THE FREEZE FREE ALGORITHM FOR PROCESS MIGRATION

BY

ELLARD THOMAS ROUSH

B.S., University of Washington, 1976
M.S., Stanford University, 1977
M.B.A., Campbell University, 1980

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois
THE FREEZE FREE ALGORITHM FOR PROCESS MIGRATION

Ellard Thomas Roush, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1995
Roy H. Campbell, Advisor

Dynamic process migration moves a running process to a new machine, and supports both load sharing and processor fault tolerance. This thesis introduces the Freeze Free process migration algorithm, which uses the following six techniques to dramatically reduce the overhead and complexity of dynamic process migration.

First, existing systems use request and response messages to initiate process migrations and transfer some elements of the process state. The Freeze Free algorithm eliminates all request and response messages from the process migration latency period. The combined process control and execution state message implicitly signals the start of a process migration. The current stack page message implicitly tells the new host to resume execution. The old host blasts the combined process control and execution state, the current code page, the current heap page, and the current stack page to the new host without delay during the latency period. This information can not be further reduced and support process migration for a broad class of processes.

Second, the Freeze Free algorithm delivers the first critical pages without page faults. The program counter identifies the current code page, and the stack pointer identifies the current stack page. A heuristic identifies the current heap page by examining the instruction stream. The system truncates the top stack page to the portion currently in use.

Third, the Freeze Free design separates process control and communication state, which allows process migration and message receipt to proceed in parallel. The new design effectively eliminates the message freeze time, which plagued prior systems.

Fourth, the Freeze Free design separates process control and file state, which allows the process to resume execution on the new host, while the system flushes data to the file server.

Fifth, the Freeze Free algorithm preallocates and partially initializes a set of data structures for use at process migration time, which moves expensive operations out of the critical path.

Sixth, the Freeze Free design reorganizes data structures so that information about an object appears only within that same object. This drastically reduces the cost of extracting and inserting state.

The net result of these techniques is a reduction in the process migration latency time by an order of magnitude, while simultaneously supporting processor fault tolerance and effectively eliminating message freeze times. Furthermore the latency cost does not change with process size. The latency time is 13.9ms on a 4kB page system, 20.8ms
on an 8kB page system, and 36.9ms on a 16kB system. By comparison, Sprite [DO91] dynamic process migration takes 330ms.

This thesis shows that process migration latency costs are now a small fraction of the demand page operations across the network. The analysis also reveals further potentially large savings in both process migration latency and cross network demand paging.

The thesis demonstrates the negative impact of increasing overhead on system load sharing speedup through experiments with different overhead costs. Small overhead is essential for good speedup. The combined process migration latency operations and cross network demand pages add an overhead that is 4.5% of the test process execution time.
Acknowledgements

Professor Roy Campbell asked me what I could do that was better than what had been done earlier. This thesis answers that question. I extend my appreciation for his support of my research in an unconventional area. I thank Professor Laxmikant Kale for serving on both my prelim and thesis committees. I thank Professor Michael Loui for serving on my prelim committee. I thank Professor Michael Faiman for serving on my thesis committee and helping me to avoid a major scheduling problem. All of the members of the extended Choices group have provided useful support and exchanged ideas: Willy Liao, See-Mong Tan, Sefika Mohlalefi, Nursalim Hadi, Amitabh Dave, David Putzolu, Lun Xiao, Aamod Sane, Chris Bachman, Tin Qian, Zhaoyu Liu, Zhigang Chen, Ashish Singhai, and David Raila. I also thank Anda Harney and Bonnie Howard for administrative support.
# Table of Contents

## Chapter

1 **Introduction** ................................................. 1  
   1.1 Process Migration Problem Scope .......................... 2  
   1.2 Thesis Statement ......................................... 5  
   1.3 Thesis Outline ............................................ 7  

2 **Background** .................................................. 9  
   2.1 System Availability ........................................ 10  
   2.2 System Model ............................................. 11  
      2.2.1 Autonomous System .................................. 11  
      2.2.2 Integrated System ................................... 12  
      2.2.3 Massively Parallel Processor System ............... 13  
      2.2.4 Summary ............................................. 14  
   2.3 Load Indicators .......................................... 14  
      2.3.1 Processor Load Measures ............................ 15  
      2.3.2 Obtaining Load Measures ............................ 15  
   2.4 Static vs. Dynamic Load Balancing ....................... 16  
   2.5 Immobility ............................................... 16  
   2.6 Time Constraints ......................................... 17  
      2.6.1 Task Characteristics ................................ 17  
      2.6.2 System Loading ..................................... 19  
      2.6.3 Migration Delay ..................................... 19  
   2.7 Load Balancing Strategies ................................ 20  
   2.8 Summary .................................................. 20  

