dejan,zint,dangel,giese]@informatik.uni-kl.de
Paper presented at the third USENIX Mach Symposium, Santa Fe, April 1993.

Abstract

This paper presents initial results in the design and implementation of task migration on the top of the Mach μkernel. The presented work is part of a broader project concerning research on load distribution. Our task migration is implemented in user space in order to improve portability, maintainability and flexibility. At the same time we paid attention not to sacrifice performance, transparency and functionality. Although we have implemented task migration in user space, some modifications to the kernel were necessary. We have designed on the μkernel abstraction level, unaware of issues such as files, signals or other UNIXisms which were main obstacles to simple and transparent process migration so far. We expect to benefit much more when we start dealing with load distribution decisions. Our design allows us to make scheduling decisions entirely based on Mach virtual memory, interprocess communication and processor load.

1 Introduction

The field of Load Distribution (LD), encompassing load balancing, load sharing and related mechanisms, such as Task Migration (TM) and remote execution, has been explored for quite some time [Bokh79, Eage86, Milo91]. Many practical and theoretical results have evolved, but none has achieved wide acceptance. There have been many obstacles, such as a relatively old operating system design, artificial support for distributed computing by extending stand-alone operating systems in a network, the lack of distributed applications, dependence on the particular modified version of the underlying operating system, limited functionality and transparency etc. We believe that contemporary μkernels provide a convenient base for LD and particularly for TM. In a μkernel, basic abstractions are supported within the kernel, while other functionality is provided within user space. This allows access to data and functionality formerly hidden inside the kernel, and thereby improves opportunities for a user level TM implementation. This and other characteristics, such as network transparency, modern virtual memory design and modularity, that modern μkernels possess (particularly Mach [Blac92]), are a promising base for yet another effort in the field of TM. Therefore, we have set as our goal to demonstrate that the Mach μkernel is a suitable environment for the implementation of TM. Our implementation aims at a transparent and portable migration in user space (possibly with minor modifications to the kernel), without paying significant penalties in performance and functionality.

The paper describes two user-space TM servers, necessary modifications to the kernel and a preliminary performance evaluation. Besides serving as a base for future work, the goal of this phase was

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

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

to provide insight into the complexities of TM implementation, level of transparency, functionality and performance penalties paid for the user level implementation.

The rest of the paper is organized in the following manner. In this section we present an overview of the field and the relevant Mach characteristics. The design of our task migration scheme is presented in section 2. The implementation is described in section 3 by surveying the modifications applied to Mach, and two versions of the TM server. In section 4, we present some preliminary performance measurements. Related research is described in section 5. Our conclusions and future research are presented in section 6.

1.1 Overview of the Field

The field of TM has often been surveyed and described, e.g. in [Gosc91, Arts89]. Here, we briefly mention the motivations for TM, its issues and important implementations.

Computing history has shown that technology continuously provides more resources, but also increases demand for these resources. No matter how much is provided, there will always be an application that needs more, be it computing power, primary memory, huge paging space or specialized hardware. One of the convenient ways to satisfy this need is TM. Other applications of TM include the areas of fault-tolerance, real-time, system administration etc. Mobile computers may become yet another important area to apply TM [Doug92].

One of the intriguing issues of TM is **transparency**. It implies that a task can transparently execute after its migration, without recognizing that it has been migrated, except possibly for the performance difference. **User v. kernel space implementation** concerns the tradeoff between efficiency and simplicity. A user space implementation, while sacrificing speed to some extent, can provide for a simple, robust solution [Arts89]. A kernel implementation may be necessary in the case of real-time requirements. **Residual dependency** is related to the amount of the task state that may be left at the originating or some other node. It is directly related to the reliability of task migration. Task migration should be recoverable, i.e., it should be possible to recreate a task in the case that its migration has failed. **Autonomy and security** are important with respect to the environment as well as for the migrated task itself. A task should in no way influence the integrity of the environment that it is migrated to. Reclaiming of resources, if requested, should happen in a limited time. Finally, **performance** consists of immediate transfer and freeze time (the time during task is suspended), as well as the transfer time of lazy migrated state, such as applied for "Copy-On-Reference" (COR).

The history of TM is relatively old. It runs from early multiprocessors and distributed systems [Bokh79, Mill81], to the modern distributed environments and massively-parallel-processors [Zajc93]. It is an area where implementation dominates. Many prototypes have been running, but only a few have emerged as working products, e.g. Locus [Walk89]. We have chosen to mention the following work. **MOS(IX)** [Bara85] was one of the first to demonstrate μkernel appropriateness for LD, splitting the operating system into two layers. **Accent** [Zaya87], the predecessor of Mach, was the first to introduce the COR technique. The COR technique allows for faster address space transfer and achieves performance improvement by lazy copy of pages, thereby avoiding unnecessary page transfers. The **V kernel** introduced a technique known as "precopying" of the task address space [Thei85]. Precopying significantly improves freeze time, the period while the task is suspended during migration. **Charlotte's** TM scheme [Arts89] extensively relies on the underlying operating system, its communication mechanisms and language. Interprocess communication has been modified in order to support transparent network communication. **Locus** [Walk89] is an exception to most TM implementations because it was the only one to achieve a product stage. It has been ported to the AIX operating system on the IBM 370 and PS/2 computers under the name of Transparent Computing Facility. Locus has achieved a high level of functionality and transparency, paying the cost in significant kernel modifications. It has been recently ported to OSF/1 operating system, running on top of Mach [Zajc93]. **Sprite** [Doug91], is one of the recent and successful process migration implementations. It will be covered in more detail in section 5.

1.2 Relevant Mach Characteristics

The purpose of this subsection is not to give a Mach description, but rather to mention particular Mach features relevant to our TM scheme. More details about Mach can be obtained in [Blac92, Golu90]. The following issues are relevant to our scheme:

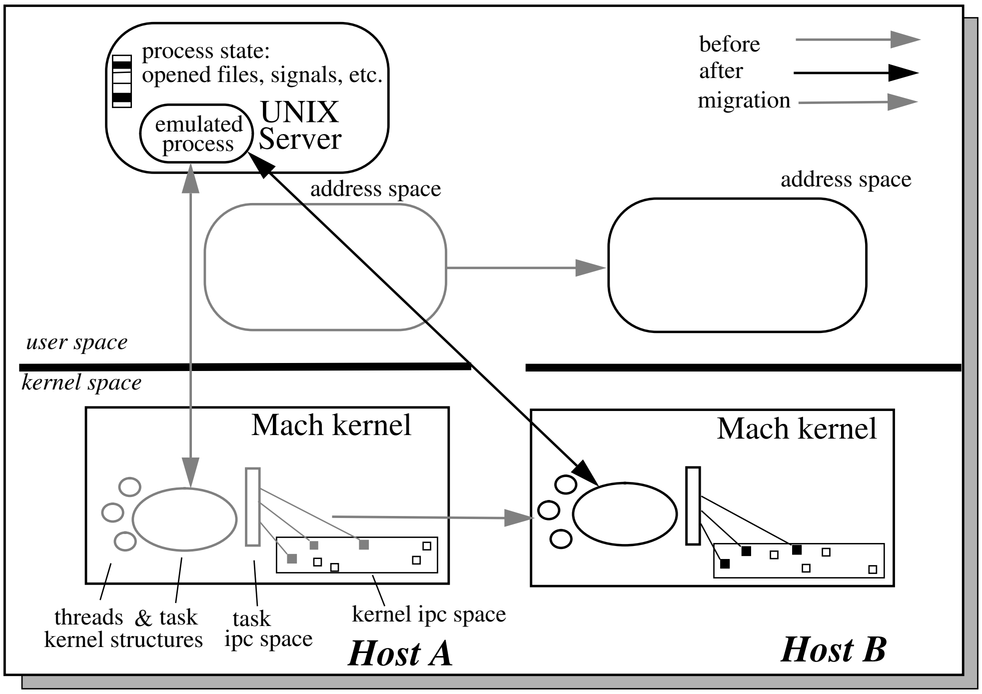

Figure 1: *The transferred task state consists of the address space, IPC space, thread states, and other state, such as task and thread kernel ports, exception ports, task bootstrap port etc. The UNIX server state, contained within the task, is transferred implicitly, as a part of the address space (shared pages) and IPC space (rights through which the emulator communicates with the UNIX server).*

