Diploma Thesis

Comparative Evaluation of Process Migration Algorithms

Mathias Noack
Dresden University of Technology
Operating Systems Group

21th July 2003

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Overview 2

1.3 Declaration 3

2 Process Migration **4**

2.1 Characteristics 4

2.1.1 Initial Placement vs. Preemptive Process Migration 5

2.1.2 User-Level vs. Kernel-Level Process Migration 5

2.1.3 Homogeneous vs. Heterogeneous Process Migration 6

2.2 Terminology 6

2.3 Process Migration Algorithms 8

2.3.1 Total Copy 9

2.3.2 Demand Paging 10

2.3.3 Flushing 11

2.3.4 Freeze Free 12

2.3.5 Pre-Copy 13

2.3.6 Queued Pre-Copy 14

2.3.7 Post-Copy 15

2.4 Summary 16

3 Design **18**

3.1 Smile 18

3.2 The Assisted Post-Copy Algorithm 19

3.3 Parallel Execution 21

4 Implementation **22**

4.1 Paging in Linux 22

4.2 Integration into Smile 23

4.2.1 Queued Pre-Copy 24

4.2.2 Post-Copy 25

4.2.3 Assisted Post-Copy 26

5 Evaluation **28**
5.1 Qualitative Evaluation 28
5.2 Test Environment 29
5.3 Quantitative Evaluation 30
5.3.1 `getpid()` Process 32
5.3.2 Random Memory Access Process 34
5.3.3 Reference Process 37
5.4 Comparison 39

6 Related Work **41**
6.1 Accent 41
6.2 RHODOS 41
6.3 Choices 42
6.4 MOSIX 42
6.5 Sprite 43
6.6 Mach 43

7 Conclusion and Future Work **44**

Chapter 1

Introduction

In the past several years, increasing clusters of workstations have been replacing single expensive mainframes as a high-performance facility. In such a distributed computing environment a lot of performance goes unused because machines generally are idle. The utilization of the additional processing power introduces a demand for dynamic allocation and re-allocation of computational resources. In many distributed systems it is possible to transfer data, such as files, between various machines. Some of them additionally allow the movement of the process' execution site, a mechanism known as process migration.

Process migration enables a variety of benefits, such as load balancing, fault tolerance, and data access locality. With the ever-increasing deployment of distributed systems and the World Wide Web, process migration continues to attract both research and product development. Especially commercial services provided over the Internet need to agree on Quality of Service. Therefore more and more distributed operating systems are designed with process migration in mind.

Despite the benefits, process migration has not achieved widespread use in the past because of the complexity of adding transparent migration to systems originally designed to run standalone. Another reason was the additional computation overhead of a migrating process that often limits the benefits of process migration [1].

To make process migration more suitable for practical use, the major goal of research is to optimize the performance of process-migration facilities.

1.1 Motivation

The algorithm that transfers the process from one machine to another has the highest influence on the performance of a process-migration facility. In the last 15 years, many process-migration algorithms have been proposed in the literature, such as Pre-Copy [2], Flushing [3], Post-Copy [4] and Freeze Free [5]. Basically they differ in the amount of transferred data and response time. These algorithms have been implemented on various systems, which makes a

direct comparison difficult to obtain. Thus, it is not known which algorithm performs best under certain conditions such as low network bandwidth.

Most of the distributed systems supporting process migration are academic systems and are already designed with process migration in mind. Usually they provide only one standard algorithm to transfer a process, which is not particulary adaptive in cases of environment changes or different process behaviors, and can cause a performance penalties.

This thesis deals with two major aspects. First, in contrast to other works [6, 3, 7, 8], the process-migration algorithms are implemented on top of the widely used Linux operating system instead of an academic system. Second, the thesis presents an analysis of a wide range of process-migration algorithms. For the first time, five different algorithms execute under the same environment. A comparative evaluation determines under which conditions each algorithm performs best. The most interesting metrics for performance analysis are the duration of the complete migration and the time period during which where the migrating process is suspended.

The results show the influence of the size and memory-access pattern of a process as well as the network characteristic on the performance of various process-migration algorithms. Such an evaluation is interesting for the development of an adaptive process-migration manager that can select the best migration algorithm for different conditions.

Furthermore, to the best of my knowledge, I present within this thesis the first working implementations of the Post-Copy and the Queued Pre-Copy algorithm. Additionally to the existing algorithm, I introduce the new Assisted Post-Copy algorithm.

1.2 Overview

This thesis aims to provide a comprehensive evaluation of process-migration algorithms. Chapter 2 gives a background on process migration in general and describes existing process-migration algorithms. Chapter 3 provides design issues, presents the Smile project, and introduces a new process-migration algorithm. A short description of the implementation is given in Chapter 4. A qualitative evaluation and a quantitative evaluation of the implemented algorithms can be found in Chapter 5. Chapter 6 presents related work about evaluation of process-migration algorithms. Finally, Chapter 7 provides a summary of this thesis and presents a perspective of the future work.

1.3 Declaration

I declare that all parts of this work were autonomously written by me using only legal resources. All resources used within this work are explicitly announced. To the best of my knowledge the content of this work is original and was not published before by me or another author.

Mathias Noack, Dresden, 29th July 2003

Chapter 2

Process Migration

This section gives a background on process-migration characteristics and defines necessary terminology. Furthermore it presents a detailed description of fundamental process-migration algorithms.

2.1 Characteristics

In a distributed environment, process migration introduces opportunities for sharing processing power and other resources between different machines. Therefore, it becomes a useful and desirable feature in distributed systems. To support process migration, the operating system must provide the possibility to extract information about a process on one machine to recreate it on another.

The major goal of process migration is to improve the performance of a distributed system in a number of areas.

Load Balancing The motivation for load balancing is the varying load of individual machines in a distributed system which causes considerable unused processing power. With process migration, the work load can be spread equally across the entire system.

Data Access Locality Sometimes a particular resource cannot be moved (such as a device), or the movement causes heavy network traffic. In this case it is often more efficient to migrate the process to the machine where the resource resides instead of the reverse. This results in reduced network traffic because the process does not have to access the required resource via the network. Thus, it increases the efficiency of the migrated process and resource usage.

Reliability and Availability Process migration can also be utilized to improve reliability and availability of a distributed system. When a shutdown of a machine is known in advance (for example in case of maintenance), all required processes can be migrated to another machine. These processes will continue to be available even after the shutdown of their former source.

Existing implementations of process-migration facilities can be found in MOSIX [9], Sprite [10], RHODOS [11], Accent [12] and Mach [13].

2.1.1 Initial Placement vs. Preemptive Process Migration

A very simple mechanism to achieve system-wide utilization of available resources within a distributed system is *Remote Execution* or *Initial Placement*. Remote execution creates the particular process on a remote machine prior to execution. Sometimes this involves the transfer of code or process environments, such as opened files. Usually remote execution is faster than process migration because it does not transfer potentially large amounts of process information.

Nevertheless, the major disadvantage of remote execution is the lack of flexibility and transparency to the process, which can only be moved at the time of its creation. Otherwise additional information about the behavior of the process can be obtained if the process runs for a period of time on the source machine. This additional information can be used to make more appropriate load balancing decisions.

Preemptive Process Migration dynamically relocates a running process to another machine in a distributed system at an arbitrary time after initiating execution on the source machine. The amount of information that has to be transferred depends on the employed migration algorithm. Usually it is larger compared to remote execution because the information can consist of the entire process environment including the process' address space.

One advantage of preemptive process migration is that after a process has begun execution, the changes of the load of a system can be estimated so that load balancing policies can make more efficient decisions.

Preemptive process migration leads to a better system-wide utilization of available resources compared to remote execution [5]. Therefore this thesis deals solely with Preemptive Process Migration. The term process migration will refer to Preemptive Process Migration throughout this work.

2.1.2 User-Level vs. Kernel-Level Process Migration

Process migration can be applied to different levels of an operating system where it results in varying performance levels, fault resilience, and reusability. Existing implementations of process migration are done at kernel-level or at user-level, including implementations as a part of an application.

User-level process migration typically yields a simpler implementation without changing the underlying operating system. To access kernel information about the process, the user-level process-migration facilities have to cross the user-kernel mode boundary by using kernel requests. These kernel requests are slow and limited, which means not all types of processes

can be migrated. Despite the high migration cost, the implementation levels closer to an application know more about its behavior. This leads to better load balancing policy decisions. An example of a user-level process-migration implementation is Condor [14].

Kernel-level process migration involves modifications and extensions to the underlying operating system kernel which leads to additional complexity. Therefore, deployment is more difficult than with a user-level implementation. On the other hand, the direct and fast access to kernel information about the migrating process results in smoother performance. Another advantage of kernel-level process migration is transparency to the client application, which does not need to be changed or designed with process migration in mind.
MOSIX [15] and Sprite [10] are examples of kernel-level process-migration implementations.

This thesis focuses solely on kernel-level process migration.

2.1.3 Homogeneous vs. Heterogeneous Process Migration

Homogeneous process migration stands for migrating processes in a homogeneous environment of a distributed system. A process can be migrated only among machines with the same compatible architecture and operating system but does not necessarily have the same resources and capabilities. Most systems providing process migration are restricted to homogeneous process migration, including MOSIX [15], RHODOS [11] and Sprite [3].

A single computer network often consists of machines that may vary in their architecture and operating systems. With *heterogeneous process migration* it is possible to migrate a process among such dissimilar machines. Obviously, the implementation of process migration in a heterogeneous environment leads to significantly more complexity and introduces performance penalties due to costly translations. Architecture and operating-system-specific features must be considered. Additionally, the state of a process must be represented in a machine-independent way to be transferred and resumed. Systems implementing heterogeneous process migration are Emerald [16] and TUI [17].

This thesis concentrates solely on process migration between homogeneous machines running the same operating system.

2.2 Terminology

When discussing process-migration characteristics and concepts, varying terminology is used. To avoid potential confusion, this section presents the terminology that will be used throughout this thesis.

The terms *host* and *node* will be used interchangeably to refer to an individual physical machine. The term *source host* will be referred to the execution site of a process prior to migration, with the new site of execution being referred to as *destination host*.

A *process* is defined as an instance of a computer program in execution. The necessary information required for resuming a migrated process is called the *process state*. This state includes execution state, communication state, register set, memory information, and other operating system dependent data. All information including the process state needed for a complete process migration is called *process information*. Principally the process information consists of the process' address space and the process state.

The term *page* refers to a contiguous, fixed-sized group of addresses within the address space of a process.