3 **Related Work** ............................................... 22  
   3.1 Programming Language Based Systems ..................... 23  
   3.2 User Level Process Migration ............................. 23  
      3.2.1 rsh .................................................. 23  
      3.2.2 Zhou and Ferrari .................................... 24  
      3.2.3 Butler ............................................... 24  
      3.2.4 Remote Execution Monitor ........................... 24  
      3.2.5 Process Server ...................................... 25  

vi
5.3 Demand Page Algorithm ............................................. 59
5.4 File Server Algorithm .............................................. 61
5.5 Summary ................................................................. 64

6 Performance Improvement Opportunities ......................... 69
6.1 Protocol ................................................................. 70
6.2 Address Space .......................................................... 71
6.3 Messages ............................................................... 73
6.4 File Information ......................................................... 76
6.5 Object Allocation ....................................................... 78
6.6 Process Modularity ..................................................... 79
6.7 Summary ................................................................. 80

7 Freeze Free Algorithm .................................................. 81
7.1 Algorithm Steps ........................................................ 82
7.2 Process Migration Latency ............................................ 83
7.3 Message Handling ..................................................... 83
7.4 Page Faults and Flushing ............................................. 84
7.5 Summary ................................................................. 84

8 Performance Measurements ............................................... 86
8.1 Test System ............................................................. 87
8.1.1 Hardware ........................................................... 87
8.1.2 Operating System .................................................. 87
8.1.3 Communication Subsystem ..................................... 87
8.1.4 File System .......................................................... 89
8.1.5 Process Subsystem ................................................ 90
8.1.6 Application Operating System Interface ..................... 91
8.1.7 Process Migration Subsystem ................................... 91
8.2 Process Migration Performance ..................................... 92
8.2.1 Process Migration Latency ....................................... 92
8.2.2 Migrate Out Steps ................................................ 93
8.2.3 Migrate In Steps ................................................... 94
8.3 Process Migration Latency Period Analysis ....................... 102
8.3.1 Latency Period Critical Path .................................... 102
8.3.2 Extraction and Insertion ......................................... 104
8.3.3 Virtual Memory ..................................................... 105
8.3.4 Data Copying ....................................................... 106
8.3.5 Communication Cost .............................................. 107
8.4 Demand Page Analysis ............................................... 108
8.5 Load Sharing ............................................................ 110
8.6 Summary ................................................................. 119
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Algorithm Comparison</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>9.1 Normalized Performance</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>9.2 Demand Page Algorithm</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>9.3 File Server Algorithm</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>9.4 Distributed Virtual Memory</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>9.5 Summary</td>
<td>126</td>
</tr>
<tr>
<td>10</td>
<td>Conclusion</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>10.1 Process Migration Latency Time</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>10.2 Message Freeze Time</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>10.3 Modularity</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td>10.4 Load Sharing vs Overhead</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>10.5 Fault Tolerance</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>10.6 Opportunities for Further Improvement</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>10.6.1 Higher Bandwidth Network</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>10.6.2 Page Transfer Protocol</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>10.6.3 Communication Protocol Handling</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>10.6.4 Virtual Memory</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>10.7 Opportunities for Added Capabilities</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>10.8 Related Areas</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>10.9 Summary</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>Vita</td>
<td>150</td>
</tr>
</tbody>
</table>
# List of Tables

3.1 Process Migration Systems and Example Times ...................................... 48

8.1 ProcessMsg Message Transfer Performance .......................................... 89
8.2 Process Subsystem Performance ............................................................ 90
8.3 Migrate Out 4kB Page ............................................................................ 95
8.4 Migrate Out 8kB Page ............................................................................ 96
8.5 Migrate Out 16kB Page .......................................................................... 97
8.6 Migrate In Messages 4kB Page ............................................................... 99
8.7 Migrate In Messages 8kB Page ............................................................... 100
8.8 Migrate In Messages 16kB Page ............................................................. 101
8.9 Migrate In Object Pool .......................................................................... 102
8.10 Major Activities During Critical Path .................................................. 105
8.11 Page Fault Overhead in $\mu$s ............................................................... 108
8.12 Page Faults Per Second ....................................................................... 109
8.13 Major Activities During Demand Page ............................................... 109
8.14 Load Sharing Experiment Results ......................................................... 111

9.1 Normalized Comparison of Process Migration Latency .......................... 123
9.2 Comparative Process Migration Latency and Message Freeze Times ...... 124
List of Figures