- **Multicomputer support,** as provided by the Mach NORMA version, made our task much easier. The support for network Interprocess Communication (IPC) [Barr91], and Distributed Shared Memory (DSM) [Barr93] is of particular significance. Network IPC has been implemented within the kernel in order to improve performance. It transparently extends local IPC over the network. DSM overcomes current pagers' limitations to support only a single kernel. DSM code is transparently inserted between kernel and pager code, allowing existing pagers to support multiple kernels.
- **UNIX emulation** [Golu90] relieved us from handling UNIX abstractions in early design phases. However, this does not preclude us from considering modifications later, in the case of low performance. We have particularly observed the unacceptable performance of the emulator and the current implementation of mapped files, which have been optimized for parallel, but not for distributed, systems.
- **Device access,** based on IPC, provided a further level of transparency [Fori91]. Formerly, it used to be very hard, if not impossible, to extract the kernel state that remained in device drivers [Free91].

The Mach μkernel has the potential to become a standard basis for building operating systems. There are a lot of projects that have Mach as an underlying environment. Finally, its predecessor Accent was the base for TM experiments [Zaya87]. Mach inherited its design from Accent, particularly in some aspects of memory management. Therefore, Mach is a promising environment for TM experiments.

2 Task Migration Design

TM is implemented on top of Mach, but outside of any operating system emulation server. This implies that we migrate Mach tasks, while the UNIX process remains on the source machine, as presented in Figure 1. The obvious disadvantage is home dependency, a residual dependency with regard to the originating machine [Doug91]. The advantage is that the same TM scheme may be used for the applications running on any operating system emulated on the top of Mach (e.g. BSD UNIX [Golu90], VMS [Wiec92] etc.), as well as for the applications running on the bare Mach. As in most TM implementations, our scheme consists of extracting the task state, transferring it across the network and reestablishing it at a new site. The world notices no difference while communicating with the migrated task, except for performance. The Mach task state consists of the address space, the capability space, the thread states and some other state. We are taking advantage of the Mach NORMA version which supports in-kernel network IPC and DSM. We do not implement capability migration and message forwarding; NORMA code takes care of that, as well as of DSM. However, not all of the task state can be transferred from within user space. We have faced the following problems in the design of TM.

- Mach objects are represented by kernel ports. Actions on behalf of the objects are performed by sending messages to their kernel ports. Therefore, it is necessary to preserve the task and thread kernel ports after migration. However, the kernel ports are not handled the same way as normal ports whose receive capabilities reside within the task IPC space. There are no receive capabilities for the kernel ports, and therefore there is no way to extract them using the existing Mach interface.
- Any shared or externally mapped memory causes a consistency problem for TM. Common examples are memory that is shared through inheritance and memory mapped files. Such memory areas are non-eligible for user space transfer with the current pagers, such as the default pager and the inode pager, which are designed to see only one kernel. The consistency problem could be overcome if Mach pagers would support multiple kernels. Basically, the same consistency support should be provided as in NORMA DSM. Internal, non-shared memory can simply be read from the source side and written into the task instance on the destination side. We call such areas "eligible" for user space transfer.

These were the only two cases where we did not have kernel support. As a solution to the first problem we provide two system calls to interpose the task and thread kernel ports. Both calls return the kernel port and leave the supplied port as a new kernel port. After that, we are the only ones who have the capability for the task kernel port and, therefore, only we can control it. All messages sent to a task end up at the extracted kernel port, whose receive capability is now in the TM server's IPC space. In the meantime, we perform actions on behalf of the task and transfer its state to the destination side. Once the state is transferred, we make the interpose call on the destination side, inserting the original task kernel port into the new task instance.

Shared memory and memory mapped files are a complex problem. However, this is not inherent in TM. Shared memory is not supported by most TM implementations, and files are handled by a distributed file system, e.g., in Sprite [Doug91], MOS(IX) [Bara85] and Locus [Walk89]. We adopt a similar approach. Shared memory and mapped files are supported by NORMA distributed shared memory. We currently handle it by exporting the pager port that represents a region of memory and then remapping it on the remote side. As soon as Mach exports the needed functionality, we may switch to it.

Except for the aforementioned two cases, everything else is implemented in user space. Our TM algorithm can be described with the following steps:

- suspend the task and abort the threads in order to clear the kernel state,
- interpose the task/thread kernel ports,
- transfer the address space, using various user and NORMA memory transfer strategies,
- transfer the threads by getting the state and setting it on the remote side,
- transfer the capabilities (we extract, transport and insert capabilities; NORMA does the actual port transfer),
- transfer the other task/thread state,
- interpose back the task/thread kernel ports at the destination site and
- resume the task.

Before migration, it is necessary to clear the thread kernel state, since it is almost impossible to transfer it. Typically, this case arises when threads are waiting within the kernel, e.g. for a message to be sent or received or a faulted page to be paged-in. Mach provides a particular call *thread_abort*, which aborts thread execution within the kernel and leaves the thread at a clear point, containing no kernel state inconvenient for migration. The execution is restarted after migration, unless instructed otherwise. Lux et al. describe how kernel state complicates migration in BirliX [Lux92].

The adopted design relieves us from issues, such as signals and distributed file systems, which have traditionally been a nightmare in TM and remote execution implementations. We expect major benefits later, when we are to make LD decisions. We shall make decisions only in terms of Virtual Memory (VM and XMM), IPC and processor load, while previous systems separately dealt with files, network traffic, local and remote IPC. Of particular interest for us are XMM and network IPC which, as we presume, dominate the costs.

3 Implementation

The underlying environment in our TM research consists of 3 PCs, interconnected via Ethernet. The PCs are based on a 33 MHz i80486 processor with 8 MB RAM and 400MB SCSI disk. We have been using the Mach NORMA version (MK7, MK12, MK13) and the UNIX server UX28. TM has been accomplished by designing and implementing two versions of migration server, a simple and an optimized one. They have been implemented in a relatively short period. We installed our first PC at the end of November 1991, and the current environment at the end of March 1992. At the end of May we migrated a task for the first time. Polishing it up took more time, which is partially due to the parallel NORMA development. The task migration is currently stable for test applications.

The routines concerning the kernel modifications fit in a file of 300 lines of C code, most of which are comments, debugging code and assertions. The simple migration server has about 600 lines of code, and is actually a library routine. It could be used either by linking it to the task that is going to initiate migration, or by providing a server interface, in which case the migration code is linked to the server. In the former case it is not possible for a task to migrate itself, since one of the first actions on behalf of the migrating task is to suspend it, causing a deadlock. In such a case, it is necessary to provide a server which will indirectly start the migration. The optimized migration server has about 13000 lines. It is based on cooperating servers running on all nodes and has a few threads of control.

In the following subsections we describe the kernel modifications and the simple and optimized TM server.