Residual dependencies occur when process information remains at the source host after migration. Thus, the migrated process is still dependent on its former node. If the source host crashes or the network connection is broken, the execution of the process will fail because of missing information.

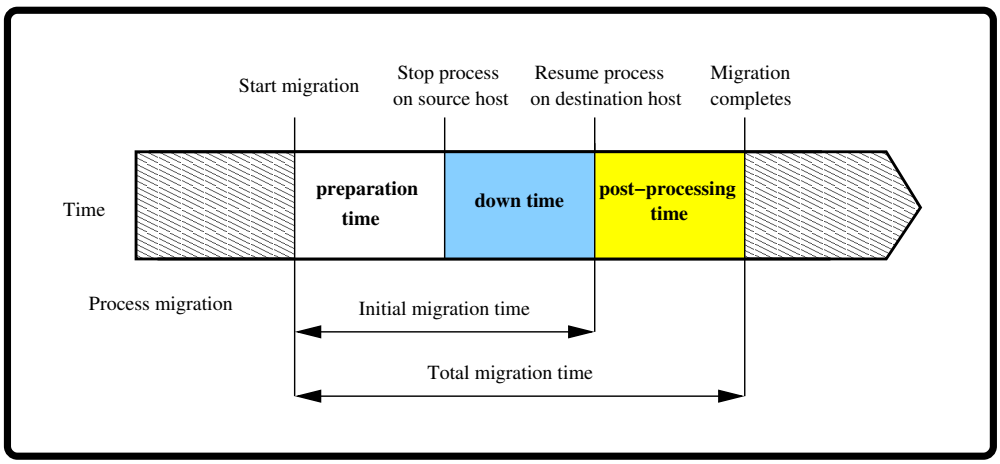

Figure 2.1: Time periods of process migration

There exist several important metrics for evaluating the performance of a process migration algorithm. Figure 2.1 gives a simplified outline of the different time periods occurring during process migration.

The *initial migration time* describes the time passed since the request for migration until the process continues execution on the destination host.

The *total migration time* is the time passed since the request for migration until the end of the process information tranfer. Process migration algorithms with residual dependencies do not have a total migration time because portions of the process information still reside at the former node, so the migration never finishes completely.

The terms *down time* refer to the time period where the migrating process is suspended. Usually during this time the process state is transferred to the destination host depending on the employed migration algorithm. The down time is particularly important to processes that communicate with other processes, because they cannot receive messages during this time.

After a certain time, the communications system detects a failed receiver and either aborts the communication connection or relies on error recovery.

This thesis introduces two more time periods, the *preparation time* and the *post-processing time*. The preparation time refers to the transfer of several parts of the process information, usually the address space, prior to actual migration. The post-processing time is used to eliminate residual dependencies on the source host by shipping all remaining data of the process to the destination host.

2.3 Process Migration Algorithms

This section discusses the basics about the structure of process-migration algorithms and presents a number of existing fundamental algorithms.

To migrate a process, much information must be transferred to the destination host, such as process' address space, execution state, communication state, and other operating-system-dependent information. Usually, the address space constitutes by far the largestunit of the process information and therefore is the element with the highest influence on the performance of process migration [18]. Various transfer strategies have been proposed to reduce the high cost of address space transfer. These strategies dominate process-migration characteristics such as performance, complexity, and fault resilience.

All existing process-migration algorithms have the following basic tasks in common:

- Decide to migrate the process
- Suspend the process at the source host
- Transfer process state to the destination host
- Reconstruct the process state at the destination host
- Resume execution of the process at the destination host
- Remove all remaining information about the process from the source host (not necessarily)

The order and the degree of completion of each task varies between the migration algorithms. The decision of when it is useful to migrate a particular process and which destination host to use are system policy decisions and will not be discussed as they are outside the scope of this thesis. A detailed treatment of these issues can be found in [9].

The major difference between process-migration algorithms is the point in time when the address space of a process is transferred. Either the entire address space is shipped at once or the source host ships parts of the address space on demand. The former method is controlled

by the source host and results in a long initial migration time. In the latter method the destination host requires remote paging to request referenced pages from the source host. This additional request results in a delayed execution of the migrated process.

Most of the fundamental algorithms perform process migration from a source host directly to a destination host. To avoid residual dependencies, a few algorithms involve a third entity such as a file server.

The following sections describe various existing process-migration algorithms.

2.3.1 Total Copy

The Total Copy algorithm is the most commonly used process-migration algorithm. It is very simple and was the first invented. Amoeba [19] and Charlotte [7] are example systems that implemented the Total Copy algorithm.

The basic idea of the algorithm is to suspend the process at the source host, transfer all process information such as the address space, open file information, open network connections, and message channels. After the transfer, the process is immediately resumed at the destination host.

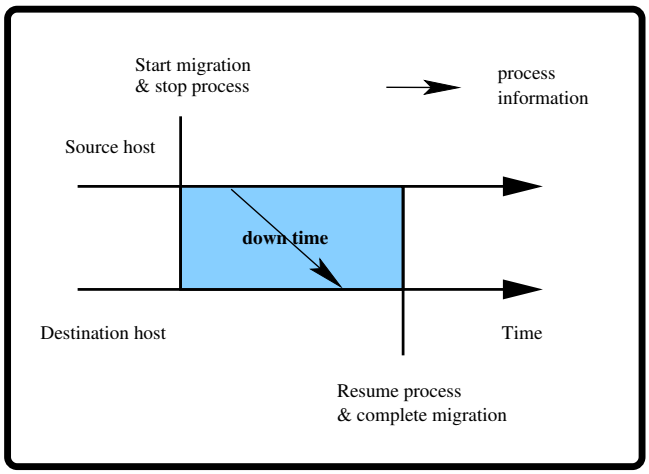

Figure 2.2: Total Copy algorithm

The following tasks must be undertaken to migrate a process with the Total Copy algorithm:

1. Suspend execution of the process at the source host
2. Transfer the entire process information
3. Continue execution of the migrated process at the destination host
4. Remove the process from the source host

The advantage of this algorithm is its conceptual simplicity, which allows a relatively straightforward implementation. Residual dependencies are eliminated by shipping all information at once to the destination host. The algorithm is efficient regarding memory costs because successfully transfered memory can be immediately released.

However, shipping the whole address space during the down time of the migration dramatically increases the appearance of communication failures since the process is unable to receive messages for long time. Figure 2.2 illustrates the Total Copy algorithm.

2.3.2 Demand Paging

The Demand Paging algorithm can only be deployed if there is remote paging support. In contrast to the Total Copy algorithm, there exist different strategies of Demand Paging. The *Copy-on-Reference* strategy transfers only the process state whereas the *Eager Dirty* strategy additionally ships all modified (dirty) pages of the process during the initial migration time. Both request all remaining information on demand from the source host. Figure 2.3 illustrates the different time periods of the Demand Paging algorithm.

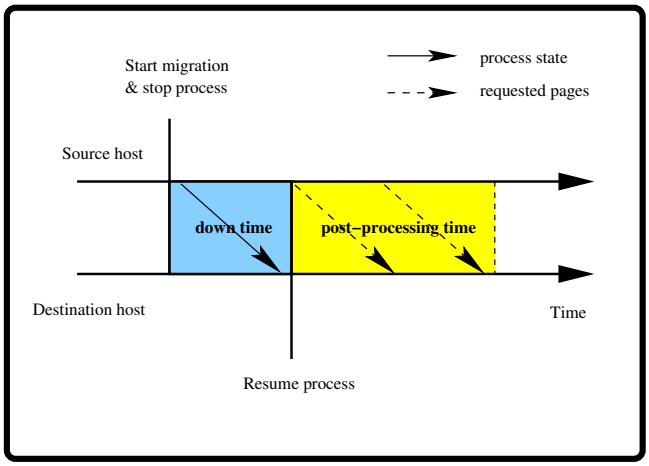

Figure 2.3: Demand Paging algorithm

The following shows a generalized variant of the Demand Paging algorithm:

1. Suspend execution of the process at the source host
2. Transfer of the minimally necessary process state
3. Resume execution of the migrated process at the destination host
4. Request information needed by the process during execution

Once the new process resumes execution, it may reference pages of the address space that still reside at the source host. Since there was no transfer of the entire address space, every

page has to be requested from the source host. To resolve a page fault, the process sends a request to the source host that returns the desired page immediately. During the request the execution of the process is suspended. This delay of the execution is potentially longer than a delay of a locally resolved page fault and increases the run-time cost of the process. The exact duration of the delay depends on the characteristics of the network connection between source and destination host.

The Demand Paging algorithm significantly reduces the amount of data shipped during the migration by avoiding the need to transfer all information of the process. Additionally, the algorithm exploits the fact that processes tend to use only a small portion of their address space during execution [18]. Therefore, it eliminates the transfer of address space pages, which will never be used after migration.

The major disadvantage of the Demand Paging algorithm is that the source host must maintain the remaining address space of the migrated process until it completes execution. These long-term residual dependencies are a problem. If a process migrates multiple times, then a page fault causes a search for the missing page at every host involved in past process migrations.

However, if the source host is to fail, the page fault handler cannot resolve the page faults and the process fails as well. This residual dependency decreases the fault tolerance of the destination host.

The Accent system [18] was the first that implemented the Demand Paging algorithm by using the *Copy-on-Reference* strategy. Further example systems are RHODOS [6] and Mach [13].

2.3.3 Flushing

The Flushing algorithm is the first algorithm that involves a third entity for process migration. It was introduced in the Sprite operating system [10], which uses a file server in addition to the source and destination host. The goal of Sprite' process-migration mechanism was to achieve the efficiency of the Demand Page algorithm while avoiding the residual dependency.

The Flushing algorithm depends upon the operating system' implementation of virtual memory. In the example of Sprite, backing storage for virtual memory is implemented using ordinary files. These backing files are stored by the network file server and are accessible from anywhere in the network. In Figure 2.4, the different time periods are illustrated as well as the role of the file server.

Briefly, the Flushing algorithm is carried out as follows:

1. Stop execution of the process at the source host
2. Flush all modified pages to a network file server

3. Transfer the process state to the destination host
4. Continue execution of the process at the destination host
5. Resolve all page faults raised by the process via requests to the network file server

The Flushing algorithm leaves no residual dependencies at the source host and significantly reduces the down time of a process compared to Total Copy. The data transfer to the file server represents an overhead not present in other migration algorithms.

However, to efficiently handle page faults using a file server, the file system needs to be highly optimized. Sprite provides a file-system-based communication that uses the same access mechanism for memory and files. This enables fast file server access and speeds up the transfer of requested pages to resolve the page faults.

