Process vs. Task Migration

Yves Paindaveine and Dejan S. Milojičić
OSF Research Institute

Abstract

This paper describes task and process migration for the OSF/1 AD Operating System (OS) server for massively parallel processors and clusters of workstations. OSF/1 AD runs in user space, on top of the Mach microkernel. A process in the OSF/1 AD server is composed of a Mach task state (memory, capabilities, threads, etc.) and of the UNIX-related state (open file descriptors, signal masks, etc.). Process migration invokes task migration to transfer the Mach task state, and supports transparent transfer of the UNIX process state. Process and task migration rely on a single system image provided at both OSF/1 AD and Mach levels.

We compare the initial and run-time costs of task and process migration for several programs with different needs. While process migration initial costs are less than 17% of task migration costs, the expected benefits can easily be as high as a 50% improvement in run-time cost over task migration. We conducted the experiments on a cluster of PCs; however, results are also applicable to massively parallel processors.

1. Introduction

Process migration is a facility to dynamically relocate a process from the node on which it is executing (*source*) to another homogeneous node (*destination*). It has been used for performance improvement, by using idle hosts; system administration, by moving uninterruptible processes; data locality exploitation; repartitioning in massively parallel processors, etc.

Process migration has not been extensively used over the past years, although it was designed and implemented in many systems such as [2][3][4][7][12][21][22][24]. We believe that there was not sufficient infrastructure in earlier systems with process migration, such as Sprite [7], V kernel [21], Accent [24]. These were research operating systems with limited use, typically at one or more Universities.

Nowadays distributed systems are common rather than exceptional. Load distribution is increasingly needed, and the existing mechanisms do not always fulfill requirements. Previous attempts showed that transparent UNIX process migration was hard to implement. Modern microkernels and operating system servers running on top of them allow for an easier and more modular implementation of process migration.

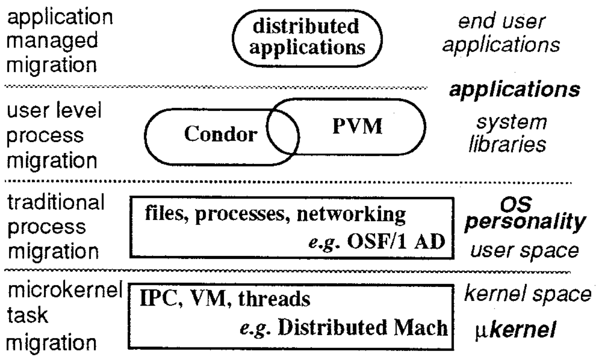

Figure 1. Migration levels: *microkernel; OS server; user level; part of application. They differ in performance, transparency and reusability.*

Migration can be achieved at various levels in a system. It can be implemented in the operating system, as was the case with many monolithic operating systems, such as MOS(IX) [4], Locus [22], Sprite [7], etc. Then there are examples of migration for microkernels, such as the V kernel [21] and Mach [13]. There are also examples of user level migration implementations such as in Condor [11], and on top of UNIX [2][12]. In these systems, migration was designed outside of the kernel, independent of applications. Finally, there is migration performed by the applications on their own behalf [8][20].

We can compare various migration implementations using characteristics such as performance, complexity, transparency and reusability. We observe that when migration is applied at higher level, the implementation is simpler; however, performance, transparency and reusability are reduced. Lower-level migration has more complex implementation, but better performance and higher degrees of transparency and reusability result. At the far end, when an application migrates itself across the nodes, duplication of most of the migration facility for each subsequent application is required, leading to poor reusability and transparency (See Figure 1). However,

This research was supported in part by the Belgian Incentive Program "Information Technology" of the Prime Minister's Science Policy Office, contract IT/IF/12, and in part by the Advanced Research Projects Agency (ARPA) and the Air Force Material Command (AFMC). Some of the results described in the paper are based on previous work of the authors while they were at the Université catholique de Louvain (UCL), Louvain-la-Neuve, Belgium and University of Kaiserslautern, Germany.

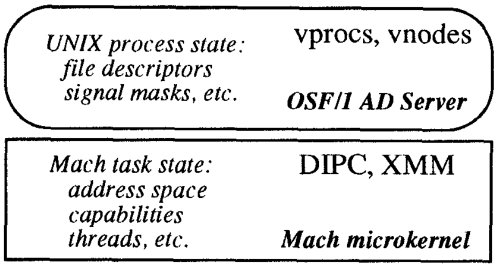

Figure 2. UNIX process and Mach task state: *Vprocs and vnodes support transparent access to remote processes and files; DIPC and XMM to remote capabilities and remote memory.*

applications at higher levels can often take advantage of having more knowledge about particular resource usage, leading to better execution times.

In our design, it is possible to migrate at both the microkernel and the operating system levels, called respectively task migration and process migration. The difference in granularity affects the migrated applications, depending on the ratio of UNIX process-related system calls to Mach kernel calls and computation. These reasons will also determine which migration mechanism to use.

In this paper we would like to demonstrate that our migration mechanisms are relatively simple to implement and cheap to use. Simplicity and low cost are based on the existing support for single system image at both the microkernel and operating-system server levels (see Figure 2). This paper describes our implementation of distributed process management and migration and our experiences using it. We shall explore the strengths and weaknesses of each implementation. The paper does not address the scheduling (*i.e.* policy) aspects of process migration; it is more focused on migration as a mechanism.

The rest of this paper is organized as follows. Section 2 overviews the underlying environment and Mach task migration. In Section 3, we describe the process migration design. Section 4 presents measurements. Section 5 compares our work to related projects. We present conclusions and future work in Section 6.

2. Background

This section overviews Mach and OSF/1 AD; distributed process management; and task migration.