3.1 Necessary Modifications to the Mach Microkernel

As mentioned in the previous section, the kernel modifications consist of the kernel port interpose, which is a necessary, permanent modification, and a temporary extension to export the memory object port until the NORMA code exports the needed functionality.

The kernel port interpose modification introduced two new system calls, one for the task and one for the thread kernel port interposition. Like most other calls, these are actually messages, sent to the host on which the task executes. Unfortunately, it can not be sent to the task or thread. The interpose routine takes care that all messages accumulated in the original kernel port (while being interposed) are handled properly and the threads blocked in the full port message queue are woken up and their requests handled. In order to avoid unnecessary blocking, the queue limit for the interposed kernel ports is increased to its maximum. Once the ports are interposed back on the destination side, the default queue limit is restored. Interposing, as presented in Figure 2, consists of exchanging the pointer to the IPC space the port belongs to, as well as modifying some other internal task state. On the return from the call, the receive capability for the task kernel port resides within the server IPC space and the receive capability for the interposing port is extracted from the server IPC space into the kernel IPC space.

Exporting the pager port is a temporary modification. The existing NORMA version does not support distributed memory shared through inheritance, although DSM functionality exists. Therefore, we adopted a temporary solution. We export the memory object for the particular region. For security reasons, the Mach interface does not export the memory object port capability, but only the related

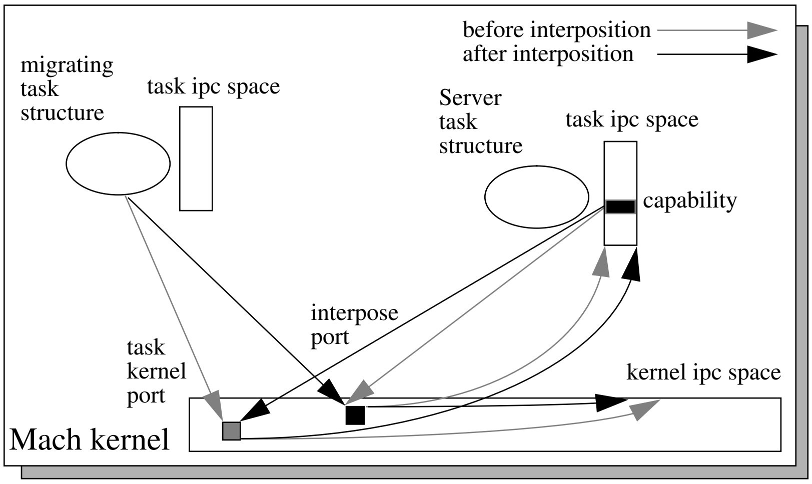

Figure 2: *Interposing the task kernel port - internal perspective: the task_interpose routine exchanges the port that represents the task with an interposing port; it prerequisites exchanging pointers to ports and the IPC spaces the ports belong to, namely kernel IPC space with the IPC space of the task that initiates migration.*

name capability [Youn89]. We extract the capability for the memory object port and map it onto the remote side. This provided the necessary functionality in the early design phases. The added system call looks up the memory object port which represents the memory region and exports a send right out of the kernel.

3.2 Simple Migration Server

At the very beginning the Simple Migration Server (SMS) was implemented in order to verify design feasibility and to get the initial insight into performance. Later on, we completely switched to it. It relies entirely on the NORMA memory transfer and is deliberately unoptimized. From the memory transfer point of view, the SMS has better performance than the Optimized Migration Server (OMS), which relies on user level address space transfer and is more flexible regarding the choice of the memory transfer strategy. Regarding capabilities, threads and other state transfer, the SMS has worse performance than the OMS, since each capability, thread state, etc., is transferred separately, resulting in a new message sent across the network. The SMS consists of a few parts that migrate capabilities, memory, thread state and the other task state. Before migration, the task is suspended and its kernel port is interposed. From now on, we are sure that no unprivileged user has access to the task. A privileged user can always obtain a send capability for the current task kernel port, regardless of the fact that we have exchanged it.

The **thread migration** consists of transferring the thread state. There are a few different flavors of the thread state which should be transferred, such as the contents of CPU and FPU registers etc. The thread states are extracted on the source side and inserted into the newly created threads on the destination side. We have never had an interest in any particular state *per se*, we just copied all of them. This improves the portability of our TM scheme across various processor architectures.

The **memory migration** is based on the default NORMA COR strategy, as opposed to the optimized server where different strategies are implemented in user space. We wanted to explore different approaches in order to get more insight into performance and functionality tradeoffs. In both approaches, the task address space is analyzed, and areas not eligible for user-space migration are transferred using NORMA, as discussed in section 2. These areas are mapped in the destination task, using NORMA DSM. The rest of the memory is either copied in user space (OMS) or using NORMA support (SMS). In either case, non-initialized areas are simply re-allocated.

The **capability migration** is performed by extracting the capabilities on the source node and inserting them on the destination node. We migrate send, send-once and receive capabilities. Port sets are migrated by extracting all receive capabilities from the port set, migrating them and reconstructing the port set on the remote side. Capabilities are migrated one by one in SMS. This has detrimental effects on performance, due to the high costs involved in capability migration. Therefore, we may switch to the same method as in the OMS: extract all capabilities, transfer them in a message to the destination node, and insert them. This requires servers on both sides.

The **other state** consists of suspend counts for task and threads, bootstrap ports, exception ports etc. Each of these states is appropriately extracted and later inserted into the migrated instance of the task. Some of the state is transferred as part of creating the remote instance of the migrated task.

3.3 Optimized Migration Server

The Optimized Migration Server (OMS) has functionality similar to SMS, except that it supports user-space memory transfer and various optimizations. Only memory eligible for user-space migration is transferred across the network by the OMS, while shared and externally mapped memory is still transferred by NORMA. The OMS treats error recovery in the case of an erroneous migration, and potentially retries migration on the same or a different destination node. The following data transfer strategies are supported:

- **Flushing residual dependency** causes the transfer of all memory that remained either on the source or on other nodes due to the previous migrations.
- **Precopy** is implemented similarly to the V kernel; however, its performance rendered it useless: it was too slow to switch from the default pager to a new pager. The basic idea is to remap the address space to the new pager and then to extract the information about modified pages. We are investigating improvements.
- **Copy on reference** is supported in the classical way, similar to Accent or NORMA, but in user space. There is also an extended version that transfers a few pages surrounding the faulting page.
- **Read ahead** is meant to transfer the whole task address space within a given time. It periodically transfers some of the remaining pages to the destination node, until all pages are transferred.
- **Direct copy** is a simple transfer of all memory from the source node to the destination node.

The applied optimizations consist of **packing all capabilities and other relevant information in messages**, requiring servers on both machines; **overlapping of various state transfer**, e.g. memory with capability and thread transfer; **truncating** zero ended memory areas and **substitution of emulator and emulation vectors** by local instances. OMS uses an External Memory Manager (EMM) to provide the functionality for user-level memory transfer strategies. The address space of the migrating task is reconstructed on the destination side by remapping memory objects. After that, the EMM receives paging requests, and either transports referenced pages over the network or retrieves them from the local paging-file. The basic COR-mechanism is modified for COR-extend to fetch some remote pages surrounding the faulting page and transport the requested pages in parallel. Extra pages are saved locally to resolve succeeding page faults. Flushing the entire address space to backing store, as done in Sprite, doesn't make much sense in our environment with local paging files. Hence OMS uses a variant of the strategy. It transfers all eligible pages in parallel with the thread state and capability migration. Network and paging file access are serviced by separate threads to increase performance. The paging file is accessed by a stand-alone file system (the same as the default memory manager uses) to avoid deadlock with the BSD server.