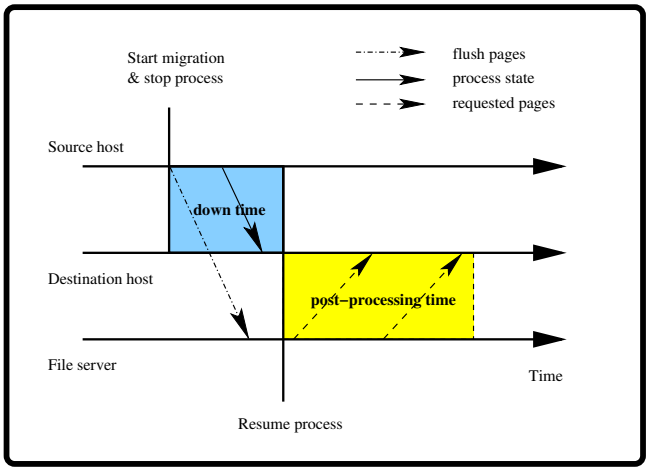

Figure 2.4: Flushing algorithm

2.3.4 Freeze Free

The Freeze Free algorithm [5] represents a highly optimized version of the Eager Dirty strategy mentioned in Section 2.3.2. Most distributed systems support message-based interprocess communication (IPC). The time period where a process is unable to receive messages is called *freeze time*. During this time, the communication subsystem can either buffer incoming messages or rely on error recovery. With all previously described algorithms, interprocess communication is a major problem during the migration because the freeze time is equal to the down time.

The Freeze Free algorithm is carried out as follow:

1. Suspend the process at the source host

2. Separate process state and communication state
3. Transfer a minimized process state including the first stack, code, and heap page
4. Continue execution of the process
5. Transfer the communication state
6. Flush all modified pages to a file server

The major improvement of Freeze Free is minimizing the freeze time so message receiption can proceed in parallel with process migration. The algorithm isolates the process state and the state of the communication from each other. The entire communication state is held in a separate memory region within the address space of the process. Incoming messages are buffered in this particular memory region and all relevant message queue information is recorded; however, the migrating process is not informed of message receipt. After the transfer of the process state, the communication state is shipped separately to the destination host. During this transfer, incoming messages at the source host are rejected and the new location of the process is transmitted to the sender. The new process at the destination host continues execution without waiting for the communication state. However, the process blocks if it tries to communicate before the communication state has arrived.

This procedure effectively eliminates message freeze time because message receipt never stops but is only delayed during the transfer of the communication state.

In contrast to previously described migration algorithms, Freeze Free only transfers the process' current code, stack, and heap page during the down time. The current heap page is determined by using a heuristic at the source host.

The transfer of the address space is done by flushing modified pages to a file server. All page requests from the destination host are satisfied either by the source host if the page has not yet been flushed or by the file server. After completely flushing all pages, even clean pages are available from the file server. However, there exist residual dependencies on the file server, and the cost of requesting a page is potentially higher compared to resolving a page fault locally.

2.3.5 Pre-Copy

The Pre-Copy algorithm was first invented in the V operating system [2] to overcome the disadvantage of high down times in the Total Copy algorithm. In contrast to the previously described algorithms, Pre-Copy does not suspend the migrating process until most of the process' address space is transferred together with the execution of the process on the source host.

Unfortunately during the execution, the process alters pages that are already transferred. Therefore, the algorithm keeps shipping modified pages until the number of these pages is

sufficiently low. Then it suspends the process and starts the transfer of the process state. This transfer must be accompanied by a final flush of the address space to transfer the remaining modified pages. Figure 2.5 displays the time flow of Pre-Copy.

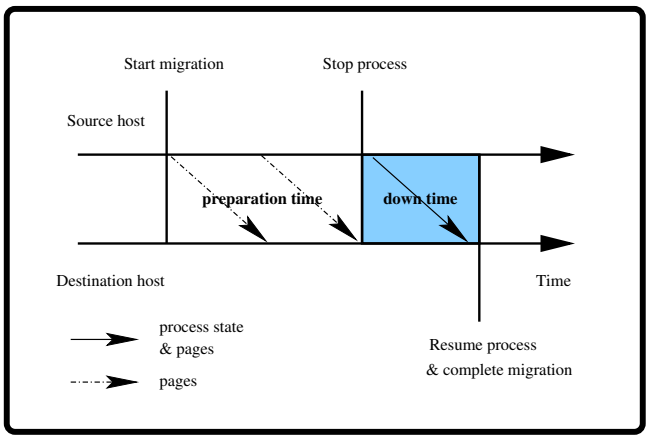

Figure 2.5: Pre-Copy algorithm

The following steps are performed by Pre-Copy:

1. Initially transfer the entire address space together with the execution of the process
2. Check if the number of modified pages is reasonably low; if not, continue transfer of modified pages
3. Suspend process and ship the process state
4. Finally, flush the remaining modified address space
5. Resume execution of the migrated process at the destination host

The Pre-Copy algorithm reduces the pages shipped during the migration of the process. After migration no residual dependencies to the source host exist which leads to higher system reliability. Depending on the exact memory-access pattern of the process, several pages have to be transferred multiple times because they were modified by the process during the preparation time. If the process modifies too many pages, the predetermined limit will never be reached and the algorithm fails. Usually the number of pages affected is the number of pages in the working set of the process, which can be reasonably low [18].

2.3.6 Queued Pre-Copy

The Queued Pre-Copy algorithm is a slightly improved variation of the Pre-Copy algorithm. It reduces the number of multiple page transfers by ensuring that frequently used pages are transferred as rarely as possible.

A short description of the steps carried out during the preparation time is as follows:

1. A set W holds all pages of a process while the process continues execution
2. Each page in W is removed from W, marked read-only in the page table, and queued on a queue R
3. The previous step reruns during a certain time interval until the migration is complete
4. If the process raises a page fault because of a write access to a page marked read-only, the page is removed from R and inserted into W and marked writable
5. A background thread continually dequeues the first page of R and transfers it to the destination host
6. If R is empty, the size of W may be gauged, and if the size is lower than a fixed limit, the migrating process is suspended and all pages in W are queued into R and transferred

Pages which are infrequently written will stay in R longer and will be transferred first. This reduces the number of multiple page transfers, but the algorithm can still fail. As in Pre-Copy, if the predetermined limit is too high, it will never be reached. The algorithm was first described in the Nomadic Operating System [20] but has not been implemented yet.

2.3.7 Post-Copy

The Post-Copy algorithm displays several similarities with the previously described algorithms. Initially Post-Copy transfers the process state. It avoids residual dependencies by transferring the remaining process information after the process has been resumed at the destination host. As in Pre-Copy, the process information is transferred in parallel with the process execution, but differs in that the process executes at the destination host. Figure 2.6 illustrates the different time periods of Post-Copy.

Process migration using Post-Copy proceeds as follows:

1. Suspend the process at the source host
2. Transfer process state to the destination host
3. Continue execution of the migrated process
4. Transfer all remaining process information in parallel with the execution of the process
5. Request pages needed from the executing process of the source host

During the execution of the process at the destination host, the transfer of pages needs to be transparent. As with Demand Paging, the process may reference pages of its address space that still reside on the source host. These references are trapped by the kernel, and a request is sent to the source host. The request takes precedence over normal transfers of remaining pages, and the desired page is sent back immediately. Nevertheless, the request takes some time and the process execution is suspended during this delay. The delay depends on the characteristics of the network connection and how fast the source host can serve the request. Potentially it takes less time to resolve a page fault locally than via a network.

Residual dependencies exist only during the finite post-processing time where the remaining pages are transferred from the source host.

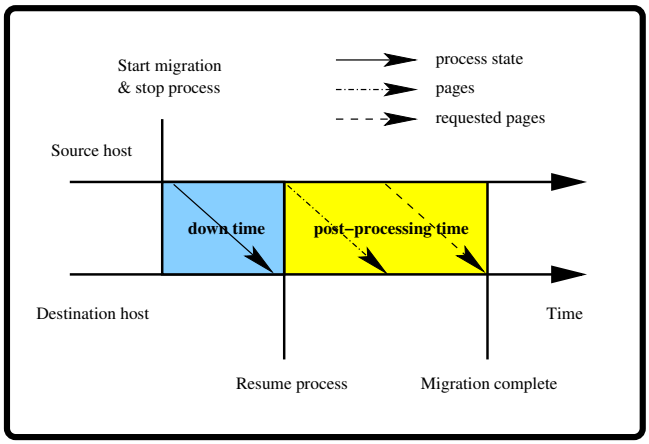

Figure 2.6: Post-Copy algorithm

The Post-Copy algorithm was first described in Post-Copy Migration [21] and has not been fully implemented.

2.4 Summary

Table 2.1 summarizes and categorizes all previously described algorithms. For each algorithm the existence of the preparation and post-processing time are listed below. Additionally, the residual dependencies and systems where the particular algorithm has been realized are shown.

The table shows clearly that all algorithms exploit at most two time periods and some of them need a third entity to avoid residual dependencies at the source host. Furthermore most of the algorithms have been implemented on top of different academic systems, so that a direct comparison is difficult to obtain. Additionally the Queued Pre-Copy and Post-Copy algorithm have only been described in the literature. Prior to my work, there was no working implementation available.

	preparation time	post-processing time	residual dependencies	systems
Total Copy	no	no	no	RHODOS Charlotte
Demand Paging	no	yes	yes on source host	MOSIX RHODOS
Flushing	no	yes	yes on file server	Sprite
Pre-Copy	yes	no	no	V kernel
Queued Pre-Copy	yes	no	no	-
Post-Copy	no	yes	no	-
Freeze Free	no	yes	yes on file server	Choices

Table 2.1: Characteristics of process-migration algorithms

Chapter 3

Design

To implement and evaluate process-migration algorithms, a suitable migration facility has to be found. This chapter presents the Smile project, which provides such a migration facility. Furthermore, it gives a short overview of design decisions and introduces a new process-migration algorithm.

3.1 Smile

The Smile project deals with the realization of service migration for the Linux operating system and was invented by Jan Glauber at the Dresden University of Technology. Service migration means the movement of an arbitrary number of processes that provide a service and communicate among each other. Smile was developed on top of Linux and has the following features:

Transparent migration of unmodified Linux applications Service migration is completely transparent to the user and applications need neither be recompiled nor be relinked.

Complete migration without residual dependencies In case of maintenance, it may be necessary to shut down a host. Thus, Smile supports complete service migration without leaving resources behind. However, during a short post-processing time, residual dependencies are tolerated for the transfer of remaining data from the source host.