3.1 Comparison Process Migration Latency Times .......................... 50
5.1 Total Copy Algorithm ......................................................... 65
5.2 Pre-Copy Algorithm ........................................................... 66
5.3 Demand Page Algorithm ....................................................... 67
5.4 File Server Algorithm .......................................................... 68
7.1 Freeze Free Algorithm .......................................................... 85
8.1 Process Migration Resource Usage (time in $\mu$s) ...................... 104
8.2 Page Size Impact on Performance .......................................... 112
8.3 Added Overhead versus Performance with 4kB Pages .................. 114
8.4 Added Overhead versus Performance with 8kB Pages ................. 115
8.5 Added Overhead versus Performance with 16kB Pages ............... 116
8.6 Combined Overhead versus Performance ................................. 118
Chapter 1

Introduction

Process migration moves a process from a source machine to a destination machine, which provides the ability to share work between machines, and potentially improve overall performance. Process migration enhances processor fault tolerance by providing the capability to move work away from a failing machine. Unfortunately, the benefits of process migration come with a price. Prior process migration systems add so much overhead to the computation of a migrated process that the overhead drastically limits the load sharing benefits. This thesis presents a new approach that dramatically reduces the overhead by an order of magnitude. This research concentrates on how to implement the process migration mechanism in a low cost manner. This thesis uses the new process migration system and a range of additional artificial overhead to demonstrate the negative impact overhead has on load sharing system speedup.

The chapter begins with a section that delimits the scope of the process migration problem tackled in this thesis. The second section declares the thesis statement and
highlights the major accomplishments of the thesis research. The final section outlines the organization of the remainder of the thesis.

1.1 Process Migration Problem Scope

Sharing work among machines encompasses a vast range of issues, and far too many for coverage in one thesis. This section defines the scope of this thesis. This research targets improvement in dynamic process migration, which is the transfer of running processes among homogeneous machines in a distributed system. This section defines each component of the dynamic process migration issue and its environment.

This thesis deals with process migration in a distributed system, where a distributed system is defined as a collection of multiple computer systems connected by a communication network and not by shared memory. Collections of workstations and server machines represent a typical distributed system. The mechanism of moving processes between processors within a shared memory multiprocessor differs from process migration in a distributed system, and is not considered further.

A single network often simultaneously supports a wide variety of computers. The transfer of a running program to a dissimilar machine encounters many translation problems. Among the more common differences are character set, integer format, floating point format, byte order, word size, instruction set, and register architecture. Thus, dynamic process migration between heterogeneous machines involves costly translations. While process migration between heterogeneous machines has been proposed [TH91],
the high translation costs severely limit its utility. This thesis deals solely with process migration between source and destination machines having the same machine architecture. For the purposes of this thesis, homogeneous machines can differ in clock speed, amount of memory, and both number and type of peripherals. Furthermore homogeneous machines can differ in portions of the machine that are hidden from the application processes but not the kernel, such as privileged instructions and the presence or absence of cache memory.

Another important element of the process environment is the operating system. A process uses operating system provided resources and services. Dynamic process migration requires the recreation of the state of the process - operating system interface on the destination machine. Translating the state of a process - operating system interface is costly and not always possible, because operating systems use different abstractions and do not always provide comparable services. This thesis deals solely with process migration between machines running the same operating system.

The major existing process migration algorithms and the new process migration algorithm are all independent of the application’s programming language. Thus an application process can be migrated regardless of its source programming language. These process migration algorithms deal strictly with the executable version of a process.

All approaches to executing a process on a new host involve three parts:

**Load Balancing Policy** – the system must determine when to transfer which process to transfer to which machine.
Process Migration Mechanism – the system migrates the selected process from the source machine to the destination machine.

Remote Execution – the process executes on the new machine.

The load balancing policy issue is a large and separate topic that is not covered here as part of the process migration mechanism.

The process migration mechanism must not add unnecessary overhead to the execution of a process on a remote host. This thesis does not cover other issues involving remote process execution.

Process migration occurs in two distinctly different situations:

Static Process Migration – the system migrates a process to another machine prior to execution. Static process migration migrates the relatively limited process environment information along with the process execution request. Static process migration is an effective means to moving work within a distributed system.

Dynamic Process Migration – the system migrates a process to another machine after the process has begun execution. The system must migrate the process environment plus the large amount of process state.

This thesis concentrates solely on Dynamic Process Migration and henceforth the term process migration refers to Dynamic Process Migration.

One of the most important issues in process migration is the minimization of the process transfer time. The time period from the issuance of a process migration request
until a process is ready to execute is called the *Process Migration Latency Time*. The primary focus of this thesis is minimizing the process migration latency time.

Message exchange is an important and frequent activity in distributed systems. Process migration must successfully deal with active messages arriving during a process migration. Prior process migration systems stopped message processing for an extended time period during process migration, and this period is called the *Message Freeze Time*. This thesis demonstrates an alternative.

Most current operating systems scatter the information about a single process throughout many of its component subsystems. The term *Excision* [Zay87b] refers to the process of extracting the process related information from an operating system kernel, and the term *Insertion* refers to the reverse operation.