OMS has been useful during development, however, despite many opportunities, currently we have abandoned the use of OMS due to the following reasons:

- Task memory regions not eligible for user-space transfer, such as shared memory or memory backed up by specific pagers, such as the inode pager, still need support of XMM code or the support of new distributed pagers. We have considered the latter case, but we found that it has not been preferred by other main Mach developers. Choosing this way would mean a departure from the mainstream development. In the former case, we duplicate the functionality.

- We have found SMS more robust for further research on LD. Its simplicity, despite somewhat slower performance, e.g. for port transfer, allows us to trace the problems more easily. OMS requires few threads of control, a separate pager, etc., making debugging harder.
- Our main goal was not TM itself, but rather LD, where TM is one of its important mechanisms. Migration candidates are usually long lived tasks (range of tenths of seconds); therefore the performance difference of a few hundred milliseconds between SMS and OMS is of secondary importance.

If we later, for any reason, find SMS inappropriate, we can always re-investigate the choice of migration techniques once again. One of the most likely reasons for reviving the OMS would be the comparison of various migration strategies. This would become an issue once we have a working LD scheme and other Mach applications, such as a distributed file system. As a short term improvement, we may merge some of the good characteristics of OMS, such as packing rights and other state, in a few messages, avoiding many short messages over the network.

4 Performance Measurements

Performance measurements are an important part of any TM implementation. Unfortunately it is hard to find adequate applications for measurements. There are a few true distributed applications, particularly for the Mach μkernel. Most researchers in the field of LD have made artificial loads or used some benchmarks; they rarely used real applications. We used the following: Artificial Load Task (ALT), WPI benchmark suite, parallel make support; matrix multiplication, simulation programs, and some other applications.

All results presented should be accepted with caution. There are many influences on its accuracy, since TM and the Mach NORMA version are continuously being changed. We present performance measurements for the sake of completeness and to give more insight into the order of magnitude. All results presented in Figures 3 to 10 are obtained as a mean of the five consecutive measurements. Only the results for Figures 11 and 12 are the mean of two out of the five measurements, since the experiment failed for some input values due to a known bug. If not otherwise indicated, SMS has been used. All measurements have been performed on norma13[1] and [UX,emulator]28.

4.1 Migration Server Measurements

In this subsection we present some low-level measurement results for both migration servers. In order to gain insight into the influence of particular migration parameters on the overall performance, we designed an Artificial Load Task (ALT). We can tune the following ALT parameters: ratios between computation, IPC activity and memory access, amount of memory (internally mapped by the default memory manager, locally and remotely shared memory), the number of threads, and the number of capabilities (receive, send for local and remote ports). ALT loops for a given number of iterations and in each loop it performs a number of remote procedure calls to the local and remote server, accesses locally and remotely mapped shared memory, followed by a computation part, currently represented by the Linpack benchmark. The main idea behind ALT is to represent the task VM, IPC and processor load behavior. We plan to experiment with various applications and thereby obtain realistic values for ALT parameters and use it as an artificial load for distributed scheduling experiments.

TM consists of three phases which transfer virtual memory, threads and capabilities. Each of these phases was measured using ALT. In Figure 3, we present measurements of transfer time v. memory size, performed using SMS, i.e. using NORMA default COR strategy. Transfer time as a function of the thread number is presented in Figure 4. Figure 5 shows transfer time as a function of the number

[1] In the meanwhile we have also ported SMS to norma14, however, we shall still present results only for norma13 version for a few reasons. First, although norma14 has been more stable for many previous problems, we did observe some new ones, preventing stable measurements. Second, we have switched within SMS to a new kernel supported address space transfer. This could favor SMS over OMS, which hasn't been ported to norma14. Finally, except for XMM, we have not observed any significant difference in relative performance behavior between the two versions. Despite some changes, the curves presented in Appendix A remained similar. We observed faster address space transfer (above 30%), but slower lazy copying.

of receive capabilities. Comparison of receive and send capabilities transfer is given in Figure 6. In Figure 7, we present an interesting side-effect of extending a VM layer to a distributed environment. While experimenting with memory transfers, we noticed unacceptable values for a particular test task. The test task address space regions are created by its parent using a *vm_write* system call on a page-by-page basis. As we closely inspected the address space, we saw that it consisted of many entries, which could be, but have not been, compacted into one. In local cases this does not represent a bottleneck, however, once that task address space is being transferred across the network, entry transfer cost is significantly increased. As we modified the program slightly to write all the memory in a big chunk instead of page-by-page, thereby reducing the number of entries, we significantly improved transfer time. This anomaly could be easily overcome by using the existing kernel function *vm_map_simplify*. From aforementioned measurements we can conclude that migration transfer time is independent of task address space size. It is a linear function of the number of internal memory regions, but they are usually limited in number, except for the unoptimized cases, such as the one presented in Figure 7. The number of threads and rights are usually small for tasks representing UNIX process, except for the servers, which are not good candidates for migrations, anyway. Except for address space transfer which is characterized by fluctuations, all curves are linear. Address space transfer has been subject to change with various norma versions, therefore we didn't further inspect the reasons for its fluctuation.

The presented measurements characterize the transfer phase. Equally important is performance during the task execution after migration, which depends on the transfer of residual state. Figure 8 presents an interesting and unexpected behavior of COR for OMS and SMS. While we expected better capability transfer performance for OMS, we did not expect better address space transfer. According to the measurements, it seems like OMS also has better address space transfer. The measurements are performed with an ALT version in which only the first integer of the page is written. Since OMS introduces an optimization of transferring the data up to the last non-null character in an area, just one integer is transferred instead of the page. As soon as the last integer within the page is written, performance of OMS decreased below the SMS performance. Performance of SMS is the same in both cases, since it has page granularity. This simple experiment demonstrates the benefits of implementing user-space TM, since it is rather easy to substitute various strategies and parameters. In Figure 9, we demonstrate the expected difference in transferring receive capabilities for SMS and OMS. The performance for send capabilities is similar and is not presented. Finally, the obvious benefit of migration towards a server with which the task communicates is demonstrated in Figure 10, where we present the task execution time as a function of the number of remote procedure calls with a server to/from which we migrate the task.

Based on above results we can conclude that the performance of our TM implementation is comparable to other kernel supported implementations, while it significantly outperforms user space implementations. For example, reported process migration in Sprite [Doug91] is few hundred milliseconds for a standard process. We have measured similar performance for our TM implementation[2]. In Condor [Litz92], it takes two minutes to migrate a 6MB process. In our servers it wouldn't cost much more to migrate any other average size task (assuming that memory regions are not chopped in many entries, which is not likely to be). Of course, the actual transfer is done lazy, preventing direct comparison. Similar results of user level implementations have been reported by other implementors, e.g. [Alon88]. It should be noted, though, that we have not aimed at optimizing performance in this phase of the project. For completeness, we shall compare our implementation with the performance of other systems. We derived formulas similar to those that the implementors of Charlotte have presented[3].

Accent time (ms) $= 1180 + 115*s$
V time (ms) $= 80 + 6*s$
Charlotte time (ms) $= 45 + 78*p + 12.2*s + 9.9*r + 1.7*q$
Sprite time (ms) $= 76 + 9.4*f + 0.48*fs + 0.66*s$
SMS time (ms) $\approx 150 + 48*n + 22.8*r + 5.5*s + 5.5*so + 58*t$
OMS time (ms) $\approx 50 + 2.4*n + 7.9*r + 1.9*s + 1.1*so + 5.4*t$

In the above formulas the following parameters have been used. In Charlotte: p is process migration management parameter with values [0,1,0.2], s is the virtual memory size in 2kB blocks, r is 0 if links are

[2] Our measurements are related to task, not process, migration, therefore, additional time should be accounted for. The same is true for most other examples.