Minimal changes to the Linux kernel For process migration it is necessary to alter and possibly create new instances of key data structures of the operating system. Most of these structures are only accessible within the kernel, such as address space information (page table). Therefore, some form of support for process migration from an operating system is required. Linux does not support process migration at all.

Smile realizes kernel-level service migration on top of Linux. Thus, the assistance of the kernel is indispensable. A kernel patch is always a simple but inflexible solution to add new functionality to the Linux kernel.

Alternatively a dynamically loadable kernel module can be used. The advantage is that no recompilation of the kernel is needed and migration support can be added or removed from the kernel at run time. Modules use a well-defined but limited interface to access kernel structures and functions. Still, dynamically adding new features to the Linux kernel by using a kernel module is more desirable for Smile than patching the kernel.

Smile is not restricted to a certain process-migration algorithm and provides a well-defined interface to integrate any migration algorithm. It even allows to migrate a service by using both the preparation and post-processing time.

Because of these features, Smile provides a migration facility that is suitable for implementation and evaluation of process-migration algorithms.

3.2 The Assisted Post-Copy Algorithm

All previously described process-migration algorithms use at most two time periods during migration. To exploit the fact that Smile provides the opportunity to use all three time periods during process migration, I invented the Assisted Post-Copy algorithm. Even after an extensive literature review, I was unable to find any previous description of this algorithm. Therefore I assume the Assisted Post-Copy algorithm has neither been implemented nor described elsewhere.

Basically, Assisted Post-Copy is a synthesis of the Pre-Copy and the Post-Copy algorithm. For the first time a process-migration algorithm uses all three time periods for process migration. Figure 3.1 illustrates the Assisted Post-Copy algorithm.

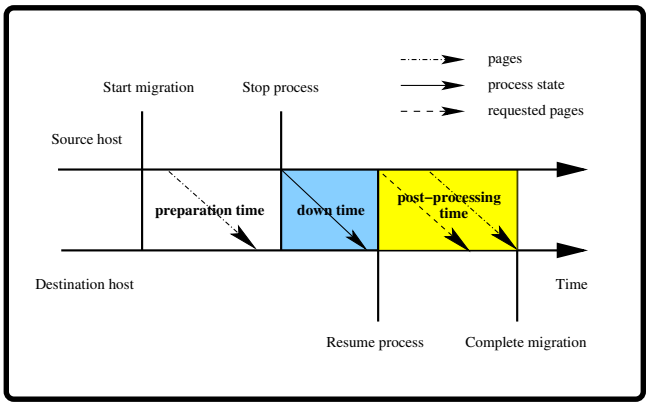

Figure 3.1: Assisted Post-Copy algorithm

Briefly, the following steps are performed by the new algorithm:

1. Transfer the entire address space once while the process keeps executing on the source host

2. Store received pages temporarily at the destination host
3. Suspend the process
4. Transfer the process state
5. Continue execution of the migrated process at the destination host
6. Transfer all modified pages from the source host
7. Request pages immediately needed by the executing process
8. Insert all clean pages locally from the temporary store

As with Pre-Copy, the entire address space of the executing process is initially transferred during the preparation time. The pages are stored temporarily at the destination host instead of being inserted into the new process' address space. After the initial transfer, the source host suspends the migrating process and ships the process state. The migrated process is immediately resumed.

When the process continues execution, it starts to reference pages. Immediately a page fault is raised due to the empty address space. Resolving the page fault by referencing the page locally from the temporary store can cause inconsistency. This is so because it is unsure wether the page remained unmodified during the preparation time at the source host. Therefore a request is sent to the source host, and if the page was modified it needs to be transferred again. Otherwise it can be referenced from the local temporary store. All pages which do not reside in the process' address space need this kind of consistency check.

To avoid that every page needs to be checked, the source host transfers modified pages in parallel with the execution of the migrated process. These pages are consistent and can be inserted in the process address space. When all modified pages reside on the destination host, the consistency check is no longer needed.

Finally, some pages still do not reside in the address space because they remained unmodified on the source host. They need to be shipped from the local temporary store to the process' address space.

The major improvement of this algorithm is the elimination of the predetermined limit needed by Pre-Copy and Queued Pre-Copy. In contrast to both algorithms, Assisted Post-Copy transfers the address space of the process only once during the preparation time.

As in Post-Copy, residual dependencies exist only for the duration of the transfer of modified pages from the source host.

Most of the existing algorithms transfer every page only once, whereas with Pre-Copy and Queued Pre-Copy it is difficult to determine in advance the number of times a page will be transferred. This depends on how often the process accesses a page.

With the new Assisted Post-Copy algorithm, it is now possible to determine the number of multiple page transfers even though it uses the preparation time. Each page is transferred a maximum of two times: once during the initial address space transfer and another time it was modified during the preparation time. This reduced number of multiple page transfers is the best that can be achieved with Pre-Copy-like algorithms.

3.3 Parallel Execution

For the majority of the implemented process-migration algorithms, parallel execution of processes is essential. Unfortunately, the Linux kernel is not completely preemptive. Thus, a running process cannot be preempted while it remains in kernel mode except if it voluntarily relinquishes the control of the CPU. This fact is a big disadvantage for the implemented algorithms because Smile provides kernel-level process migration.

For example, a migrating process has no chance to access and change its address space while the migration facility running in kernel mode is actually transferring the process' address space by using the Pre-Copy algorithm. Therefore the advantages of Pre-Copy cannot be fully exploited and the algorithm behaves similarly to the Total Copy algorithm. The Queued Pre-Copy, Post-Copy and Assisted Post-Copy algorithms are in the same situation. All require real or quasi parallel execution of processes to be efficient. Only the Total Copy algorithm does not need any parallelism because it doesn't perform any parallel activities.

Nevertheless, there is a chance of pseudo parallelism even when the migration takes place on a uniprocessor. In case of a slow network connection, the send queue of the network device can be full or the receive queue at the destination host can be empty. Then the migration algorithm, which runs in parallel with the execution of the migrating process, blocks and releases the CPU. Now the migrating process has a chance to take over control of the CPU. However, this will happen rarely, so it does not improve the efficiency of the migration algorithms at all.

To achieve more realistic results, SMP (Symmetric Multi-Processing) machines are much more suitable for the actual implementation and evaluation of process-migration algorithms. The chance of parallel execution of all participating processes increases significantly and depends only on the SMP scheduler. But still it can happen that both processes run on the same processor. Additionally, assuming to use SMP machines is realistic because in practice process migration is mostly used in a distributed environment consisting of multiprocessor servers.

The mechanism of moving processes between SMP machines using shared memory will not be considered in this thesis.

Chapter 4

Implementation

This chapter presents my implementation of process-migration algorithms on top of Smile. First a short overview of the Linux paging mechanism is provided. Second, it describes in detail the first working implementation of the Queued Pre-Copy, Post-Copy and Assisted Post-Copy algorithms.

4.1 Paging in Linux

To simplify the understanding of my implementation, a background of the paging mechanism in Linux is necessary. Linux provides the concept of *virtual memory* with the *virtual address* as the main ingredient [22]. The paging unit of Linux translates virtual addresses into physical ones by using *page tables*. Every virtual address refers indirectly to a physical address by specifying certain offsets to access the different page table levels. Linux assumes three levels of page tables: the page global directory, the page middle directory and the page table. Figure 4.1 shows the three-level paging model and the format of a virtual address.

The page global directory includes the addresses of several page middle directories, which in turn include the addresses of several page tables. The page table contains *page table entries* that include, along with several status and protection bits, the address of a physical page. To translate a virtual address, Linux traverses the different page tables in the given order until it finds a valid page table entry.

Each platform on which Linux runs must provide translation macros that allow the kernel to traverse the page tables for a particular process. In this way, the architecture-independent part of the kernel does not need to know the format of the page table entries and how they are arranged.

Every Linux process has its own address space, which is subdivided into virtual memory areas (`vm_area`) containing contiguous virtual addresses. Each `vm_area` provides a set of function pointers for opening, closing or manipulating it. One of these is the `no_page()` function

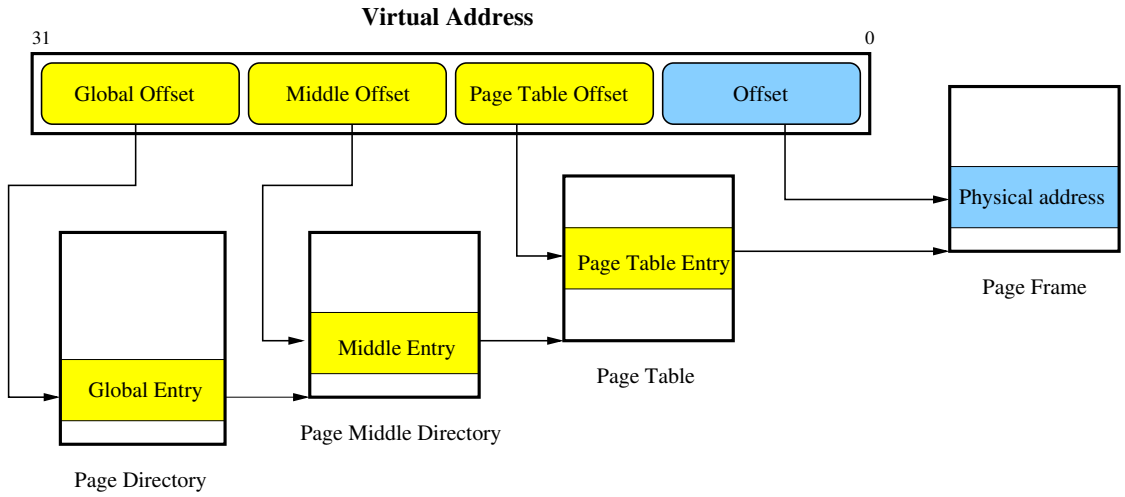

Figure 4.1: The Linux page table organization

pointer. It usually contains a function that is called when a page fault occurs and the page is not present and not swapped out. By replacing this pointer with a pointer to a different function it is possible to change the behavior of the page-fault handler, for example, to request a needed page from a different location.

4.2 Integration into Smile

One part of this thesis deals with the actual implementation of process-migration algorithms and the integration into Smile. The implementation is restricted to algorithms without residual dependencies because Smile does not support residual dependencies.

Within this work I implemented and integrated the following process-migration algorithms into Smile:

1. Total Copy
2. Pre-Copy
3. Queued Pre-Copy
4. Post-Copy
5. Assisted Post-Copy

For further considerations within this thesis, the term *process-migration algorithms* refers to the implemented algorithms only.