### 1.2 Thesis Statement

The Mirchandaney et al. [MTS89] theoretical study shows that short process migration latency is essential to good performance gains, and provides a quantifiable target measure. The best of current systems do not achieve the needed speed. This thesis holds that the needed process migration latency time is achievable. This thesis research demonstrates an order of magnitude improvement in process migration latency time over the best of existing systems (Sprite at 330ms). It achieves at 14ms a process migration latency, which is well under the 25 milliseconds that this thesis extrapolates from the work of Mirchandaney et al (see section 2.6.3).
This thesis presents the new Freeze Free algorithm for Process Migration. The Freeze Free algorithm achieves a large performance improvement through improvements in six areas. The Freeze Free algorithm simultaneously reduces process migration latency, effectively eliminates message freeze time, and supports processor fault tolerance.

The Freeze Free algorithm minimizes information transfer during the process migration latency period by shipping just the combined process control and execution state, the current code page, the current heap page, and the current stack page. No process migration system can transfer less and still migrate a broad class of processes.

The Freeze Free algorithm eliminates request and response messages from the process migration latency period. The Freeze Free algorithm also eliminates signal messages by piggybacking the signals on data transfer messages.

The program counter identifies the current code page, and the stack pointer identifies the current stack page. A heuristic obtains the current heap page from a scan of the instruction stream. The Freeze Free algorithm delivers the first critical pages without page faults.

Disentangling the process control and communication states permits process migration and message receipt to proceed in parallel.

Disentangling the process control and the file state permits the migrating process to resume execution on the new host, while the old host flushes file cache blocks.

The Freeze Free algorithm places information about an object strictly within that object. This change greatly reduces the cost of extracting and inserting state information.
The *Freeze Free* algorithm preallocates and partially initializes a set of data structures. This change moves costly operations out of the critical path.

The thesis demonstrates the value of low process migration overhead through an experiment measuring the effects of overhead on load sharing system speedup. Low process migration overhead is essential for good system speedup.

### 1.3 Thesis Outline

This thesis presents a detailed study of the mechanics of process migration, and demonstrates a new more effective process migration algorithm with a working implementation. Chapter 2 presents the benefits of process migration, examines some of the principle constraints on process migration, and explains why process migration performance is very important. Chapter 3 describes a wide of range process migration systems and their major contributions. Chapter 4 provides an overview of the principle issues involved in process migration. Chapter 5 describes the four major process migration algorithms that are milestones in the development of the field. Chapter 6 identifies the major outstanding limitations of existing process migration systems, and how the *Freeze Free* algorithm overcomes these limitations. Chapter 7 presents the new *Freeze Free* Algorithm for process migration and shows how it overcomes existing bottlenecks. Chapter 8 reports performance measurements on the new *Freeze Free* Algorithm for process migration. Chapter 8 also describes a load sharing experiment with dynamic process migration that demonstrates the utility of dynamic process migration and the effects of overhead
on system speedup. Chapter 9 compares the performance of prior algorithms against the new Freeze Free algorithm, where the speed of each algorithmic step comes from this implementation. Chapter 10 summarizes the accomplishments of this thesis research and identifies related future research areas.
Chapter 2

Background

The original motivation was the desire to improve distributed operating systems by effectively harnessing the processor power available in a distributed system. Research began with an examination of load balancing policies. A review of dozens of load balancing strategies revealed 1) that the process migration mechanism is the foundation for load balancing and 2) that the existing process migration mechanisms are very costly. As a consequence, the research focus shifted to the process migration mechanism. Several major limitations existed. One theoretical study explored how fast process migration latency had to be in order to make process migration effective, and the results indicated that current process migration latency times greatly exceeded the time needed for good system load sharing speedup.

This chapter examines the environment in which process migration operates. The chapter begins with a section examining the availability of machines. The subsequent sections cover system models used for process migration, load indicators, static versus
dynamic load balancing, immobility versus process migration, time constraints, and load balancing strategies. The migration delay time constraint provided the initial justification for seeking a major reduction in process migration latency.