[3] Results on Accent, V and Charlotte origin from [Arts89] and for Sprite from [Doug91], since they are more recent.

local, 1 otherwise, and q is the number of nonlocal links. In V, Sprite and Accent, s is VM size in kB, f is the number of open files and fs is file size in kB. In SMS and OMS, r is the number of receive rights, s is the number of send rights, so is the number of send-once rights, and n is the number of regions. According to these formulas, a typical UNIX process (or the corresponding task in relevant cases) would migrate in 330ms in Sprite (exec-time process), 680ms in V, 13 sec in Accent, 750ms in Charlotte, 500ms in SMS and 250ms in OMS. It should be noted that the presented performance measurements have been performed on different computers, therefore direct comparison is inappropriate.

4.2 WPI Benchmarks

We obtained interesting results while running the Jigsaw benchmark [Fink90]. The Jigsaw benchmark stresses the memory management activities of the operating system. It consists of making, scrambling, and solving a puzzle. During each phase, there is an allocation of huge amounts of memory. We repeat similar experiments enhanced by TM in four cases. The first case consists of locally executed Jigsaw. In the second case, we migrate the task after some amount of memory allocation, and thereby distribute memory on both machines. In the third case, the task is migrated after all memory is allocated on the source node. Finally, in the fourth case, the task is executed remotely, resulting in memory allocation mostly on the destination node. These four cases are presented in Figures 11 and 12. The worst performance is achieved in the third case where the task has to access all of its memory remotely. The first and the fourth case have similar performance, since all of the task's memory is on the node where the task executes. The second case exhibits the best performance for the tile sizes which cause excessive paging. In this case, the task has memory on both machines, so it starts thrashing later and achieves better performance. From this example we do not benefit too much, unless we apply similar reasoning as in memory overloaded systems. Distributing the memory across the network could improve performance in much the same way as distributing the load in an overloaded machine.

Finally, in Figure 13 we present the difference between the execution of a real make program and the WPI synthetic gcc compilation, *scomp*. For the real make, there is a performance penalty of more than 500% for the remote execution, while the *scomp* exhibits about a 50% degradation. This simple example demonstrates the inappropriateness of the existing emulator and the current implementation of a mapped file system for TM. The emulator has been designed to support UNIX emulation in a non-distributed environment. The task communicates through the emulator with the server by using shared memory and IPC. In the local case using shared memory for emulator/server communication does not impose a significant overhead. However, it turns out to be inappropriate for a distributed system, due to the unnecessary transfers of a complete page across the network, even for small information exchange. Emulator issues have also been investigated in [Pati93]. Similar reasoning is valid for mapped files. It is not the technique of memory mapped files itself that is a bottleneck; rather it is the current implementation which needs to be optimized for a distributed environment. We expect significant improvements when the existing emulator is replaced and file access optimized. This is, however, related to the particular operating system personality. Therefore, we plan to upgrade to OSF/1 server with a new distributed file system that is currently being developed within the OSF cluster project [Roga92].

4.3 Parallel Make and other Applications

One of the traditional TM applications is a parallel make. We are using *pmake*, written by Adam de Boor for Sprite. We modified it by inserting our migration routines. Due to the emulator problems, we are still not using it extensively. We expect major results to evolve when we switch to a new distributed file system. With current file system support, performance is significantly degraded. We are also running other applications, such as matrix multiplication, simulations, various benchmarks, etc. Most of them have been ported straightforwardly, e.g. simulations, which have been used as common UNIX programs. For matrix multiplication and the traveling salesman problem we have put additional effort to port them to Mach instead of UNIX. These programs run smoothly for small problem sizes, however, for big problem sizes we still encounter problems, which are likely to stem from some current limitations in Norma versions. We expect to have more results on these applications once we start making LD decisions.

5 Related Research

There is a lot of related research in the area of TM and LD. We have chosen some that represents TM on μkernels, monolithic kernels and in user-space.

- **Chorus** was expanded to support TM [?], as a basis for load balancing experiments. The TM consists of elements similar to our scheme, but is applied on the process, instead of the Actor, level (Actors correspond to Mach tasks). It is more oriented towards a hypercube implementation. Some limitations arise from the fact that Chorus currently doesn't support port migration. It is too early, though, to draw deeper comparison with our research, since both projects are at the early stage of development.

- Closely related to our research are University of Utah - **"Schizo"** project [Swan93] and the **TCN** project, performed by Locus for Intel Paragon and OSF/1 AD [Zajc93]. Both projects are Mach based. "Schizo" is targeted at a distributed environment consisting of autonomous workstations. It investigates issues such as autonomy and privacy. "Schizo" inherited some aspects of the Stealth project [Krue91], namely prioritizing VM, file and CPU scheduling, extending it with prioritizing IPC, more robust failure handling, etc. We consider a cluster environment in our work, and therefore do not concern such issues. Our work on task migration has been used as one of the mechanisms for the "Schizo" project. The **TCN** project is concerned with traditional issues of monolithical systems, providing process migration and explicitly considering files, signals and sockets. Currently, TCN is only partially concerned with μkernel TM issues. In the OSF/1 environment, the Mach interface is not exposed to the user, and therefore atomicity of process migration is not affected. A complete solution would be a combination of our work and the work on TCN, i.e. a process migration that makes use of task migration.

- **Sprite** supports one of the important TM implementations [Doug91]. It is tightly coupled with the distributed file system. The address space transfer is optimized by using various caching techniques and by flushing the state onto the server. Compared to Sprite, we have acted on the lower level. We have never dealt with issues such as files or signals. In our scheme, these abstractions are transparently supported by the network IPC and DSM, which correspond to handling open I/O streams and caching in Sprite [?, Doug92]. Therefore, there is an illusion that we have achieved a simpler design by retaining the existing file system instead of introducing the distributed one. However, for performance reasons, the current implementation of signals, files etc., needs to be optimized for distributed systems. This is, however, the flavor of the operating system personality emulated on top of Mach and is related neither to Mach nor to our scheme. It will be hard to compete with Sprite in performance, since their approach allowed for various optimizations regarding caching on the server and client side. Sprite is not only a challenge to our TM, but to Mach itself. All μkernel advantages, such as portability, flexibility, maintainability etc., should be confirmed with performance comparable to monolithic kernels, such as Sprite [Welc91]. Sprite has recently been ported to Mach [Kupf93], unfortunately process migration is not supported, preventing us from interesting comparison.

- **Condor** TM [Litz92] is dedicated to long running jobs that do not need transparency and allow for limitations on system calls the migrated task may issue. Performance penalties in Condor are much higher because it is necessary to dump the core, combine it with executables and return it to the submitting machine. The Condor approach is acceptable for long-running, computation-intensive jobs; for other classes of jobs, it is too expensive. Due to the less expensive migration, our scheme pays off for short running tasks.

6 Conclusion

We have presented a TM design and implementation on top of the Mach μkernel. We explained the reasons why we believe that our choice of a state-of-the-art message passing μkernel for TM implementation is at least comparable to other platforms, while in some aspects it has advantages. We have described minor modifications we had to apply to the Mach μkernel in order to transparently migrate tasks, and two implementations of migration server. Afterwards, we presented some measurements of TM performance, as well as performance of the migrated applications. Finally, we compared our research to the related projects in this area.

As we base our research on Mach, we have tried to use existing Mach features as much as possible. Most of the issues, e.g. network IPC, DSM, and exporting the task/kernel state, have already been provided by Mach.