2.1 Mach and OSF/1 AD

The Mach 3.0 Microkernel [1] runs on all nodes of the processing environment, providing the following basic facilities on each node: task and thread management, IPC, memory management and device access. All services and resources exported by the microkernel or by any task are represented as capabilities.

One of the important aspects of the underlying environment is the support for a single system image which can significantly reduce the complexity of the migration mechanism. It also facilitates migration transparency. Today, microkernels such as Mach [1], Chorus [19], Spring [9], etc. support such transparent network IPC.

Mach benefits from two kernel subsystems to provide transparent internode communication using a message-passing interface: Distributed IPC (DIPC) [10] and distributed memory management, known as eXtended Memory Management (XMM) [5]. Distributed IPC extends Mach IPC in a way that permits applications running on any node to view Mach abstractions such as tasks, threads, memory objects and ports in a transparent way. XMM supports distributed shared memory.

OSF/1 AD version 1 [23] was the first prototype of the OSF/1 operating system to provide a scalable, high-performance single system image of UNIX. It is intended for massively parallel processing environments, such as the MPP Intel Paragon, but also for clusters of interconnected workstations. OSF/1 AD is composed of specialized servers distributed on the nodes (file service, process management, credentials service, token service, *etc.*). The major features are the distributed file system with a single file name space across the nodes and the process management framework.

A UNIX process is "mapped" to a Mach task and threads managed by the micro-kernel. In OSF/1 AD, a thread executes the code of the process. To provide UNIX functionalities to the Mach threads, an emulation library (*emulator*) residing in each task intercepts the system calls and converts them into service request message for a UNIX server.

At the OS server level, a *virtual process* (or *vproc*) framework supports transparent operations on the processes, independently of the actual process's location. By analogy, *vprocs* are to processes what *vnodes* are to files, providing transparency of both location and heterogeneity at the system call interface. In OSF/1 AD TNC, the process management

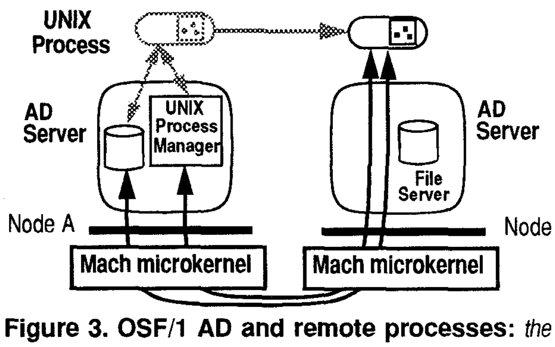

Figure 3. OSF/1 AD and remote processes: *the AD server on node A has the full functionality of a traditional UNIX. Node B is a repository for a file server and Mach tasks but has no process management functionality. Tasks created remotely or migrated away dispatch the system calls back to node A.*

subsystem (produced by Locus Computing Corporation) is distributed across the nodes of the processing environment. However, the version of OSF/1 AD we started with did not contain this distributed mechanism, but a simpler centralized subsystem on a single node that allowed only Mach tasks to be created remotely. In this case (see Figure 3), the process management subsystem exists only on node A and tasks created remotely dispatch every call back to it.

2.2 Distributed Process Management

Our main motivation for distributed process management in a multicomputer environment is the need for transparent access to all processes and support for remote processing. Process migration has proven difficult to implement in distributed operating systems, but the difficulty arose more from building a reliable distributed operating system than from building a transparent migration facility. Our design for this subsystem was based on the available documentation at that time for OSF/1 AD version 2 [16]. The implemented *vproc* framework is based on this simple and open architecture which divides into three modules with different semantics.

A **Process Management Agent** (PMA) or *system call* agent, distributed on each node, maintains physical data structures (*e.g.* process locks, wait channels, signals, etc.) for the processes co-resident on the same node. It corresponds to the UNIX process management core present in traditional UNIX operating systems. Any operation on behalf of a process on any other process (be it a local process or not) is performed by invoking the *vproc* framework.

The **Virtual Process Interface** consists of a *vproc* structure and an operation table associated with each *vproc* structure. It provides transparent access to any process regardless of its actual physical location. The operation table is set to local PMA operations or to *client stubs* according to the location of the process. The *server stubs* receive remote requests to be executed locally. The *vproc* framework allowed us to replace a minimal distributed process management subsystem easily with a subsystem providing a single system image and a transparent distribution of the UNIX processes on the different nodes [14].

A **Process Management Master** (PM Master), located on a single node in the current implementation, handles inter-process, process-session and process group relationships as well as process identifier (*pid*) resolution and initial *pid* brokerage. The PM Master is contacted only by the *vproc* framework whenever the aforementioned relationships or the process location are involved (*e.g.* during the *fork()*, *exit()*, *wait()* and *killpg()* system calls).

Since the PMA has exclusive access to the local state of the process, there must be some means for the PMA and other subsystems, such as distributed file servers, to share per-process information (e.g user limits, resource usage) without

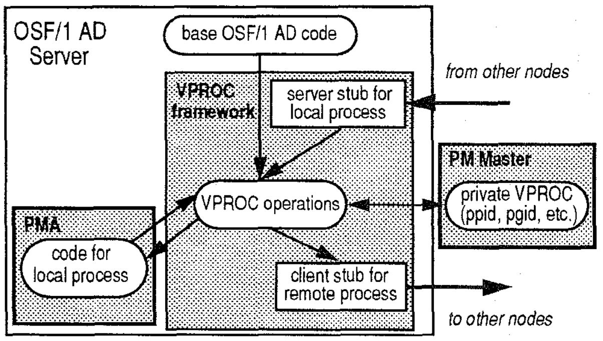

Figure 4. Distributed process management: *VPROC framework hides process location by interposing local process management. Transparent access to a remote process is managed by client stubs calling the appropriate remote server stub. PMM handles process relationships.*

accessing the corresponding physical data structure. A third-party server – the *credentials server* – plays this role [23].