2.1 System Availability

Distributed systems are spreading. The steady reduction in hardware costs has led to organizations placing machines on everyone's desk. Humans do not constantly use their machines, which results in considerable unused processor capacity. In 1985 Theimer [TLC85, p. 2] reported that over one third of their work stations were idle at even the busiest times of the day. In 1987 Nichols [Nic87, p. 9] stated that 50 – 70 work stations out of 350 were idle during the day, and over 100 were idle at night. Mutka and Livny studied the use of MicroVAX II work stations and found over a 5 month period that work stations were available for outside use according to conservative criteria about 70% of the time [ML87a]. The machine utilization rates continue to drop. In 1990 Litzkow and Livny [LL90, p. 100] reported that 70% of their machines were idle. In 1991 Douglis and Ousterhout [DO91, p. 759] stated that 66 – 78% of their machines were idle. Also in 1991 Krueger and Chawla [KC91, p. 342] reported that on average 91% of their work station capacity was unused. Anderson et al report that more than 60% of their workstations were available 100% of the time [ACPT94, p. 11]. The latest results indicate that more than 4 machines are available per user. The net result is that a lot of computer power
is left unused, and in the future even more wasted computing power can be expected. Process migration provides the means to harness this idle processor power.

Price pressures are also driving increases in the use of workstations. The larger sales volume of smaller systems spreads costs over more units. This reduces the per unit costs and drives up profits, which motivates more investments. As a result, the price–performance ratio of smaller systems is increasing faster than larger systems. For example, workstation price–performance is increasing at 80% per year, while supercomputer price–performance is increasing only 20–30% per year [ACPT94, p. 1]. The end result is that increasing amounts of work will shift to the ever less expensive workstations in distributed systems.

2.2 System Model

A review of process migration systems and load balancing strategies, revealed that each is based on a conceptual model of the computer system. This section identifies three system models that guide the policy for process migration and load balancing.

2.2.1 Autonomous System

The Autonomous System model treats the distributed system as a collection of independent computer systems that have the secondary characteristic of having communications links. An Autonomous System belongs to one user. The Autonomous System is only available when it is not being used by its owner. In this context the owner is the person
who has primary control over a machine, and not necessarily the legal title. When the owner resumes use of the machine, the system evicts processes originating outside the machine by process migration. Processes normally execute on the originating machine. A special command causes processes to be executed on remote machines.

Condor [LL90][LLM88] is the prototype for the model of an Autonomous System with load balancing. The Condor default settings consider a machine idle when the machine has a CPU load average of less than 0.3 tasks and has had no keyboard or mouse activity for at least 15 minutes. Even with such conservative measures, they report that about 70% of their machines would have been idle without Condor. Condor manages a queue of jobs for remote execution on idle machines, using the Up-Down algorithm to allocate CPU resources in a fair way. In the course of 630 days Condor provided 6,000 CPU work days and completed over 144,000 jobs.

2.2.2 Integrated System

The Integrated System model treats the distributed system as a unified system. This model is closer to the Time Sharing model of a uni-processor system. The Integrated System model expands the Time Sharing model to include multiple CPU’s. The Integrated System views the CPU’s, storage devices, and I/O devices as a set of resources whose utilization is to be maximized for accomplishing the greatest amount of useful work. In an Integrated System the hardware may be the same set of work stations used by an Autonomous System, but the policy is different. An Integrated System does not have
the concept that an individual "owns" a work station. The system migrates processes to bring processor loads closer to a balance.

Amoeba [Tan92, chapter 14][MvRT+90] is the prototype for the Integrated System model. Amoeba manages machines as a processor pool. The Amoeba Run Server monitors the load on each of the processors. When a new program is to be executed, the Run Server uses heuristics to select a processor and causes the program to be executed there. The Run Server balances the load on the processors in the processor pool.

2.2.3 Massively Parallel Processor System

The Massively Parallel Processor (MPP) System model views a massively parallel processor machine primarily as a vehicle for running a large application with multiple threads. Under the MPP model, the application controls many items that on other systems are controlled by the operating system. Specifically the application itself is likely to control the load balancing of the processing units which make up the MPP, which differs from the general operating system level approach used in distributed systems.

The MPP community typically tackles problems having enormous computational requirements. That provides the justification for expending large programmer efforts on optimizing each particular program. The high cost of the application specific programmer optimization generally renders this approach unaffordable elsewhere.
2.2.4 Summary

While the three systems currently use their machines differently, two trends will bring them closer together. Existing process migration systems require a period of time that is large enough to be noticed by a human being. For example, Condor sometimes takes over 2 minutes to migrate a large process. When process migration can be accomplished in less time than a human being can notice, then the issue will be handled as a simple scheduling policy option. The Freeze Free process migration algorithm with a distributed file system makes this practical now.

Many manufacturers are building MPP supercomputers using the same processors, memory, and operating systems that are used in work stations [ADV+95, p. 267]. Thus many MPP supercomputers now resemble a distributed system, which is a collection of processors connected by a communications network without shared memory. As they begin to more closely resemble a distributed system, the software issues will also become more similar, including process migration.

2.3 Load Indicators

One of the primary benefits of process migration is load balancing. In general it is impossible to predict the run time of a process, so load balancing uses a measure of the current load to predict the future work load. This requires a useful measure of a processor’s load and an efficient way to collect the processor loads.
2.3.1 Processor Load Measures

A wide variety of system load measures are available. Twenty different system load measurements in a UNIX system appear in one paper [MW93, table 1]. A determination must be made as to which measures should be used.