The Total Copy and the Pre-Copy algorithm have been previously implemented in various academic systems such as MOSIX [9], RHODOS [6], and Amoeba [19]. Both algorithms are the most commonly used process-migration algorithms without residual dependencies.

As for the Post-Copy algorithm, a first attempt at implementation on top of the Mach distributed system had to be aborted, and a second incomplete one exists on top of the Linux operating system [21]. Therefore, within this thesis, I present the first working implementation of the Post-Copy algorithm.

The Queued Pre-Copy has only been described in the literature [20] and awaits its first actual implementation, which is presented here.

Furthermore, this chapter outlines the implementation of the new Assisted Post-Copy algorithm introduced in Section 3.2.

In contrast to other works, I implemented the process-migration algorithms on top of the widely used Linux operating system instead of on an academic system. For the first time, five different algorithms execute under the same environment and can be compared. This enables an expressive and comprehensive evaluation.

The following sections describe details of the implementation of the Queued Pre-Copy, Post-Copy and Assisted Post-Copy algorithm.

4.2.1 Queued Pre-Copy

Basically, Queued Pre-Copy is a slightly improved variation of the Pre-Copy algorithm. During execution of the migrating process at the source host, the address space is traversed and each page table entry is stored in a dynamic list `sendq`. The virtual address of each page (`addr`) is kept as well for later use.

In contrast to Pre-Copy, only clean, unmodified pages are copied to the destination host during the initial transfer. Modified pages are not transferred and marked as delayed using the `delayed` flag within the `sendq` list. The function `pte_mkclean()` (`linux/include/asm/pgtable.h`) clears the dirty bit in all page table entries (`pte`). Each time a process modifies a clean page, the dirty bit of the page table entry is set by the processor. Therefore it becomes an indicator for recently used pages.

```
initial_transfer(pages);
while ((delayed_pages > CKPT_PRECOPY_MAXPAGES) and (reruns < CKPT_PRECOPY_RERUNS)) {
    if (page is clean and delayed)
        transfer_page(page);
    if (page is dirty) {
        pte_mkclean(pte);
        set_delayed(page);
    }
}
stop_process(process);
```

```
final_flush();
```

After the initial transfer, the number of delayed pages are compared to a predetermined fixed limit `CKPT_PRECOPY_MAXDPAGES`. The algorithm traverses the process' address space again using the `sendq` list if the subset of delayed pages is too large to achieve a reasonably short down time during migration. In contrast to the initial transfer, the algorithm transfers all clean pages only if the `delayed` flag is set. For all modified pages, the dirty bit is cleared and `delayed` is set again. The traversing of the address space continues until the subset of delayed pages is sufficiently low. Then the process is suspended and the process state as well as the remaining address space is finally shipped to the destination host.

As with Pre-Copy, the Queued Pre-Copy algorithm can fail if `CKPT_PRECOPY_MAXDPAGES` is never reached. This can happen if the migrating process modifies too many pages. Therefore, to prevent the algorithm from looping infinitely, a maximal number of reruns is specified in `CKPT_PRECOPY_RERUNS`.

4.2.2 Post-Copy

The Post-Copy algorithm suspends the migrating process and transfers the process state to the destination host. Immediately after the transfer, the process continues execution and starts to reference pages. The process' address space remains empty because no pages have been transferred so far.

The occurring page faults of the executing process are trapped by the kernel and handled by an appendant page-fault handler that invokes a virtual memory area (`vm_area`) specific `no_page()` function. During the down time, in every `vm_area` the `no_page()` function has been replaced by a function that performs the following tasks:

```
spin_lock();
send_virtual_address(addr);
receive_page();
spin_unlock();
```

This particular `no_page()` function sends the virtual address of the needed page to the source host that still holds all pages of the migrated process. A request handler at the source host immediately ships the requested page to the destination host.

An additional *transfer process* ships the remaining address space from the source host in parallel with the execution of the migrated process. For each virtual memory area (`vm_area` of the process' address space, it invokes the `get_user_pages()` function. This function tries

to access every page within the `vm_area`. If a page fault occurs because of a missing page, `get_user_page()` resolves this page fault by invoking the previously described `no_page()` function.

For simplicity of the implementation, a *spin lock* prevents the transfer process as well as the page-fault handler of the migrated process from requesting a page simultaneously. Normally this kind of serialization is done at the source host by protecting the page table of the process from manipulation by two processes. Thus, it does not matter if the serialization happens on the source or destination host. The latter is much easier to realize, so it has been implemented within this work.

The process migration completes and the transfer process exits with the last transferred page. Finally, no process information remains at the source host, so no residual dependencies exist.

4.2.3 Assisted Post-Copy

This section describes my implementation of the Assisted Post-Copy process-migration algorithm. No other previously described implementation could be found in the literature.

During the preparation time Assisted Post-Copy behaves similarly to the Pre-Copy algorithm. After the initially transferring of the entire address space during the preparation time, suspending the process and copying the process state, the execution of the process is resumed at the destination host. Additionally, all pointers to page table entries are kept in a list at the source host for later dirty bit lookup during the post-processing time.

The major problem of Assisted Post-Copy is to determine if a referenced page at the destination host was modified during the preparation time at the source host. To solve this problem, all transferred pages are stored temporarily in a list `dataq` instead of being directly inserted into the process' address space. The page table of the process remains empty after the initial transfer. Thus, every page access results in a page fault during the post-processing time.

The implementation of the post-processing time is similar to previously described Post-Copy implementation. All page faults are handled by the same `no_page()` function that sends requests to the source host. The difference is that the source host transfers only pages which were modified during the preparation time. All other pages have already been transferred and can be fetched from the local `dataq` list. The following pseudo-code example describes the request handler at the source host using the Assisted Post-Copy algorithm.

```
while() {
   wait_for_request();
```

```
    if (page is modified)
        send_page();
    else
        send_page_not_modified();
    }
}
```

The request handler checks if the requested page was modified during the preparation time. If so, the page needs to be transferred again. Otherwise, an short message informs the destination host that the requested page was not modified and can be fetched locally from the `dataq` list. This mechanism avoids multiple page transfer.

The transfer process exits after the last modified page has been received and inserted. All still missing pages are locally transferred into the address space from the `dataq` list and finally, the process migration completes.

Chapter 5

Evaluation

This thesis presents for the first time a comparative evaluation of a wide range of process migration algorithms measured on top of the same environment.

The following chapter discusses performance issues and the actual results of my performance measurements obtained from process-migration algorithm implementations described in Section 4. Subsection 5.1 provides a qualitative evaluation that estimates all potential results of the different time periods performed by the implemented algorithms. A description of the test-bed system and the chosen representative processes to migrate is provided in Subsection 5.2. Furthermore, the Subsection 5.3 presents and discusses the actual measurement results carried out in the described test environment.

The last section summarizes the results of experiments and presents an argumentation.

The major influence of process migration on a process behavior is the suspension of the process and the interruption of any communication with the process. Therefore, the down time of a process is the most important time period to estimate the performance of an algorithm. Along with the down time, another important metric that will be considered is the total migration time.

Besides the time periods, another influence of process migration on a process is delay of execution. In the following sections, I shortly discuss this delay for all evaluated process-migration algorithms.

5.1 Qualitative Evaluation

For the Total Copy algorithm, the down time equals the total migration time. On the one hand, I expect that the down time of Total Copy is the highest of all process-migration algorithms because all process information is transferred during this time period. On the other hand, the total migration time is to be expected as the lowest of all algorithms because information has to be neither requested nor need to be transferred twice. Furthermore, Total Copy uses neither the preparation time nor the post-processing time.

In contrast to Total Copy, the Pre-Copy algorithm exploits the preparation time to transfer information prior to actual process migration. This reduces the amount of data transferred during the down time. Therefore, with Pre-Copy the down time will be significantly shorter compared to Total Copy but certainly higher than using Post-Copy or Assisted Post-Copy. It depends on how many pages have to be finally flushed.

A comparison of the total migration time is difficult to obtain because it is strongly affected by the preparation time and the number of multiple page transfers. If the migrating process modifies a large number of pages, I expect that the total migration time of Pre-Copy will be the highest of all migration algorithms.

Compared to Pre-Copy , the Queued Pre-Copy algorithm reduces the number of multiple page transfers and therefore the total migration time. Furthermore, I expect the down time of Queued Pre-Copy to be slightly lower than the Pre-Copy down time.

I anticipated that the Post-Copy algorithm will have a significantly shorter down time because it ships only a minimum process state during this time period. During the post-processing time both the migrated process and the algorithm request pages from the source host. Thus, the execution of the migrated process is significantly delayed and the total migration time of Post-Copy results in a higher rate compared to Total Copy.

The Assisted Post-Copy algorithm will probably result in the shortest down time because it transfers a further reduced process state to the destination host. The algorithm uses both preparation and post-processing time, whereas each time period is shorter compared to the original algorithms like Pre-Copy and Post-Copy. Furthermore, I expect that the Assisted Post-Copy algorithm will result in the highest total migration time.

This qualitative evaluation is hypothetical only and awaits confirmation from actual measurements.

5.2 Test Environment

This section describes the test environment that I used for the performance measurements.

The distributed test system consisted of two SMP machines, one Dual Athlon with 1800 Mhz, 512 MB RAM and a Dual Pentium III with 450 Mhz, 256 MB RAM. They were connected within a LAN (Local Area Network) via Ethernet and running Linux 2.4.20 as the host operating system. To achieve optimal results, both machines were lightly loaded, so no other processes influenced the evaluation. The *time-stamp counter* provided by processor was used to measure the different time periods during process migration.

The address space of a process constitutes by far the largest part of process information. Thus, the actual transfer of this address space dominates the cost of process migration. For

my measurements I migrated representative processes with different memory-access pattern and sizes of the address space.I chose three different kinds of processes to undergo process migration, each has different properties. Therefore, the results obtained for theses representative processes are characteristic of other processes with similar properties.

`getpid()` Process

The process obtains its process identifier infinitely by using the system call `getpid()`. It represents a class of *light-weight processes* that frequently use system calls and thus enter the kernel mode. Light-weight processes own a small address space and rarely alter it.

Random Memory Access (RMA) Process

The process owns a large address space and randomly modifies pages as fast as possible. Therefore it represents *heavy-weight processes* and a worst-case scenario for migration algorithms, because most processes touch a relatively small portion of their address space on average during their lifetime [18].

Reference Process

A number of researchers have reported on the time taken to migrate a 100 KB (Kilobyte) single-threaded user process [5, 19, 7, 6]. To compare my results with others published in the literature, I measured the time periods during the migration of such a process with a total size of 100 KB. The process frequently modifies all address space pages and does some computation. Because of its size and memory-access pattern, the reference process can still be classified as a light-weight process.