2.3 Mach Task Migration

The process migration described in this paper relies on the task migration package that was developed by Milojicic *et al.* at the University of Kaiserslautern [13]. It is important to note that although implementation is done at a relatively low level, *i.e.* on top of the Mach microkernel, it is possible to achieve simple design and implementation.

Task migration has been implemented in three forms. In the Optimized Migration Server (OMS), different address-space migration strategies are supported by the user-space pager by mapping the address space of the migrated task and accepting requests from the remote task. In this way, strategies such as *copy on reference, eager copy, flushing* and *pre-copy* are implemented.

In its second form, the so-called Simple Migration Server (SMS), migration is achieved by relying on the kernel support for address space transfer. The default kernel strategy for address space migration is *copy on reference.* Process migration uses this form for its ease of use.

User-level migration demonstrated that it is possible to achieve migration completely in user space with minimal changes to the underlying microkernel. The only required change was done in order to extract the task kernel port.

Finally, task migration is also available in the Mach microkernel: SMS was reimplemented in the kernel space. This was a relatively straightforward task. No significant performance improvement was observed, because the migration costs were dominated by the network messages, and not by the local inter-domain calls.

3. Process Migration

The rationale behind this higher level of granularity -UNIX process migration - with respect to task migration is based on the unnecessary residual dependencies on the PMA left over by task migration alone. As an additional reason for UNIX process migration, we want to provide an open testbed for the study of load leveling policies in a truly distributed system offered to user-space (UNIX) processes. The process migration testbed described in this paper was previously designed and implemented by Y. Paindaveine when a research assistant at the Université catholique de Louvain [15]. The following design decisions have guided us throughout the implementation process:

We decided to preserve the full UNIX model (such as the UNIX permissions) but also not to restrict which processes could be migrated (not even on the number of threads a UNIX process can contain). As a consequence, using pipes, signals, etc. does not prevent a process from being migrated. Traditional process resource usage (*e.g.* user and system times) are returned as accumulated values from previous usages and migration(s), although the usage since the process has existed on its current node can also be obtained, being essential information for a migration policy.

Process migration can be initiated at an arbitrary time; neither the running process nor any other process interacting with it (not even their emulator) is aware of the migration. Single system image is preserved by retaining the process identifier after migration (its uniqueness is ensured by the *pid* allocation mechanism), its process relationships and all UNIX states as if it had never migrated. Transparency also means maintaining continuous access to all its resources. The solution adopted leaves no forwarding stub processes or chains: it leaves no residual dependencies on previous process management subsystems. Processes using shared memory when migrated continue to execute, as before, but now with distributed shared memory as provided by XMM.

3.1 Issues

Continued access to open files and sharing of the file position without modification to the semantic behavior is a particularly intricate point for most operating systems endowed with process migration. Since every UNIX resource is accessed through the use of capabilities, each time a file is opened, a file structure containing its state (file reference, file position, etc.) is created and associated with a capability in the file server. The file server distributes capabilities for this structure which can easily circulate between the nodes. Since a child process receives a copy of its parent's capabilities, the problem of migrating open file descriptors is simplified by the OSF/1 AD distributed file system design. The files themselves are not migrated but rather remain on the file server where they physically reside.

Another issue is the control over latency between the time when the decision to migrate is taken and the time when the migration really begins. Due to the fact that the different states are isolated and manipulated only by the managing entity, our implementation is satisfactory compared to implementations where distributed state is involved.

More specifically, the issues of process migration are the definition of a **minimal process state** to allow the process to resume its execution successfully on the destination node, leaving no residual dependencies on previous source node(s); and the guarantee that the process state can be extracted from the source node in a **stable, consistent and resumable form**. Resulting from the process management design and from the message-based microkernel, the process state is divided into three semantically different states: the Mach task state; the UNIX process-local state; and process-relationship state. Each state is managed by a different migration module. The variety of states does not make process migration more complex. Instead, the different state semantics favor the layering and modularity of the process migration mechanism. Our experience shows that getting a consistent state that is guaranteed to be restartable is not a trivial problem.

The **Mach task state** encompasses an execution state (per thread: register values, condition codes, scheduling information, priorities, etc.), an address-space state (memory objects for text, data, stack and bss), a communication state (capabilities and ports with their local pending-message queues but also messages in transit) and additional kernel state (task/thread kernel ports, bootstrap ports and exception ports). The Mach task state is managed by the task migration facility with the support of distributed shared memory (XMM) and distributed IPC (DIPC).

The **local process state** is maintained by the PMA that runs on the same node as the process. It corresponds to the typical UNIX **proc** and **user** structures, and consists of locks, signal masks, etc. Open file descriptors, although part of the UNIX process state, are stored in and managed by the emulator which communicates directly with the file servers. Open file descriptors do not pertain to the local process state. Since the emulator is migrated as part of the Mach task, there is no need to explicitly extract this information from the source node PMA and to transfer it to the destination node PMA. Other state related to the emulation of a UNIX process (also transferred with the Mach task state) involves capabilities for the PMA: file tokens, pipes and sockets. The system calls currently served by the PMA as well as the new requests coming from the process are part of the local process state.

The local process state is the sole property of the PMA and freezing the state does not require any other servers to be involved in the migration mechanism. The process can receive messages during migration, since the system calls to subsystems other than the PMA are not part of the local process state and the replies will only be delayed in their execution; they are queued in the task port space and migrated or forwarded in a uniform fashion to the process.

The **process-relationship state** is managed by the PM Master. It remains unmodified since no inter-process relationship is modified by the transparent process migration facility. The PM Master only has to be contacted to update the process's current execution node after process migration.