Kunz [Kun91] attempted experimentally to determine the best measures. The experiment created an artificial work load containing CPU-bound, I/O-bound, and balanced tasks for a network of UNIX work stations. Controlling migration based on the “number of tasks in the run queue” yields better performance improvements than “size of the free available memory”, “rate of CPU context switches”, “rate of system calls”, “1 minute load average”, and “amount of free CPU time”. However, each of these indicators improves system performance. Next Kunz combined two of these indicators and measured the system improvement. Surprisingly, the “number of tasks in the run queue” performs better alone than it does when combined with another measure. If this result is repeated in other studies, it would cast doubt on the usefulness of complex load criteria. The study shows that a useful, easy to obtain load measure is already available.

2.3.2 Obtaining Load Measures

Systems use messages to bring the load information to one site so that load balancing decisions can be made. Messages are relatively slow by computer terms. This has motivated a search for ways to minimize message traffic. Eager et al analyzed the performance improvement in a distributed control strategy with increases in the number of probes to other nodes for their load information. Their results [ELZ86, Figure 6] show noticeable
improvement through 3 probes, lesser improvement up through 5 probes, and not much improvement with higher numbers. Ryou and Wong [RW89] also investigated whether adequate results could be obtained when querying less than the full set of nodes. They achieved satisfactory performance improvement by querying only a subset, half in their example. This shows that reasonable load balancing decisions can be made with a small number of messages.

### 2.4 Static vs. Dynamic Load Balancing

Iqbal et al [ISB86] studied the relative merits of static and dynamic load balancing. For jobs that require a small amount of computation, static outperforms dynamic load balancing. As the length of the jobs increases dynamic load balancing becomes more important and provides better performance. A related result is that the static method of load balancing becomes less and less important with longer jobs. Combining static and dynamic load balancing produces superior results to either method alone, and results in fewer job migrations. Krueger and Livny [KL88] also conclude that dynamic process migration can achieve considerable additional improvement over just static process migration.

### 2.5 Immobility

Zhou and Ferrari [ZF87] experimented with load balancing under various conditions, including situations with tasks that should not be migrated. Some jobs should not be
moved because they interact heavily with the user’s terminal, such as text editors and command line interpreters. Other programs are tied to other interfaces, such as mail and message programs. It is intuitively obvious that at some point load balancing becomes impractical as the percentage of immobile work increases. Zhou and Ferrari found that systems achieve most of the benefits of load balancing when up to 70% of the work load is immobile. This study shows that only a small part of the work load must be mobile, for process migration to effectively support load balancing.

2.6 Time Constraints

Several theoretical and empirical studies have investigated the time properties of task execution, system loading, and the effects of delay on process migration.

2.6.1 Task Characteristics

Leland and Ott analyzed 9,524,227 processes from VAX 780’s and 750’s. They identified 4 categories of processes:

1. Short with little CPU or disk usage (the vast majority of tasks),

2. Heavy CPU and little disk usage,

3. Heavy disk and little CPU usage, and

4. Heavy use of both disk and CPU (almost no tasks).
They found that 98% of the smallest processes use only 35% of all CPU time. While the 0.1% largest processes use 50% of CPU time. Additionally they found that on average the longer a process executes the longer the remaining CPU time [LO86, p. 57].

Cabrera examined 156,624 processes from 10 runs on VAX machines. Between 78 – 95% of all processes consume less than 1 second of CPU time [Cab86, p. 450]. Of the sub—one second tasks, 73 – 84% require less than 0.4 seconds of CPU time. When process life times were examined, 40% of the processes which have already lived for T time units will live a life time of 2T units of time [Cab86, p. 454].

Zhou examined 297,595 tasks from a Berkeley VAX installation, 1,168,579 tasks from a Bell VAX installation, and 592,661 tasks from a Lawrence Berkeley Laboratory VAX cluster installation. Zhou found “85 – 95% of total CPU time is consumed by jobs requiring more than 1 second or 10 – 25% of the jobs”[Zho87b, p. 119]. He further found that “60 – 83% of the jobs consume less than half a second of CPU time, and only 3 – 7% of the jobs require more than 4 seconds, in all of the traces”[Zho87b, p. 118].

In a study of the probability of load balancing success, Rommel [Rom91, p. 932] concludes that the presence of CPU hogs and ordinary size processes yields a high probability that load balancing will improve performance. The above studies on task characteristics confirm that this situation is likely.