All described representative processes have several requirements in common. During the migration only one process was transferred at the time. The processes access only their address space and do not have any open files or open network connections or performed any kind of IPC (Inter-Process Communication). Each process is transferred without shared libraries or data segments that are read-only, so they can be reopened or remapped at the destination host.

5.3 Quantitative Evaluation

I carried out a series of experiments on the previously described test environment to determine average durations of time periods during process migration. In this section I describe the actual evaluation of the process-migration algorithms listed in Section 4.2. Furthermore I give a detailed analysis of my results obtained for the representative processes described in the previous section.

For the measurements each representative process has been migrated separately with all implemented algorithms. During the migration, the following time periods have been measured:

- Preparation time
- Down time
- Post-processing time
- Total migration time

To show the influences of the network characteristics, I carried out each experiment with a different network bandwidth. For the first measurement I used 10 MB/s, for the second 100 MB/s, and finally for the third 1 GB/s (Fiber) Ethernet. Each process has been migrated 100 times and the results represent the times on average.

The preparation time is determined at the point of time when the information has been sent to the network device to be transferred. The moment when the device is actually sends the information to the destination host depends upon the speed of the network connection. In case of a slow network connection, when the source host suspends the process and starts transferring the process state, the destination host still receives information sent during the preparation time. This *delayed transmission* of the process state causes a longer duration of the down time for all migration algorithms that exploit the preparation time. However, the problem is limited by using fast network connections.

It is difficult to find an optimal value for the predetermined limit needed for the Pre-Copy and Queued Pre-Copy algorithm because it strongly depends on the memory-access pattern of the migrating process. This limit contains the maximum number of pages transferred during the down time, so it primarily determines the duration of the down time. However, I decided to set the limit a maximum 30% of the process' address space.
Additionally both algorithms are allowed to traverse the address space six times to look for modified pages. This fixed number of reruns prevents the algorithms from looping infinitely.

For the representative processes, I expect that the predetermined limit does not significantly influence the performance of the employed algorithm. The reason for my assumption is that the `getpid()` process accesses only a small portion of its address space and therefore the limit is always reached after the initial transfer. In contrast, the RMA process accesses its entire address space within a short period of time and therefore the limit is always violated.

Generally, process migration delays the execution of a migrated process. Obviously, the down time is the major influence on the delay of the exectuion. Another influence is the additional time a process needs to access a page after migration. Normally, frequently used pages reside in a page cache at the source host, but after migration the page cache of the destination host is invalid. Therefore the pages have to be read either from the main memory or backing storage. This delay is caused by all algorithms.

Especially, process-migration algorithm which rely on remote paging additionally delay the process execution, because every page has to be requested from the source host. Therefore, I measured only the delay caused by the Post-Copy and Assisted Post-Copy algorithm.

To migrate a process completely, a minimum amount of data needs to be transferred. This data includes the process information and algorithm-specific message data. The Total Copy algorithm always transfers the minimally necessary amount of data, because information have to be neither requested nor transferred twice. All other algorithm ship additional data.

The following sections present the results of my measurements for each representative process.

5.3.1 `getpid()` Process

To migrate the `getpid()` process completely, Total Copy transfers 62208 bytes. The process' address space is only 57344 bytes, which classifies this process as a light-weight process. All other algorithm nearly transfer the same amount of data, because the number of multiple page transfers is reasonably low. The Assisted Post-Copy algorithm transfers the most data with 63720 bytes.

The costs in Table 5.1 reflect in detail the durations of each time period by using 1 GB/s network bandwidth.

	Preparation Time	Down Time	Post-processing Time	Total Time
Total Copy	0	2.6	0	2.6
Pre-Copy	1.9	1	0	2.9
Queued Pre-Copy	0.9	1.8	0	2.7
Post-Copy	0	0.9	5.7	6.6
Assisted Post-Copy	1.9	0.8	1.4	4.1

Table 5.1: Time periods of `getpid()` Process using 1 GB/s (in ms)

As expected, the down time of Total Copy with 2.6 ms is the highest, whereas for the Assisted Post-Copy algorithm a minimum down time of 0.8 ms was measured. The problem of the delayed receipt of the process state only slightly influences the down time of Assisted Post-Copy because the number of previously transferred pages was reasonably low. Because of the further reduced process state of Assisted Post-Copy, the down time is decreased by around 10% compared to Post-Copy and by 70% compared to Total Copy.

The process occasionally modifies only a tiny portion of its address space. Thus, almost the entire address space can be transferred during the preparation time so that Pre-Copy achieves nearly the down time of Post-Copy. As for Queued Pre-Copy the delayed pages during the

preparation time have to be transferred during the down time and cause the higher duration of 1.8 ms.

Obviously Total Copy has the lowest total migration time because it neither transfers information several times nor needs to request data from the source host like Post-Copy. As for Pre-Copy and Queued Pre-Copy the total migration time differs slightly from Total Copy, but both significantly reduce the down time by transferring the address space prior to the actual migration. Post-Copy results in the highest total migration time because, all pages have to be requested from the source host. Another reason for the high total migration time is the additional computation overhead (see below) caused by the implementation of Post-Copy.

The influence of the predetermined limit used by Pre-Copy and Queued Precopy is very low because the `getpid()` process modifies only few pages. Therefore, the limit is always reached after the initial transfer. The Queued Pre-Copy nearly halves the preparation time compared to Pre-Copy because it does not initially transfers modified pages. These pages are sent during down time and therefore Queued Pre-Copy doubles the down time compared to Pre-Copy.

The diagram in Figure 5.1 illustrates the durations of the time periods for each algorithm by using 10 MB/s, 100 MB/s and 1 GB/s.

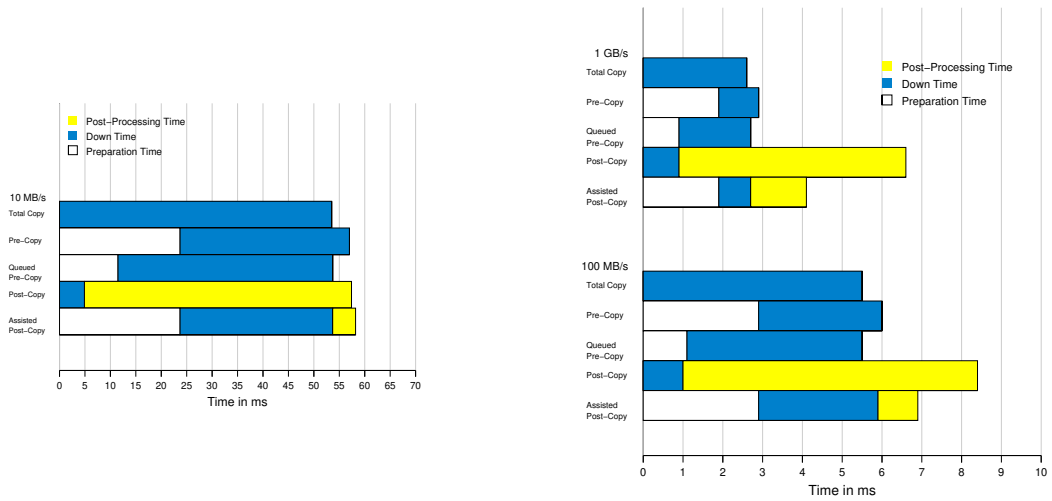

Figure 5.1: Time periods of the `getpid()` process using different network bandwidth

The diagrams clearly show the influence of the delayed transmission on the down time of the Assisted Post-Copy algorithm. For example, the transfer of the process state via 10 MB/s Ethernet takes 4.9 ms, which is the down time of Post-Copy. Normally the down time of Assisted Post-Copy is less than 4.9 ms because of the further reduction of the process

state, but the receipt is delayed by about 25 ms. Additionally this delay extends the total migration time of Assisted Post-Copy. Also the down time of Pre-Copy and Queued Pre-Copy dramatically suffer from this problem.

Another interesting aspect is the post-processing time of the Post-Copy algorithm. With a high-speed network the difference between post-processing time of Post-Copy and total migration time of Total Copy is significantly high. In contrast, with a low-speed network, both times are almost equal. The reason for this difference is the implementation of Post-Copy which adds overhead to the transfer time of a page. For instance, the source host runs through the address space to find and send the correct page, whereas Total Copy sends all pages in a row. This additional overhead is independent from the network speed and is always constant. For example, transferring the whole address space of `getpid()` process using Total Copy with 1 GB/s costs about 2 ms, whereas with Post-Copy it costs about 5 ms. The overhead of 3 ms represents more than half of the total time. In contrast, with a 10 MB/s network only 5% of the total time is actually caused by the overhead. Thus, with a high-speed network the additional time is remarkable.

Besides the delayed transmission, the relations of all durations measured with different network speed are equivalent.

Table 5.2 illustrates the influence of Post-Copy and Assisted Post-Copy algorithm on the `getpid()` process execution. The table displays the additional execution time needed after migration with different network speed.

Network speed	10 MB/s	100 Mb/s	1 GB/s
Post-Copy	7.4	0.9	0.8
Assisted Post-Copy	0.6	0.5	0.4

Table 5.2: Delay of `getpid()` process execution (in ms)

The `getpid()` process needs only a few pages to continue execution. For example, with Post-Copy and a 10 MB/s network it takes 7.4 ms to request all necessary pages. This delay and the down time of Post-Copy nearly represent the total delay of execution.

The Assisted Post-Copy algorithm further reduces the delay compared to Post-Copy. The transfer process is faster than the migrated process to request pages from the source host. Therefore, the migrated process can fetch the pages from the local store.

5.3.2 Random Memory Access Process

This section gives an indication of the cost of migrating the Random Memory Access (RMA) process. The minimum amount of data that has to be transferred is about 4 MB, where the

address space is the dominating element. The process frequently modifies its entire address space, thus it can be classified as a heavy-weight process.

Table 5.3 illustrates the results obtained for the RMA process during migration with a network speed of 1 GB/s.

	Preparation Time	Down Time	Post-processing Time	Total Time
Total Copy	0	82.9	0	82.9
Pre-Copy	365.5	58	0	423.5
Queued Pre-Copy	36.1	56.3	0	92.4
Post-Copy	0	1	390.1	391.1
Assisted Post-Copy	79.1	6.3	435.8	521.2

Table 5.3: Time periods of the RMA process using 1Gb/s (in ms)

Again, the down time of Total Copy dominates with 82.9 ms, but it is now dramatically higher compared to the other algorithms. Remarkable is the reduction of the down time of the Post-Copy algorithm by about 98% compared to Total Copy, because of the constant size of the process state. The Assisted Post-Copy algorithm can reduce the down time down to 6.3 ms, whereas the delay of the transmission is about 5.3 ms. Assisted Post-Copy transfers nearly twice as much data compared to Total-Copy.