3.2 Process Migration Mechanism

This subsection deals with the various phases of migration, *i.e.* what information, how and when to transfer. The overall mechanism is relatively simple: (a) freeze the process, (b) detach it from the current PMA and initiate the migration, (c) attach it to the destination node PMA, (d) resume the execution on the destination node and (e) clean up the migration-related process state on the source node.

(a) Freezing the process. The migrating process is frozen when all of the state is in a **stable, coherent and resumable** form. For this to take place, incoming requests to operate on the process are enqueued, the process's execution is suspended and completion of every system call currently being executed by the PMA is awaited.

The virtual process is used to isolate the process and to ensure that subsequent requests which operate on the *local process state* are enqueued in the migration-related process state. This allows for **suspending the communication channels**. The incoming requests are enqueued after being received by the PMA but before being handled. When the migration completes, these requests will be forwarded to the appropriate PMA.

Next, the **process's execution is suspended** by suspending the execution of all the threads in the process, in order to prevent them from modifying the process state. Finally, the completion of the system calls currently being executed on behalf of the process is awaited until there are no further system calls in progress. Waiting for the completion of every system call increases migration latency. The alternative is to abort the system calls that do not modify the process's state (*e.g. wait(), sigsuspend()*), and restart their execution on the destination node PMA.

(b) Detach the process from the PMA and initiate migration. At this stage, it is guaranteed that the local process state will remain stable and coherent and that the process is not exiting due to a previous system call. The local process state is detached from the PMA: the process is removed from the *active processes* list and placed onto a *migrating processes* list for recovery purposes. The process's data structure, the extracted resource usages and credentials entry are copied and marshalled for transmission (the capabilities are copied as well). A copy of the local process state is now ready for

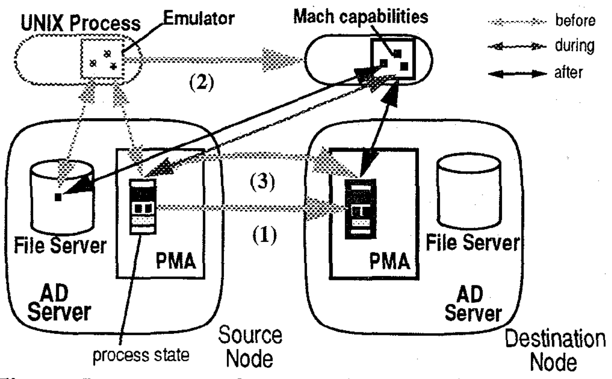

Figure 5. process migration phases: *after being frozen, (1) the process is state is extracted from the source node and migration request is sent to the destination node; (2) the destination node pulls over the task state; (3) the task is reattached to destination PMA. Dependencies on file servers remain unchanged.*

transfer and is sent to the destination node in an attempt to migrate the UNIX process, illustrated by (1) in Figure 5. At this stage, a migration refusal is a possible reply to this attempt.

(c) Install and attach the process on the destination node. Upon receipt of the local process state and acceptance of the migration, the PMA on the destination node allocates a new process data structure and a virtual process with the same process identifier. When the resources have been allocated, the PMA retrieves the Mach task state "mapping" the UNIX process by invoking the task migration mechanism (illustrated by (2) in Figure 5). The **receiver-pull approach** (as opposed to sender-push), by controlling the incoming task migrations, prevents the destination node from being flooded. After the PMA has received the Mach task state, the remaining process state is installed as in process creation. No provision is made to recreate the scheduling priority of the migrating process (since a stalling source node might have changed the process priority, according to the source node load). At this point, the process is local to the destination node, as is the local process state. However, if the process were restarted immediately, subsequent system calls involving process management would be directed to the source node PMA. In order to prevent these performance penalties, the process is reattached to the destination node PMA by replacing the capabilities pointing to the source node PMA with capabilities pointing to the destination PMA (illustrated by (3) in Figure 5). The other capabilities (*e.g.* open files) remain unchanged.

Up until the task migration, the source node PMA can abandon migration and restart the process. If the destination node crashes after the task migrates, the process dies like any other local process. Should the source node crash, one might think that the process could be restarted on this node if no system call request was enqueued on the source node. However, due to the lazy transfer of virtual memory, the process is still dependent on the source node for its address space and consequently dies. Upon migration refusal, the PMA cleans up the received state and capabilities. The refusal may emanate from an exceeded resource usage limit or a exhaustion of a resource (most likely memory). The source node is notified of the refusal and we skip to (e).

(d) On the destination node, inform the PMM of the migration and restart the process. The PM Master is informed of the new execution node of the process and the process is then placed on the *active processes* list and restarted in whatever state it was in before migration. Although clean up and forwarding of system call requests has not yet taken place on the source node, the process is able to resume execution and, if there are no pending system call requests (as is often the case in our experience), it starts executing immediately. The destination node PMA has now completed and returns control to the source node, along with the new PMA capabilities.

(e) Clean up the source node, and forward the pending requests. Upon successful migration, the source node PMA forwards to the destination node the pending system-call requests that have been queued since process migration started, if any. The PMA then performs the clean-up by destroying all data structures and capabilities saved before migration. The process data structure is taken off the *migrating processes* list and returned to the *free processes* list. The *vproc* is freed if not used anymore on this node. At this point, no residual dependencies exist on the source node PMA (there is no such concept as a home node in this design).

The source node PMA, however, is still pointed to by the *vproc* capabilities for this process on other nodes: the remote *vproc* capabilities are lazily updated. This PMA reports an *absent process* error to the calling PMA which will, in turn, contact the PM Master for a new capability. The source node PMA does not perform embedded remote procedure calls on behalf of the calling PMA (to prevent deadlock, among other reasons). In the case of migration refusal (a legitimate reply to the migration attempt), the process is reinstated, moved to the *active processes* list and the refusal is propagated to the calling process.