Two important conclusions can be drawn from these studies. First, long running tasks can be recognized by the simple expedient of examining what a task has already done. Second, as the median life time of a process according to one study is 0.4 seconds [Cab86, p. 455], process migration has to be much quicker.
2.6.2 System Loading

Zhou [Zho87a, p. 80] found that the number of tasks ready for execution is a good load indicator. In the studied system the average number of tasks queued for execution was 6, and 33% of the time the net change was 3 or more. On average the CPU queue changed 2.31 in a 30 second interval. Thus system loads change often and by wide swings.

2.6.3 Migration Delay

Mirchandaney, Towsley, and Stankovic [MTS89, p. 1522], used theoretical models to examine the effects of migration delays from 0.1S to 100S on system performance, where S is process service time. They found that short delays ( ≤ 1/2 S) produced substantial gains. At high delays ( > 10 S) performance improvements disappear. Only the best of current implementations approach the small delays needed for better performance improvements.

The impact of process migration latency is closely related to processor speed. So faster processors probably require corresponding improvements in migration times. Since the Cabrera study, processor speeds have probably improved by at least a factor of eight. Taking the service time from the Cabrera study (400ms) and adjusting for processor improvement (adjust by a factor of 8) yields a modern service time of 50ms. Applying the Mirchandanay et al formula of one half the service time yields a target process migration latency time of ≤ 25ms. None of the existing systems achieve a process migration latency time that is even close. The new Freeze Free design achieves an even smaller process migration latency time.
Arpaci et al report “parallel programs on a 50 node cluster with high migration costs suffer a slowdown of two times or more than systems with lower migration costs” [ADV⁺95, p. 273]. This confirms that process migration costs impact overall system performance.

### 2.7 Load Balancing Strategies

Investigators have produced an enormous number of studies related to a very diverse set of load balancing strategies. Casavant and Kuhl [CK88] and Wang and Morris [WM85] both produced taxonomies of load balancing strategies with extensive annotated bibliographies. Theimer and Lantz [TL88] find that a centralized scheduler could be scaled to thousands of machines, which is more than adequate for today’s needs. Other systems, including Sprite, have found centralized schedulers quite satisfactory.

### 2.8 Summary

Studies show that in current distributed systems usually 3 out of 4 workstations are idle. So there is plenty of computer power available for a migrating process. The “number of tasks in the run queue” provides an adequate load measure, and is easily obtained. Studies show that adding dynamic process migration improves performance over just static process migration. Even when 70% of the processes are immobile, process migration can provide effective load balancing. Empirical evidence shows that past execution history is a fair predictor of future activity. Centralized schedulers can handle distributed systems
with thousands of machines. Thus we have sufficient machine resources, knowledge, and capabilities to implement useful dynamic process migration.

Currently most systems do not use dynamic process migration. An obstacle must exist, or most systems would use dynamic process migration for its load balancing and fault tolerance benefits. The Mirchandaney et al theoretical study points at the overhead of dynamic process migration as being the roadblock. Chapter 8 shows with actual performance numbers that dynamic process migration overhead is indeed the major barrier.
Chapter 3

Related Work

This chapter examines existing process migration systems and related work. The chapter places the emphasis on the migration mechanisms and the effectiveness of the design. The first section examines one programming language based system supporting migration, and shows the most important similarities and differences from operating system based process migration. The second section examines existing user level process migration systems. The fundamental performance limitations of user level process migration systems led to the decision to concentrate on kernel level process migration systems. The third section surveys the process migration mechanisms used by kernel level systems and their performance. The results of previous process migration systems provided useful guidance for designing an improved process migration system.
3.1 Programming Language Based Systems

Moving work does not have to be performed at the process level. Programming language based systems successfully move smaller units of work between machines. A brief examination of one programming language based system serves to highlight the differences and similarities. The Charm Parallel Programming System [Kal93] supports the migration of tasks [DR94], which Charm calls *char*es. *Char*es communicate via messages. Charm forwards messages for migrating *char*es, and updates the address information on the source machine. *Char* identifiers are location dependent, so Charm assigns new identifiers at migration time. Charm migrates just the *char*'s private, persistent data. Charm migrates a *char* with no persistent data in 800μs. Thus this programming language based system deals with the same problems in managing identifiers and communication links. The big difference lies in the much smaller state that a programming language based system migrates compared to process migration. For example, *char* migration does not involve code transfers.

3.2 User Level Process Migration

This section surveys ten process migration systems implemented at the user level.

3.2.1 rsh

The Unix *rsh* facility supports the execution of a subset of shell commands on a remote machine. One of the *rsh* options is the execution of a program on the remote machine.
\texttt{rsh} supports only static process migration. One of the many limitations is that \texttt{rsh} does not preserve the environment, such as the current working directory.