The down times of Pre-Copy and Queued Pre-Copy are now significantly higher compared to Post-Copy. Because more than a few pages have to be shipped during the down time. The preparation time of Pre-Copy shows that the algorithm failed to reach the predetermined limit. Queued Pre-Copy fails the limit as well, but it rarely transfers pages during the preparation time. All modified pages are delayed until the maximum number of reruns is reached.

Because of the multiple page transfer, Pre-Copy ships about 7 times more data than Total-Copy. In contrast, Queued Pre-Copy does not transfer significantly more data because it avoids multiple page transfers.

The results in Table 5.3 clearly demonstrate the advantage of Queued Pre-Copy over Pre-Copy. Instead of transferring the whole address space, Queued Pre-Copy delays the transfer of all modified pages. This delay causes a lower preparation time and total migration time. Additionally, traversing the address space without shipping modified pages is much faster, so the process has less time to alter its address space. Consequently, the number of pages that have to be transferred during the down time are decreased, so the down time is reduced as well. However, Queued Pre-Copy has a preparation time that is 10 times lower compared to Pre-Copy, but has nearly the same down time with 56.3 ms.

Noticeably, the Post-Copy is 5 times slower to complete the process migration than Total Copy because of the same additional overhead described in Section 5.3.1. Both the execution time of the process and the post-processing time are delayed.

The diagram in Figure 5.2 shows a comparison of all process-migration algorithms migrating the RMA process with different network bandwidths.

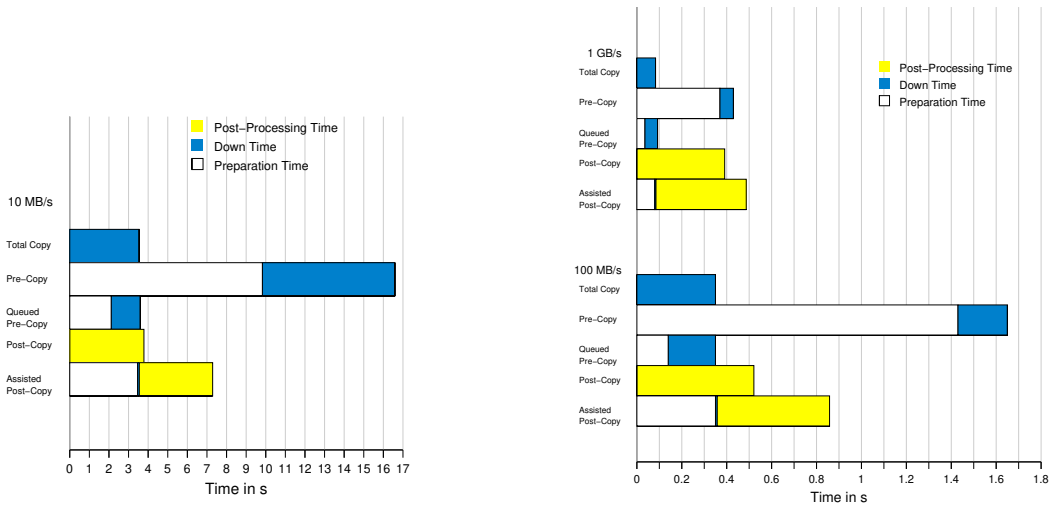

Figure 5.2: Time periods of the RMA process using different network bandwidths

The results of the lower bandwidth confirm the predicted disadvantages of Pre-Copy. Total Copy needs only 20% of the time of Pre-Copy to completely migrate the particular process. Because of the slow transfer during the preparation time, the process has enough time to alter so many pages that the predetermined limit cannot be reached. Queued Pre-Copy minimizes this problem because it does not transfer the modified pages during the preparation time. Therefore with 100 MB/s, Queued Pre-Copy reduces the preparation time by about 90% compared to Pre-Copy and has almost the same total migration time as Total Copy.

Further investigation of the predetermined limit show that it has no influence to the performance of the Pre-Copy algorithm at all. During the time the entire address space is transferred, the migrating process alters it completely.

With a lower bandwidth, the post-processing time of Assisted Post-Copy is always below the post-processing time of Post-Copy because of the reduced amount of pages that have to be transferred. In contrast to that, Assisted Post-Copy needs about 440 ms to ship just a portion of the address space with 1GB/s, whereas Post-Copy transfers nearly the entire address space in about 390 ms. The reason for this additional time is the computation overhead that

the implementation of Assisted Post-Copy adds to the post-processing time. Similar to the computation overhead of Post-Copy compared to Total Copy (see Section 5.3.1) it is constant and only noticeable with a high network bandwidth.

This problem does not occur with light-weight processes because of the small number of modified pages transferred during the post-processing time with Assisted Post-Copy.

Table 5.4 illustrates the influence of Post-Copy and Assisted Post-Copy algortihm on the RMA process execution. The table displays the additional execution time needed after migration with different network speed.

Network speed	10 MB/s	100 Mb/s	1 GB/s
Post-Copy	3700	500	400
Assisted Post-Copy	2800	400	300

Table 5.4: Delay of RMA process execution (in ms)

The delays of the RMA process execution are more remarkable compared to the `getpid()` process. To completely migrate the RMA process, at least 1000 pages need to be transferred. The process itself requests half of these pages, whereas the other half is transferred by the particular transfer process. For example, the execution of the migrated process is delayed about 3.7 s while using Post-Copy and 10 MB/s network speed. This time is needed to request approximately 500 pages from the source host.

Assisted Post-Copy can reduce the delay with lower bandwidth because most of the pages are already reside locally. This advantage does not occur with higher network speed, because the difference of the delay between Post-Copy and Assisted Post-Copy is minimal.

However, Post-Copy and Assisted Post-Copy achieve the lowest down time and a reasonable total migration time even with a reduced network speed.

5.3.3 Reference Process

A number of published works about performance measurements of process-migration systems report the down time while migrating a 100 KB process as a reference [5, 19, 7, 6]. To be comparable with other systems, I present in this section my results obtained from the migration of a 100 KB process.

The memory-access pattern of such a reference process is not further described in the literature. Therefore, my Reference Process accesses the address space frequently and does some computation.

First, Table 5.5 provides the results of migrating a 100 KB process in various process-migration system. Additionally, the table lists information about the year the results were published, the test environment, and the employed algorithms.

Operating System	Year	Platform	Network Mb/s	Down Time in ms	Algorithm
V kernel	1985	Sun 10MHz 68010	Ethernet 10	680	Pre-Copy
Accent	1987	Perq workstation	Ethernet 10	13,000	Demand Paging
Charlotte	1989	VAX 11/750	Pronet	750	Total Copy
Sprite	1991	SPARCstation1	Ethernet 10	330	Flushing
Mach OMS	1993	Intel80486 33MHz	Ethernet 10	250	Demand Paging
Choices	1995	SPARCstation2	Ethernet 10	14	Freeze Free
RHODOS	1997	SUN 3/50	Ethernet 10	118 351	Demand Paging Total Copy
Linux	2003	Dual Pentium 450 / Dual Athlon 1800	Ethernet 10	89 33 5	Total Copy Assisted Post-Copy Post-Copy

Table 5.5: Process Migration Systems and Down Time

To obtain comparable results, I used a 10 MB/s Ethernet network. The last line of Table 5.5 presents the down time measured during Total Copy, Assisted Post-Copy and Post-Copy.

One reason for the shorter down time is the better hardware of the test environment, which can quickly process the incoming information. Besides, only the Post-Copy algorithm with 5 ms down time is faster than the highly optimized Freeze Free algorithm with 14 ms. Additionally, the latter transfers the first stack, code and heuristically determined heap page (each 4 KB page size) during the down time. If I add these pages to Post-Copy with a transfer time of 2 ms per page, the algorithm achieves a down time of 11 ms, which is still better than Freeze Free.

Most of the process-migration systems employ the Demand Paging algorithm. As a tarde-off for a minimal down time, residual dependencies are tolerated. As I expected the algorithms implemented on top of Linux achieve better performance results.

For completeness, Table 5.6 illustrates the results of migrating the reference process with 10 MB/s network bandwidth in detail.

	Preparation Time	Down Time	Post-processing Time	Total Time
Total Copy	0	88.5	0	88.5
Pre-Copy	52.4	36.5	0	88.9
Queued Pre-Copy	37.1	51.4	0	88.5
Post-Copy	0	4.9	93.4	98.3
Assisted Post-Copy	52.4	33.4	3.5	89.3

Table 5.6: Time periods of the Reference Process using 10 MBb/s (in ms)

The results show the same proportions and characteristics as the results obtained for the `getpid()` process, even with a different memory-access pattern.

5.4 Comparison

In this section I summarize the results obtained from the measurements of the representative processes. Furthermore I generally compare the most important metrics, the down time and the total migration time of each algorithm. The following two tables illustrate the efficiency of the implemented migration algorithms under different conditions. For comparison, the down time is always preferred to the total migration time.

The Table 5.7 rates the efficiency of each algorithm to migrate a light-weight process with either high-speed or low-speed network connection.

Network speed	Low Speed		High Speed	
	Down Time	Total Migration Time	Down Time	Total Migration Time
Total Copy	- -	++	- -	++
Pre-Copy	o	+	++	+
Queued Pre-Copy	-	++	o	++
Post-Copy	++	+	++	- -
Assisted Post-Copy	-	o	++	o

++ very good, + good, o moderate, - bad, - - worst

Table 5.7: Algorithm efficiency while migrating a light-weight process

As presented in Table 5.7, the Post-Copy algorithms offer the best properties for migrating a light-weight process with low network speed. As always the down time is minimal and the total migration time is reasonably low. Total Copy and Queued Pre-Copy suffer from an unacceptably high down time but provide a sufficiently low total time, whereas Pre-Copy performs moderately. The worst algorithm for low network speed is the Assisted Post-Copy algorithm because of the longest total migration time and only a moderate down time.

High-speed network connections allow a completely different argumentation. Here the Pre-Copy algorithm achieved the best results followed by the Assisted Post-Copy and Queued Pre-Copy algorithm. The total migration time of Post-Copy suffers from the previously described overhead problem (see Section 5.3.1) and the down time of Total Copy is highest of all algorithms.

The efficiency of each migration algorithm while migrating a heavy-weight process is shown in Table 5.8. Again, different kinds of network speeds are used.