The migration phases are simple and proceed from the definition of a process's state and the logical analysis of the server. Our experience with process migration is that it is fairly easy to design.

4. Measurements and Implementation Status

We conducted our experiments on a cluster of three i386/25MHz machines with 16Mbytes of main memory connected by a 10Mbit/s Ethernet on a non-dedicated network, with homogeneous low system load. The costs associated with process creation (see Table 1) provide us with an indication of the consequences of a single centralized PM Master and can be attributed to the inter-node communication costs from the underlying kernel IPC (see Table 2).

In order to keep the costs associated with a central coordinator constant for migrated and unmigrated processes, we adopted the testing configuration as shown Figure 6. The single PM Master is viewed in a symmetric way by the two test nodes.

Null process creation (*fork; exec; exit*)	Time
Local, on Master node	131 ms
Local, on non-PM Master node	218 ms
Remote (average)	750 ms

Table 1. Process creation costs

Basic IPC mechanisms (Mach 3.0 version 14.8)	Time (single node)	Time (inter-node)
Mach kernel call (*host_get_time()*)	540 ms	-
Null asynchronous IPC (send only)	750 ms	6500 ms
Null synchronous IPC (RPC)	750 ms	7500 ms
Additional capability transfer	350 ms/cap	1300 ms/cap
Inline memory transfer (1 KB)	2 ms	12 ms
Memory transfer (4 KB)	6 ms	28.7 ms

Table 2. Basic communication costs

A centralized monitor collects statistics from processes and migrations: *total migration time* or *freeze time* (time during which the process execution is suspended); *task migration time* (time spent migrating Mach task state); *freezing time* (latency time between the decision to migrate and the migration itself); and traditional process execution times. No provision was made, however, to correct the accounting overhead generated during migration.

The state transfer consists of a fixed overhead to transfer

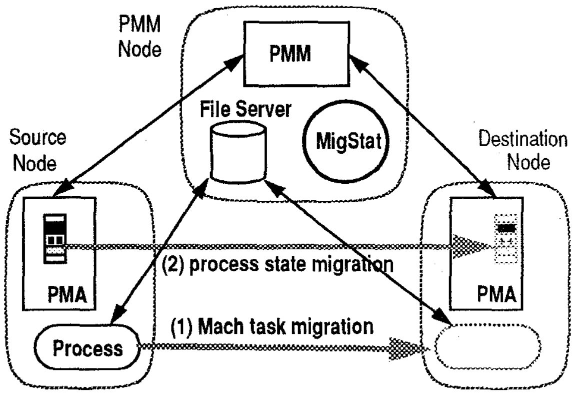

Figure 6. Experimental configuration: *the master node on which the PM Master, the root file server and the statistics collection monitor run, and two test nodes. In (1), only task migration is used between the source and destination, keeping the process on the source node PMA. In (2), the process is also migrated.*

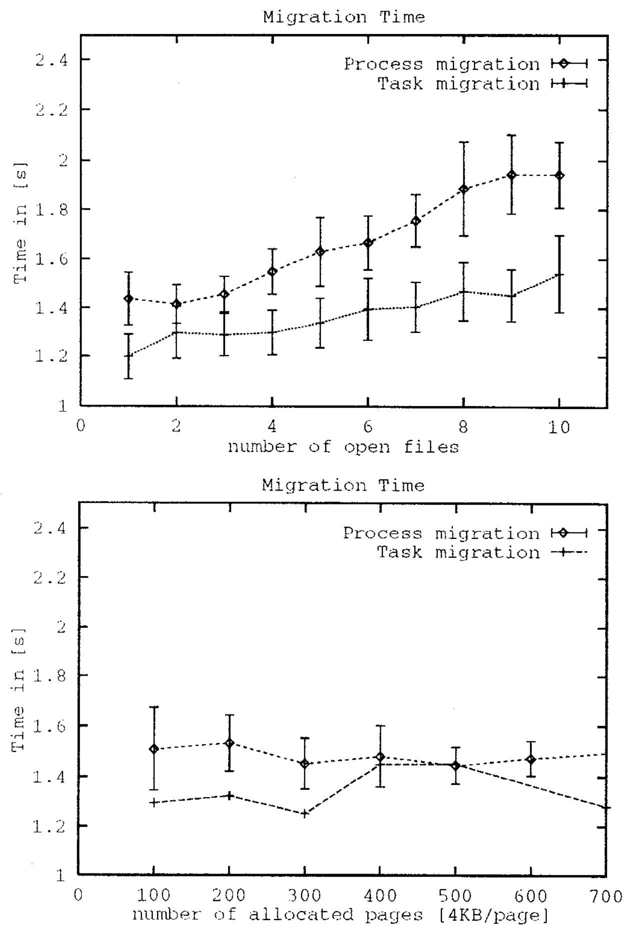

Figure 7. Process and task migration times *as a function of the number of open files (above) and of the pages allocated before migration (below).*