The work we presented in this paper is a part of wider research on load distribution. Task migration is a problem that we have to solve efficiently, but we expect to achieve major contributions later, when we start using task migration in the framework of load distribution. In the present phase of our work we can summarize the following contributions.

- We showed that it is possible to implement TM on top of the Mach μkernel without much effort, and that μkernels are a suitable level to implement TM on. We have acted in user space with minimal modifications to the kernel. The user space implementation significantly improved maintainability and extensibility.
- Our implementation has achieved high transparency without paying the price in performance comparable to other user space implementations, such as Condor. TM is completely transparent to the migrating task and to the tasks it communicates with. There is no need to link the task with any special libraries, nor are there any limitations to the calls that the task may issue. Since we do not depend on hardware, TM is also portable across the various hardware architectures. We act on the Mach level, transparently supporting various operating system emulations, e.g. UNIX or VMS, as well as applications running on bare Mach.
- We supported memory transfer strategies, such as precopy, flushing and COR in user space. Formerly, only a single strategy per TM implementation has been provided, implemented in the kernel.

OSF Grenoble Research Institute and the University of Utah are currently investigating our TM scheme as a possible technology for use in their "OSF/1 Cluster" and "Schizo" projects respectively. A collaboration with the OSF Grenoble Research Institute is underway, which should involve continuation of our work, namely load information and distributed scheduling.

Our further research is related to TM improvement, as a short-term goal, and to LD, as a long-term goal. In the first phase of TM we have provided functionality; now it is necessary to improve performance. We are going to combine the migration servers into one unique server that will merge the best characteristics of both. Then we need to modify or completely eliminate the emulator, since it has been designed for local execution and as such is not acceptable for distributed systems. It would also be necessary to redesign the mapped file system in order to improve performance. We shall continue to profile various applications and try to learn more about the task characteristics as a function of IPC, VM and processor load. In particular, we shall stress our experiments on IPC and VM dependency. Processor load behavior has been much more researched and existing results could be readily applied, while VM and IPC are less investigated. As a part of this research, we are currently implementing a load information management scheme for Mach and a user level IPC profiler.

As a long-term goal, we have started to work on distributed scheduling. We are interested in the relation of global and local scheduling as suggested by Krueger [Krue87]. Krueger shows that not every local scheduling algorithm matches the distributed one and vice versa. In traditional systems it was hard to access and modify local schedulers, while Mach may be suitable for such experiments [Blac90].

7 Availability

All mentioned programs are available upon request. They have still not been put for anonymous ftp access due to the continuous adjustments to new norma versions. It should be noted that OMS server is not supported anymore, although the last version is available. Beside TM related programs, we have available preliminary versions of load information manager (including user space IPC profiler) and simple distributed scheduler, supporting few basic strategies. Complete LD scheme with numerous strategies is expected to be finished in the second half of 1993.

8 Acknowledgements

We would like to thank: CMU and OSF people for providing us with Mach and with continuous support; Joe Barrera for important hints during development; Fred Douglis and Brent Welch for discussing Mach and Sprite peculiarities; Prof. Dušan Velašević for initial encouragement and support; Prof. Amnon Barak and Prof. Andzej Goscinski for continuous support; Prof. Nehmer for making all of this happen. Special thanks are due to Fred Douglis, Mike Kupfer, Jelena Vučetić, Nikola Šerbedžija, Sean O'Neill and Holger Assenmacher for proofreading the paper and giving many useful suggestions. Finally we would like to thank the program committee, the chairman David Black, and our liaison to program committee, Jay Lepreau, for many useful suggestions and for giving us the opportunity to present our work.

References

[Alon88] ALONSO, R. and KYRIMIS, K. (February 1988) *A Process Migration Implementation for a Unix System.* Proceedings of the USENIX Winter Conference, pages 365–372.

[Arts89] ARTSY, Y. and FINKEL, R. (September 1989) *Designing a Process Migration Facility: The Charlotte Experience.* IEEE Computer, pages 47–56.

[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.

[Barr93] BARRERA, J. (1993) *Operating System Support for Multicomputers.* Technical Report CMU-CS-93-?, PhD Thesis in Preparation, Carnegie Mellon University.

[Blac90] BLACK, D. (October 1990) *The Mach Timing Facility: An Implementation of Accurate Low-Overhead Usage Timing.* Proceedings of the USENIX Mach Workshop, pages 53–73.

[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.

[Bokh79] BOKHARI, S. H. (July 1979) *Dual Processor Scheduling with Dynamic Reassignment.* IEEE Transactions on Software Engineering, SE-5:326–334.

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

[Doug92] DOUGLIS, F. Personal communication 1992.

[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.

[Fink90] FINKEL, D., KINICKI, R. E., AJU, J., NICHOLS, B., and RAO, S. (October 1990) *Developing Benchmarks to Measure the Performance of the Mach Operating System.* Proceedings of the USENIX Mach Workshop, pages 93–100.

[Fori91] FORIN, A., GOLUB, D., and BERSHAD, B. (November 1991) *An I/O System for Mach 3.0.* Proceedings of the Second USENIX Mach Symposium, pages 163–176.

[Free91] FREEDMAN, D. (January, 1991) *Experience Building a Process Migration Subsystem for UNIX.* Winter USENIX Conference, pages 349–355.

[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.

[Gosc91] Goscinski, A. (1991) *Distributed Operating Systems The Logical Design.* Addison-Wesley.

[Krue87] Krueger, P. and Livny, M. (August 1987) *When is the Best Load Sharing Algorithm a Load Balancing Algorithm.* Technical Report 694, 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.

[Lux92] Lux, W., Kuhnhauser, W. E., and Hartig, H. (June 1992) *The BirliX Migration Mechanism.* Proceedings of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems Programs, pages 83–90.

[Mill81] Miller, B. and Presotto, D. (1981) *XOS: an Operating System for the XTREE Architecture.* Operating Systems Review, 2:21–32.

[Milo91] Milojicic, D., Pjevac, M., and Velasevic, D. (September 1991) *Load Balancing Survey.* Proceedings of the Summer EurOpen Conference, pages 157–172.

[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.

[Roga92] Rogado, J. (October 1992) *A Strawman Proposal for the Cluster Project.* Technical Report, OSF RI, Grenoble.

[Swan93] Swanson, M., Stoller, L., Critchlow, T., and Kessler, R. (April 1993) *The Design of the Schizophrenic Workstation System.* Proceedings of the third USENIX Mach Symposium, pages 291–306.

[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.

[Walk89] Walker, B. J. and Mathews, R. M. (Winter 1989) *Process Migration in AIX's Transparent Computing Facility.* IEEE TCOS Newsletter, 3(1):5–7.

[Welc91] Welch, B. (November 1991) *The File System Belongs in the Kernel.* Proceedings of the Second USENIX Mach Symposium, pages 233–250.

[Wiec92] Wiecek, C. A. (April 1992) *A Model and Prototype of VMS Using the Mach 3.0 Kernel.* Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pages 187–204.

[Youn89] Young, M. W. (November 1989) *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* Technical Report CMU-CS-89-202, PhD Thesis, Carnegie Mellon University.

[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.