To migrate a heavy-weight process with low network speed, the Post-Copy algorithm is again the best choice. Another option is Assisted Post-Copy, which needs additional time

Network speed	Low Speed		High Speed	
	Down Time	Total Migration Time	Down Time	Total Migration Time
Total Copy	-	++	- -	++
Pre-Copy	- -	- -	-	-
Queued Pre-Copy	o	++	-	+
Post-Copy	++	+	++	-
Assisted Post-Copy	+	o	+	- -

++ very good, + good, o moderate, - bad, - - worst

Table 5.8: Algorithm efficiency while migrating a heavy-weight process

to complete the migration. Also Queued Pre-Copy performs acceptably with low network speed. The Pre-Copy algorithm is absolutely inefficient with the chosen predetermined limit. It always failed to reach the limit and that causes dramatically high costs.

The Post-Copy algorithm achieves the best result for a migration of a heavy-weight process with a high-speed network. The down time is the lowest but the total migration time is much higher compared to Total Copy. Assisted Post-Copy also obtains a acceptable down times, but the total migration times suffer again from the implementation overheads. The Pre-Copy algorithm fails again even with higher network speed. That shows that the predetermined limit depends strongly on the process' memory-access pattern, which has to be determined individually for each process. But the behavior of a process is rarely predictable. Also the Total Copy algorithm is ineligible for migrating heavy-weight processes because of the high down time.

The measurements have shown the influence of the algorithm on the execution time of the migrated process. Depending on the network speed the delay can be up to 3.7 s. With Assisted Post-Copy, the delay is lower than with Post-Copy.

Generally with low network speed, Post-Copy is always a good choice, and with high network speed Queued Pre-Copy performs best. Furthermore, the measurements show that the *new* process-migration algorithms like Queued Pre-Copy, Post-Copy and Assisted Post-Copy achieve better average performance than the fundamental algorithms such as Total Copy and Pre-Copy.

Chapter 6

Related Work

Several performance measurements of process-migration implementations on top of various systems have been published in the literature. This section presents related work for the evaluation and assessment of process-migration algorithms.

6.1 Accent

Important efforts regarding process migration and address space transfer can be found in [18]. The Demand Paging algorithm was introduced and implemented first on top of the Accent [12] distributed environment. The work in [12] presents a detailed analysis of the memory-access pattern of a process and compares precisely the Total Copy and the Demand Paging algorithm (including eager dirty and copy-on-reference strategies).

The results published in this work show that the number of bytes exchanged between nodes by using Demand Paging drops by an average of 58%. This demonstrates the effectiveness of Demand Paging, but it hides the problem of residual dependencies.

Furthermore it confirms the assumption that processes access a relatively small part of their address space.

6.2 RHODOS

RHODOS [11] is an experimental distributed operating system developed at the Deakin University, Australia. It consists of a microkernel and provides message passing. The process-migration facility of RHODOS has been designed to utilize different migration strategies and hence to allow their comparison.

The logical design of the RHODOS multiple strategy process-migration manager has been proposed in [23]. This facility allows to determine which migration algorithm can be used to achieve the best performance under certain conditions.

A performance comparison between a Demand Paging algorithm (copy-on-reference strategy) and a Total Copy algorithm on top of RHODOS is published in [6]. The only performance metric that has been measured and evaluated was the down time.

With Total Copy, the down time was three time higher than using Demand Paging on the same process. The measured results show clearly the low initial cost of process migration using Demand Paging but in parallel, the run-time cost increases dramatically. However, a detailed measurement of the different time periods performed by process migration was not described.

6.3 Choices

Choices [8] is the host operating system for the implementation of the previously described Freeze Free process-migration algorithm. The work in [5] describes the design and implementation of the Freeze Free algorithm. Furthermore some attempts towards normalizations of the performance of process-migration algorithms were done in this work to allow an evaluation of different migration strategies. A direct comparison was dropped from further coverage in [5] because of the amount of work to implement different process migration algorithms on top of Choices.

The File Server (or Flushing) algorithm implemented in Sprite and the Demand Paging algorithm implemented in Accent are compared against the Freeze Free algorithm. Because of the varying processor speeds, the normalization of the performance was done by analyzing every step of each algorithm performed at the same speed as the implementation of the Freeze Free algorithm.

The results confirm the advantages of the Freeze Free algorithm against other algorithms with residual dependencies, even to a file server. Nevertheless, normalizing performance of process-migration algorithms is often hard and inaccurate.

6.4 MOSIX

MOSIX [15] presents the widely-used load balancing system. The distributed operating system was invented at the Hebrew University of Jerusalem and provides a preemptive process-migration facility. It uses decentralized algorithms for automatic work distribution, load balancing and memory ushering.

The employed process-migration algorithm in MOSIX is the eager dirty strategy of Demand Paging. Only dirty pages and the user area of the migrating process are transferred at the time of migration. All other information is faulted in as needed once the process resumes execution at the destination host.

6.5 Sprite

Sprite [10] was developed at the University of California in Berkley. It provides a network-wide single system image with specialized file service, remote file access and transparent process migration.

Sprite was the first system that overcame the problem of residual dependencies to the source host by employing the Flushing algorithm. When a process is migrated, it is frozen at the source host and all memory from its address space is flushed to a so-called page file on disk. The process state is transferred as usual and every page can be requested from the page file. The Sprite network file system ensures that in most cases the page does not cause disk operations because the file server uses its memory as a cache for the page file. The Flushing algorithm is optimized for the Sprite network file system and results the measurements can be found in [3].

Unfortunately, no other migration algorithms have been implemented on top of Sprite and therefore no evaluation is available.

6.6 Mach

The Mach [13] microkernel was developed at the Carnegie Mellon University, and the migration mechanism on top of Mach was invented at the University of Kaiserslautern.

Mach implements two kinds of migration facilities that provide different process-migration algorithms. The Simple Migration Server (SMS) is a very robust migration facility and was invented first. It only implements the Demand Paging (copy-on-reference strategy) for process migration.
The Optimized Migration Server (OMS) provides user-level process migration and supports the Demand Paging (copy-on-reference strategy), Total Copy and Pre-Copy algorithm.

Unfortunately no detailed process-migration times or evaluations of the different strategies have been reported.

Chapter 7

Conclusion and Future Work

Within this thesis, I present for the first time a comprehensive comparable evaluation of a wide range of process-migration algorithms including Total Copy, Pre-Copy, Queued Pre-Copy, Post-Copy and Assisted Post-Copy. Previously, a quantitative evaluation of process-migration algorithms was difficult to obtain because the implementations were done on different architectures and vary in their hardware, network and process environment.

To enable a direct comparison, I implemented the migration algorithms on top of the widely used Linux operating system in contrast to previous work which used academic systems only. For the first time, a considerable number of process-migration algorithms have been implemented and evaluated on the same system.

Furthermore, to the best of my knowledge the Queued Pre-Copy and Post-Copy algorithm presented within this work have not been previously implemented, and therefore, I provide the first working implementation. In addition to the existing algorithms, I introduced the new Assisted Post-Copy algorithm, which is the only one that exploits both the preparation time and the post-processing time during migration.

For comparison, I measured the algorithms under the different conditions such as varying memory-access pattern and network speed. The results of my measurements demonstrate the advantages and disadvantages of each algorithm under different conditions. Mostly, the newly invented algorithms like Post-Copy, Queued Pre-Copy and Assisted Post-Copy outperform the existing algorithms such as Total Copy and Pre-Copy.

In the future, the introduced algorithms need to be optimized. The delayed transmission presented in Section 5.3.1 is one problem that demands further considerations. Furthermore, the computation overhead of Post-Copy and Assisted Post-Copy presented in Section 5.3.1 can be reduced by optimizing the implementation of these algorithms. Another interesting challenge is the employment of algorithms with residual dependencies, such as the Freeze Free and Demand Paging, for additional comparison. The delay of process execution also demands further investigation and more detailed measurements are needed.

The presented investigation opens many avenues for future research. The creation of an adaptive process-migration manager is now conceivable. This work delivers the criterion for the decision policy for a manager that allows to choose the best algorithm to migrate a process under certain conditions. With an employed adaptive process-migration manager, the performance of migration facilities can be significantly improved.

Bibliography

[1] Orly Kremien and Jeff Kramer. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[2] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[3] Fred Douglis. Experience with Process Migration in Sprite. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 59–72, Berkeley, CA, 1989. USENIX Association.

[4] Michael Richmond and Michael Hitchens. A new process migration algorithm. *ACM SIGOPS Operating Systems Review*, 31(1):31–42, 1997.

[5] Ellard Thomas Roush. *The freeze free algorithm for process migration.* PhD thesis, University of Illinois, 1995.

[6] Damien De Paoli. Copy on Reference Process Migration in RHODOS, November 1997.

[7] Yeshayahu Artsy and Raphael A. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, 22(9):47–56, 1989.

[8] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and Implementing Choices:an Object-Oriented System in C++. *Communications of the ACM*, September 1993.

[9] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System.* Springer Verlag, 1993.

[10] Fred Douglis and John K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[11] A. Goscinski. Research of Distributed and Open Systems and Processing: The RHODOS Project. Technical report, Deakin University, Australia, 1994.

[12] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th symposium on Operating systems principles*, pages 64–75, 1981.

[13] W. Milojicic, W. Zint, and A. Dangel. Task Migration on Top of the Mach Microkernel - Design and Implementation. Technical report, University of Kaiserslautern, 1992.

[14] M. Lizkow, M. Livny, and T. Tannenbaum. Checkpoint and Migration of UNIX Processes in the Condor Distributed Environment, April 1997.

[15] Amnon Barak, Oren Laadan, and Yuval Yarom, editors. *The NOW MOSIX and its Preemptive Process Migration Scheme*. TIEEE TCOS, 1995.

[16] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[17] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The Tui system. *Software Practice and Experience*, 28(6):611–639, 1998.

[18] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 13–24. ACM Press, 1987.

[19] Chris Steketee, Weiping Zhu, and Philip Moseley. Implementation of Process Migration in Amoeba. In *International Conference on Distributed Computing Systems*, pages 194–201, 1994.

[20] Asger Henriksen and Jacob Gorm Hansen. Nomadic Operating Systems. Master's thesis, University of Copenhagen, December 2002.

[21] Michael A. Richmond. Post-Copy Migration: A new process migration algorithm. Master's thesis, University of Sydney, November 1996.

[22] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel. From I/ O Ports to Process Management.* O'Reilly & Associates, Inc., 2003.

[23] Damien De Paoli. The Multiple Strategy Process Migration Manager for RHODOS: The Logical Design, 1993.