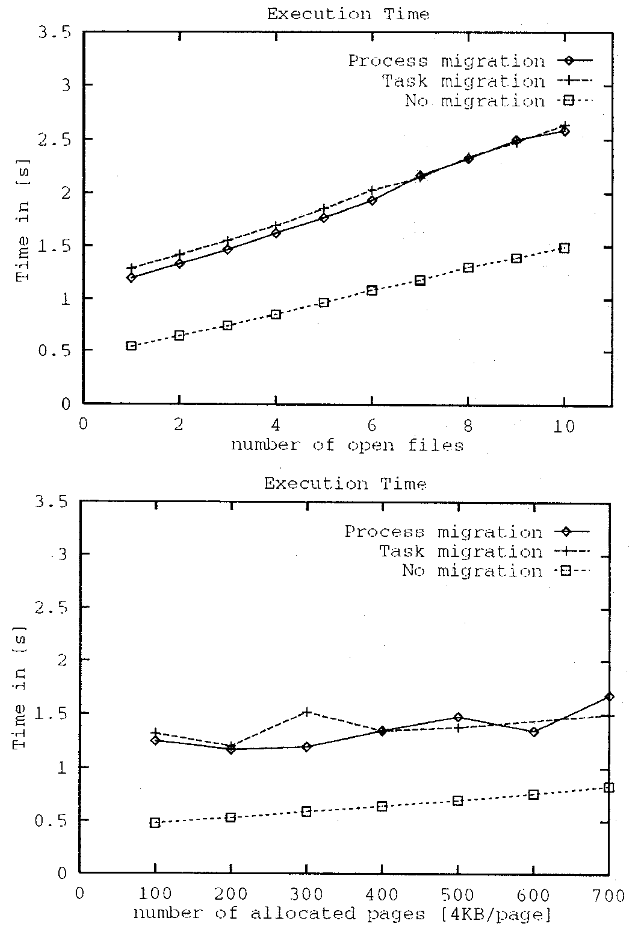

Figure 8. Process execution times *(diminished by the migration time) as a function of the number of open files (above) and of the pages allocated before migration (below).*

data and allocate and initialize a new process, as well as a variable overhead proportional to the amount of virtual memory, capabilities and open files. The average total migration time for a null process (46KB address space, 2 threads, 8 capabilities, 4 ports and 13 memory regions) is 1450ms for process migration and 1240ms for task migration (split into 90ms and 300ms for migration of threads and capabilities and 850ms for address space and kernel state). From Figure 7, we deduce the following formulæ:

$process\_migration\_time = 1450 + 0.02*s + 81*f$ [ms]

$task\_migration\_time = 1240 + 0.02*s + 36*f$ [ms]

where s equals the size of the virtual space (in pages of 4KB) and f the number of open files. Comparison with other implementations is difficult, due to the different hardware (memory, CPU, network and network connection board).

Elsewhere, task migration time has been reported as a function of the number of threads, capabilities and virtual memory regions; the same task migration algorithm has been shown to be competitive with previous implementations of migration [13].

The migration time is linearly dependent on the number of open file descriptors but independent of the task address space. The former is explained by the migration of a linearly increasing number of capabilities for the file descriptors and associated file tokens and sometimes of a request queue attached to a capability (the queue is likely to grow as the freeze time increases). The latter is due to lazy evaluation of the virtual address space which is only migrated on reference, explaining the difference in execution time between migrated and unmigrated processes (see Figure 8).

The migration of the process structure only takes an additional 200ms on average which is 14% of total process migration time. Depending on the total execution time of the process and on the ratio of system calls to other activities

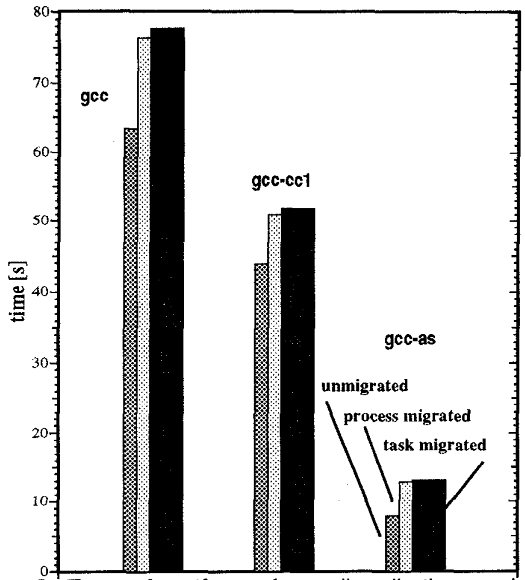

Figure 9. Execution times *for small applications, such as a single gcc, gcc-cc1 and gcc-as.*

(computation and I/O which are dependent on a remote file server in both cases), this overhead can swing in favor of process migration. Mach task migration, however, is most suitable for applications such as ray-tracing algorithms, scientific number-crunching programs or any compute-bound applications not requiring full UNIX functionalities to migrate on nodes on which the server does not even execute (*i.e.* pure compute nodes).

We made preliminary measurements by migrating small, simple applications, as well as more complex ones, with both mechanisms. The first set of experiments targeted gcc (gcc is only a small front-end calling a parser, an assembly-code generator, a compiler and a linker in turn), gcc-cc1, gcc-as and troff (not shown). The second set measured the benefits of migrating real jobs *i.e.* a Make process for huge compilations, and text formatting. We expect the advantages of process migration over task migration to be even more highlighted when processing performance dominates network throughput more clearly.

Small, simple applications. Little difference results from the migration of short applications as illustrated in Figure 9. When migrated on a process basis, I/O-bound processes (such as gcc) exhibit a 20% degradation in execution time (including migration time), and similarly, 23% when migrated on a task basis. The relatively small overhead generated after migration, such as the delayed transfer of memory, is likely to account for still less when the computation time to migration time ratio increases and when a load-leveling algorithm takes advantage of the uneven load across the nodes to improve the global system throughput. The migration time ranges as above, except for gcc executing a *wait()* system call in the source node PMA, whose completion the process migration mechanism awaits before initiating migration. Even if, in both cases, the process does not perform any useful execution, process migration could benefit here from an aggressive early task migration.

More complex applications with longer execution times. A first application consists of compiling and link-editing tools, a second application consists of text formatting. Migration on a process basis of the first application results in a 3% degradation in execution time, and in nearly 50% when migrated on a task basis. The first application illustrates the attractive notion of migrating a parent process inducing the implicit migration of a whole group of processes: while computing and the process interactions are managed on one node in the first case, task migration only distributes computing, unaware of the semantic links between the processes which also need to be handled by the remote OS personality in addition to the normal OS dependencies. In this sense, process migration is very useful since the semantics linking the processes are confined to the domain of the OS personality. Text formatting shows little difference for single applications with greater execution time however.