A Measurements

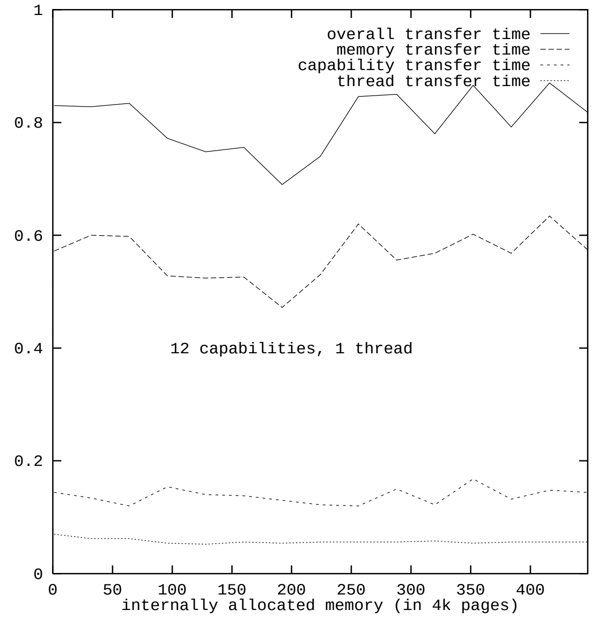

Figure 3. Transfer Time (in sec) v. Memory Size

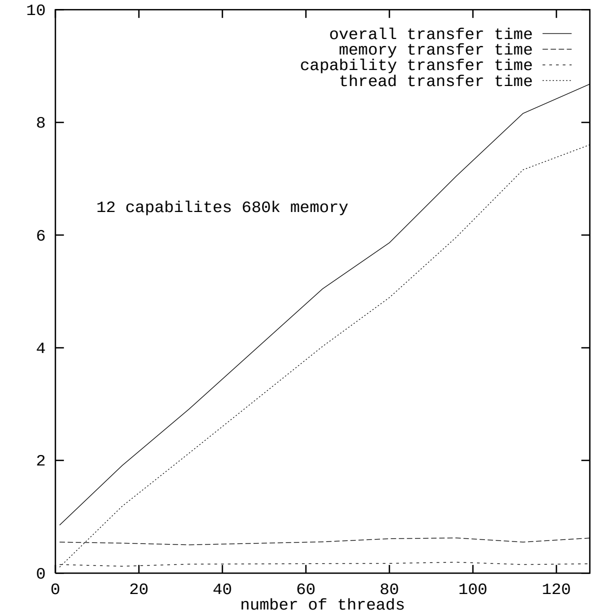

Figure 4. Transfer Time (in sec) v. Thread Number

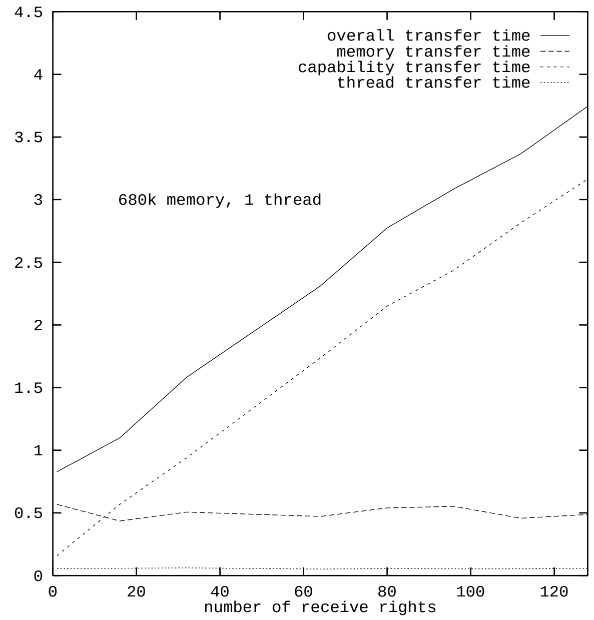

Figure 5. Transfer Time (in sec) v. Number of Receive Rights

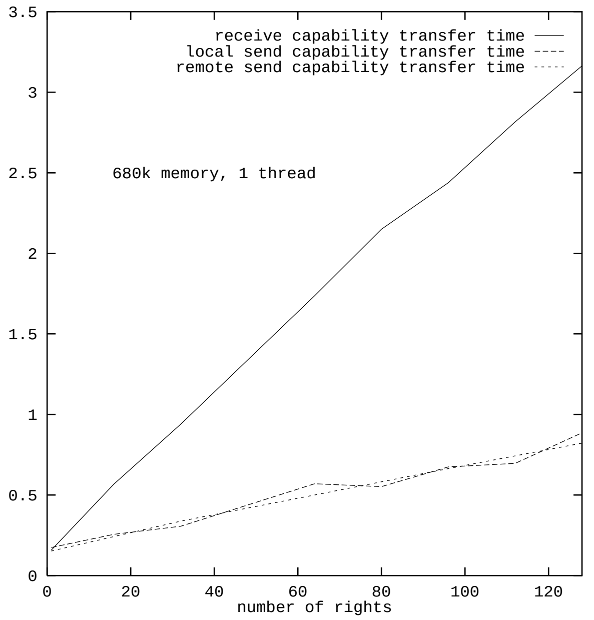

Figure 6. Rights Transfer Time (in sec) v. Number of Rights

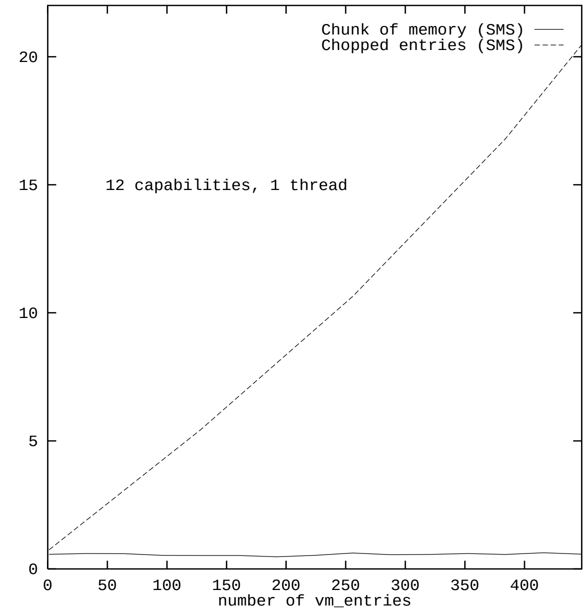

Figure 7. Transfer Time (in sec) v. Memory Contiguity

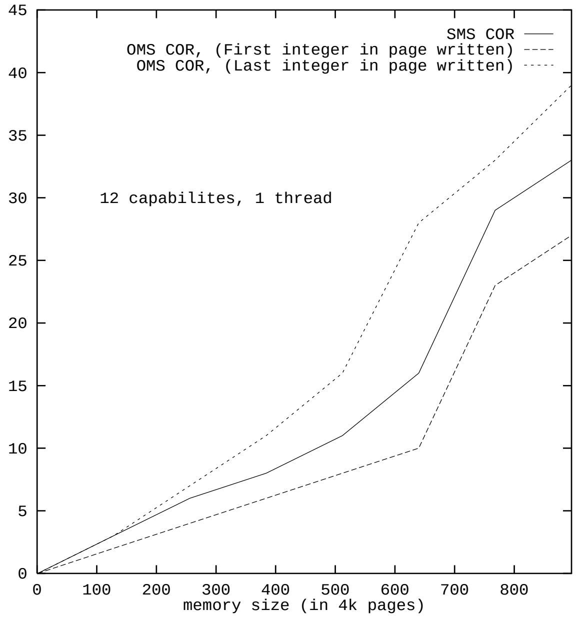

Figure 8. Execution Time (in sec) v. Memory Size, (First/Last Integer in Page Written)