Figure 10 also demonstrates the advantage of **data locality exploitation**: a relatively small application is migrated to the node where the files physically reside. The migrated application is even slightly more performant than the unmigrated one. Again, task migration can not exploit fully this advantage showing a slight performance degradation on short-running applications.

Continuous use of the distributed process management

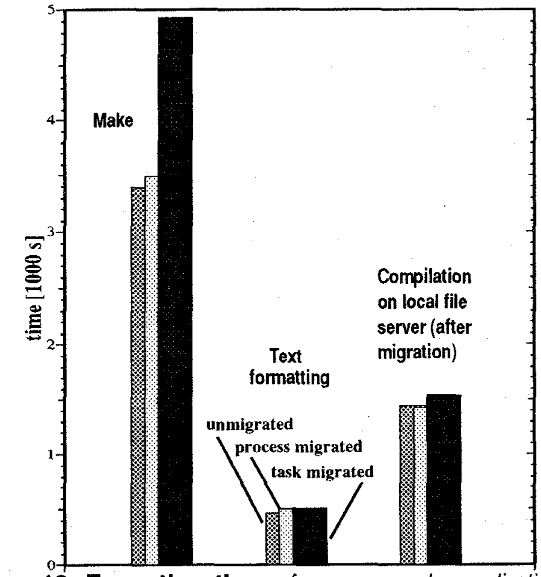

Figure 10. Execution times *for more complex applications: Make, a text formatting and a small Make; migration towards the file server node co-resident with the targeted files.*

facilities revealed the drawbacks of a single PM Master. If we observe the behavior of processes, we find that most processes are short-lived [6] which means that a local PM Master on each node is a very attractive design since most, if not all, interactions take place on the same node. In the absence of a distributed implementation of the PM Master, the cost associated with a single PM Master can be partially overcome by minimizing its involvement in system-call processing and by caching extensively in the PMAs.

The number of semicolons in C code devoted to process migration is approximately 800 out of a total of 5000 for distributing process management, most of the process migration code being devoted to sanity checks and debugging. Similarly, task migration turns in around 400 semicolons of C code. In the current implementation, the copy of the local process state comprises 1120 bytes for the *proc* and equivalent *user* structures, 100 bytes for the *credentials*, 76 bytes for the *rusage* and 9 capabilities.

We export comprehensive information on processes (e.g elapsed time since creation, elapsed time since last migration, migration count or *avencreate* for process creation rate) and we added a new system call, *pcntl()*, to give control over the processes in much the same way as *fcntl()* and *ioctl()* control files and devices. This system call is used mainly to:

- **migrate a process**, taking as arguments a process identifier and the destination node;
- **allow/disallow migration for a given process**, used to specify processes which should not be migrated (*e.g.* servers attached to a device, debugging and instrumentation tools, or newly arrived process);
- **adding a process to a migration process group**, to force the migration of the whole group of processes when one member is migrated (convenient for processes that share data or communicate often with one another).

5. Related Work

Process migration in DEMOS/MP relies on message passing [18]. A process is transparently accessed through established links by referring to triples: *(last-known-machine, creating-machine, local-unique-id)*. The problems arise with the message redirection after several migrations. Our implementation benefits from Mach IPC and *vproc*, which introduce transparency at the Mach and OSF/1 AD level respectively.

The MOS(IX) operating system supports a sophisticated process migration and load balancing scheme [4]. The process migration does not leave any residual dependency; address space is copied eagerly, but incurring higher initial costs.

In Charlotte, migration heavily relies on the underlying communication mechanisms [3]. Migration is fault-tolerant, and processes leave no residual dependency on the source machine. We support similar recovery mechanisms.

Migration in the V kernel promotes a *"pre-copying"* technique for the address space transfer, improving freeze time [21]. Dirty pages referenced during the precopying phase are copied again. In one version of task migration (OMS), we support *pre-copying*, abstracted away the migration of processes and, to a large extent, of tasks.

The Copy-On-Reference (COR) technique is introduced in Accent [24]. It reduces initial migration cost by lazily copying the address space. It is assumed that the program will not access all of its address space, thereby saving the cost of the useless transfers. Our default scheme also relies on COR.

Chorus supports process migration as a basis for load balancing experiments on the hypercube [17]. It is similar to our task and process migration, except that it does not explicitly separate the two layers. Some limitations arise from the fact that Chorus does not support port migration.

Process migration in Locus was one of the first to achieve the product stage with TCF [22]. Its successor TNC [23] is the result of porting some of the TCF functionality to the OSF/1 AD operating system, the same base that we used. TNC is not concerned with task migration issues because the Mach interface is not exposed to the user. Therefore, the atomicity of task/process migration is not affected.

Sprite introduces the idea of a home node that maintains the process state, and the *flushing* technique for address space transfer [7]. Process migration is tightly coupled with the distributed file system. Compared to Sprite, our migration schemes benefit from transparent resource access at Mach and UNIX level, achieving a simpler design. However, it will be hard to compete with Sprite in performance, given its various optimizations through caching on the server and client sides.

Alonso and Kyrimis perform minor modifications to the UNIX kernel in order to support process migration in user space [2]. This implementation is limited to processes that do not communicate and are not dependent on location or process identification. Mandelberg presents yet another user-level process migration scheme for UNIX [12]. This migration scheme does not support tasks that perform I/O on non-NFS files, spawn subprocesses, or utilize pipes and sockets.

Freedman reports a user process-migration scheme which relies on cooperation between the migrated process and the migration module [8]. Little attention is paid to support for files, sockets, and devices; it is not expected that they will be accessed in the middle of execution. In a similar manner Skordos checkpoints and restarts his fluid analysis application, effectively achieving a form of user-space migration [20].

Condor is a software package that supports user-space checkpointing and process migration in locally distributed systems [11]. Condor process migration is dedicated to jobs that do not need transparency and that can accept limitations on the system calls the migrated tasks may issue; there is no support for signals, memory-mapped files, timers, shared libraries, IPC, etc. Performance penalties limit its use to long-running jobs.

6. Conclusions

In this paper we described a process migration design and implementation for the OSF/1 AD operating system. Our migration is based on two levels. We migrate tasks at the Mach level, and then we extend the migration facility for process migration at the OS server level. Then, we presented some performance measurements. Our work demonstrated the feasibility of a relatively easy and straightforward migration design and implementation.

The model of the message-based kernel and the modularity of the OSF/1 AD server (itself divided into more specific servers) have a beneficial impact on the low complexity (no distributed state is involved) and maintenance cost of both migration facilities. We believe that forthcoming versions of operating-system servers would allow even easier and more efficient implementation of process migration.

Furthermore, we have demonstrated reasonable costs. The results given for a homogeneous low system load show the impact of migration for small applications as well as real jobs. Little overhead results from migrating the additional process structure when there is an OS personality on the remote node. Task migration suffers from residual dependencies on the previous OS personality and from being unaware of process relationships, making process-group migration an interesting point to explore further.

Acknowledgments

We would like to thank Chris Peak, Éamonn McManus, Fred Douglis and anonymous reviewers for reviewing this paper and providing many useful suggestions.

References

[1] Accetta, M., et al., "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the Summer USENIX Conference*, Atlanta, GA, 1986, pp 93-112.

[2] Alonso, R., Kyrimis, K., "A Process Migration Implementation for a Unix System", *Proceedings of the USENIX Winter Conference*, February 1988, pp 365-372.

[3] Artsy, Y. Finkel, R., "Designing a Process Migration Facility: The Charlotte Experience", *IEEE Computer*, September 1989, pp 47-56.

[4] Barak, A., Shiloh, A., "A Distributed Load-Balancing Policy for a Multicomputer", *Software-Practice and Experience*, vol. 5, no 9, September 1985, pp 901-913.

[5] Bryant, B., Sears, S., Black, D. and Langerman, A., "An Introduction to Mach 3.0's XMM Subsystem", *OSF RI Operating Systems, Collected Papers*, vol. 2, October 1993.

[6] Cabrera, L., "The Influence of Workload on Load Balancing Strategies", *Proceedings of the Winter USENIX Conference*, June 1986, pp 446-458.

[7] Douglis, F., Ousterhout, J, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *Software-Practice and Experience*, vol. 2, no 8, August 1991, pp 757-785.

[8] Freedman, D., "Experience Building a Process Migration Subsystem for UNIX", *Winter USENIX Conference*, January 1991, pp 349-355.

[9] Hamilton, G., Kougiouris, P., "The Spring Nucleus: A Microkernel for Objects", *Sun Microsystems Laboratories Inc., Technical Report SMLI TR-93-14*, April 1993.

[10] Langerman, A., Black, D., Dominijanni, M., Sears, S., Dean, R. W., Milojicic, D., "NORMA IPC Version Two: Architecture and Design", *OSF RI Operating Systems, Collected Papers*, vol. 3, April 1994.

[11] Litzkow, M., Solomon, M., "Supporting Checkpointing and Process Migration outside the UNIX Kernel", *Proceedings of the USENIX Winter Conference*, San Francisco, January 1992, pp 283-290.

[12] Mandelberg, K. I., Sunderam, V. S., "Process Migration in Unix Networks", *Proceedings of USENIX Winter Conference*, February 1988, pp 357-363.

[13] Milojicic, D., Zint, W., Dangel, A., Giese, P., "Task Migration on the top of the Mach Microkernel", *Proceedings of the third USENIX Mach Symposium*, Santa Fe, New Mexico, April 1993, pp 273-290.

[14] Paindaveine, Y., "Distributed Process Management and Process Migration in OSF/1 AD", *Technical Report 98*, Université catholique de Louvain, August 1994.

[15] Paindaveine, Y., "Process Migration in OSF/1 AD: Problems and Lessons learned", *Technical Report 100*, Université catholique de Louvain, December 1994.

[16] Patience, S., Rabii, F., "The Design of the Process Management Component of OSF/1 AD Version 2", *OSF RI Operating Systems, Collected Papers*, vol. 3, April 1994.

[17] Philippe, L., "Contribution à l'étude et la réalisation d'un système d'exploitation à image unique pour multicalculateur", *Technical Report 308, Ph.D. Thesis*, Université de Franche-comté, 1993.

[18] Powel, P., Miller, B., "Process migration in DEMOS/MP", ACM *Proceedings of the 9th Symposium on Operating Systems Principles*, October 1983, pp 110-119.

[19] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., Neuhauser, W., "CHORUS Distributed Operating Systems", *USENIX Computing Systems*, Fall 1988, vol 1, no 4, pp 305-370.

[20] Skordos, P. A., "Parallel Simulation of Subsonic Fluid Dynamics on a Cluster of Workstations", *MIT A.I. Memo No. 1485*, February, 1995.

[21] Theimer, M., Lantz, K., Cheriton, D., "Preemptable Remote Execution Facilities for the V System", *Proc. of the 10th ACM Symposium on OS Principles*, December 1985, pp 2-12.

[22] Walker, B. J., Mathew, R. M., "Process Migration in AIX's Transparent Computing Facility", *IEEE TCOS Newsletter*, Winter 1989, vol. 3(1), pp 5-7.

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

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