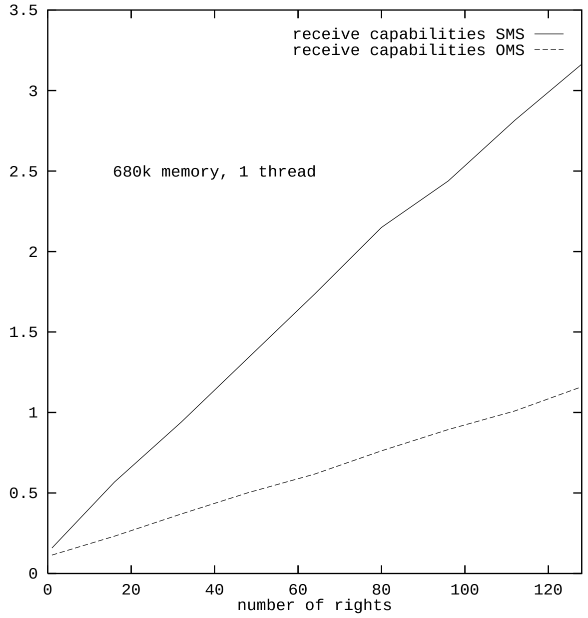

Figure 9. Transfer Time (in sec) v. Number of Receive Rights

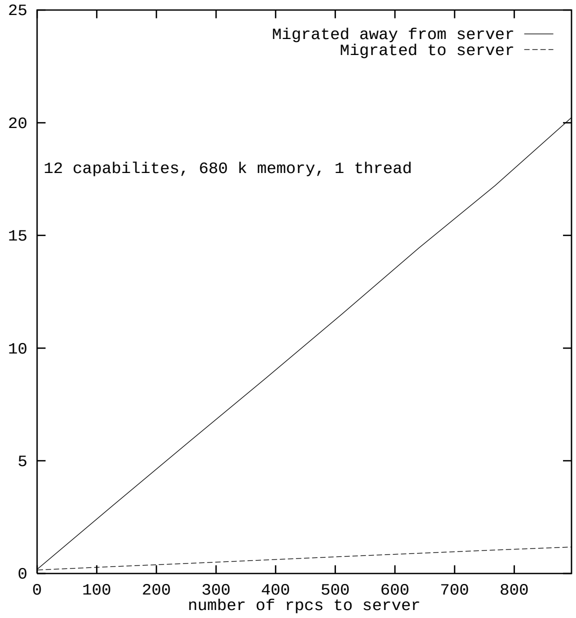

Figure 10. Execution Time (in sec) v. Remote/Local IPC

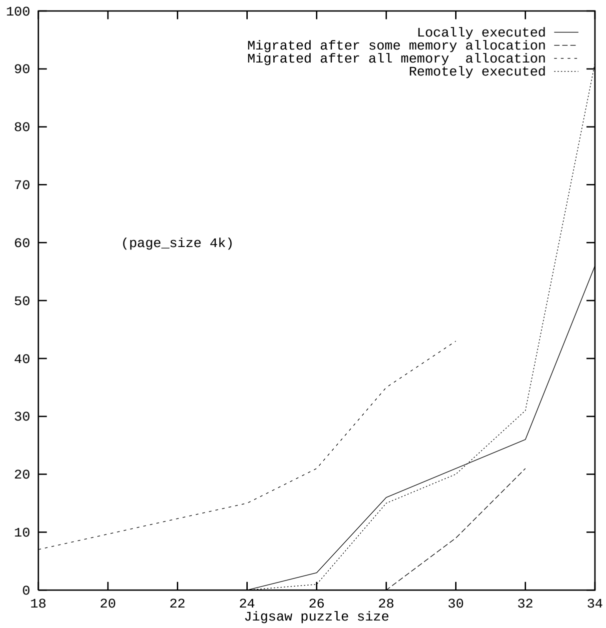

Figure 11. Solving Time (in sec) v. Jigsaw Puzzle Size

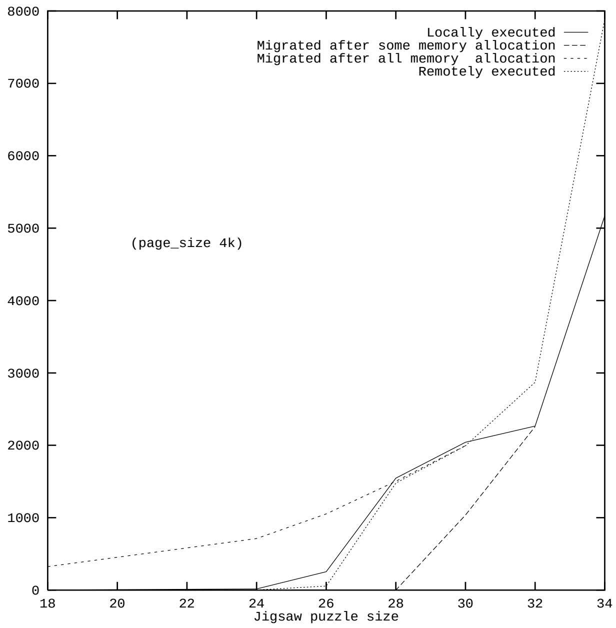

Figure 12. Paging (Number of Pageins & Pageouts) v. Jigsaw Puzzle Size

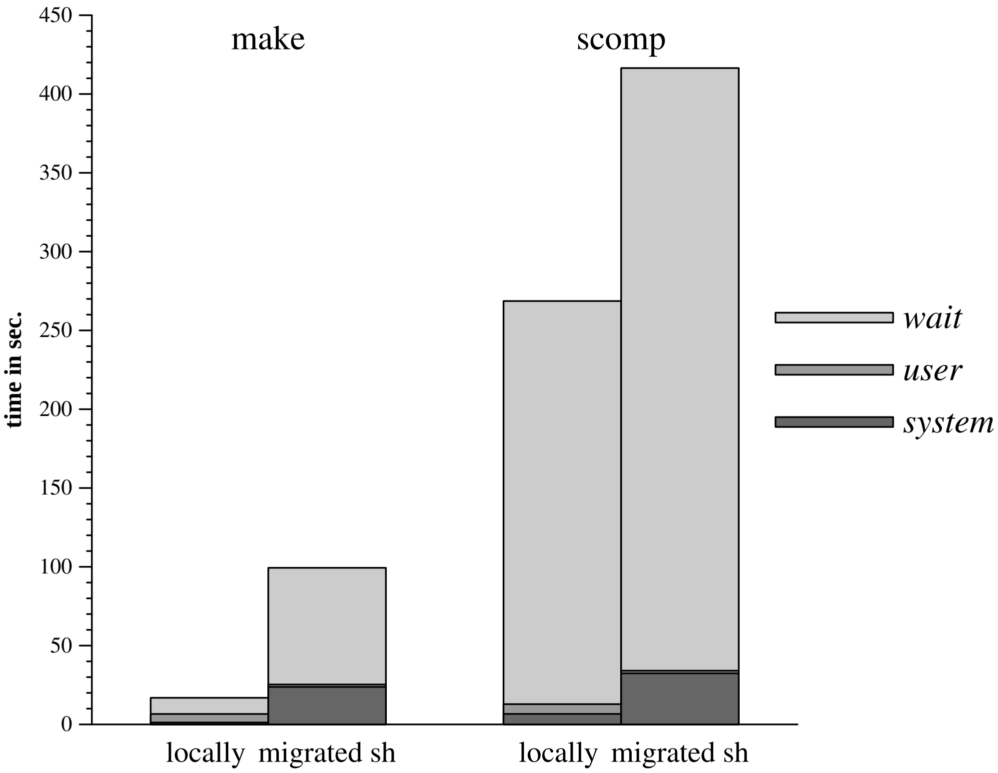

Figure 13: *make ratio.*