3.2.2 Zhou and Ferrari

Zhou and Ferrari implemented static process migration for Unix 4.3 BSD running on DEC VAX machines. They modified the C shell to execute remotely selected types of user commands. A Load Information Manager (LIM) constantly monitors system loads and performs job placement. A Load Balancing Manager (LBM) exists on each host to support the remote execution.

3.2.3 Butler

The Butler system [Nic87] at Carnegie Mellon University is a collection of programs that run on Andrew work stations. The Butler system manages a collection of idle work stations and automatically selects a work station for a work request. Butler supports only static process migration. Butler ships environment information to the destination host machine, which is an improvement over \texttt{rsh}. Butler follows the Autonomous System model. Butler provides a 2 minute warning when an owner reclaims a machine, and then kills remaining programs.

3.2.4 Remote Execution Monitor

The Remote Execution Monitor (REM) [SCTT88] was developed on Berkeley's Unix. The REM system gives application processes the ability to manage remote processes,
which means that an application can create and kill remote processes. REM supports varying levels of fault tolerance through replication of processes on multiple machines. REM adds a message protocol that supports location independence. REM achieved speedups of up to 3.5 [ST88, p. 361].

3.2.5 Process Server

The Process Server system runs on top of the Cedar system [Hag86]. The Process Server supports static process migration for those processes using only strings, streams, and files for communication.

3.2.6 Condor

Condor [LL90][LLM88] supports process migration on networks of Unix work stations. Condor is essentially a batch facility that allocates processes to idle work stations. The idle criteria are quite strict; the Unix load must be less than 0.3, and the keyboard and mouse must be inactive for at least 15 minutes. When the owner of a machine begins to use the work station, Condor migrates its processes away. Condor targets primarily long lived processes, and supports just processes that only use files and perform internal processing. Condor forwards file requests to the submitting machine. Condor uses the Up-Down algorithm [ML87b] to schedule machines fairly. Condor performs process migration by checkpointing a process to a file, transferring the file, and then restoring the process from the checkpoint file. Checkpointing and placement consume approximately
5 seconds per megabyte of the checkpoint file, and the average cost of checkpointing and placement is approximately 2.5 seconds.

3.2.7 Mandelberg and Sunderam

Mandelberg and Sunderam implemented an unnamed process migration system on Unix. The system requires that a migrating process not perform any of the following actions: communicate over pipes or sockets, access non-NFS files, or spawn processes. The system uses a pair of programs to support the application terminal interface after migration. On the new host a program pretending to be the terminal interface exchanges data with a program on the old host, who in turn exchanges data with the real terminal interface. The system supports dynamic process migration by moving the entire process state in one large operation. The user process migration level system requires approximately 3 seconds per 100kB of core image [MS88, p. 362].

3.2.8 Dediu, Chang, and Azzam

Dediu et al implemented an unnamed process migration system on Unix designed to support very long running processes. Their design collects all of the process state into a file, transfers the information across using a file transfer protocol, and then reconstructs the process on the new host machine. The transfer time is on the order of minutes [DCA92, p. 234].
3.2.9 PMS

The Process Migration System (PMS) [Fre91] approaches process migration with a totally different perspective. Currently most systems do not provide universal location transparent access to computer resources, because of the cost and difficulty. Some application processes are CPU-bound and make little use of operating system resources. PMS targets these applications. PMS provides a process migration facility that migrates only those processes which only use resources that have location access transparency. PMS delegates responsibility to the application for correctly handling access to any resources lacking location transparent access. PMS demonstrates its approach on a compute bound ray-tracing program.

3.2.10 Utopia

Utopia supports static process migration. While the system can migrate “almost all applications” [ZZWD93, p. 1320], Utopia still does not support a few features for migrated processes, such as process groups. Utopia supports file access for migrated processes on those systems that have a uniform shared file name space. Andrew, Athena, and HCS meet this requirement, while NFS does not. The system is very unusual in that Utopia functions on multiple operating systems: Ultrix, Sun-OS, HP-UX, AIX, IRIX, and OSF/1; all of which derive from Unix.

Utopia provides automated load balancing. Scalability to distributed systems with large numbers of machines is an important criteria, as Utopia run at sites each of which
contain several thousand work stations. Utopia supports clusters of work stations within a larger distributed system, which provides a form of hierarchical control.

Utopia includes utilities, such as a command interpreter and a batch facility, that take advantage of the parallel processing power. A parallel make facility demonstrates speedups of 17 on a 25 machine system [ZZWD93, p. 1332].

3.2.11 Summary

User level implementations of process migration systems have two principle problems. The user level implementation can not access all of the kernel state for an arbitrary process. Therefore user level implementations can never migrate certain classes of processes. Secondly, the user level implementation must cross the kernel - application protection boundary every time it requests a kernel service, including accessing kernel state. The boundary crossing is expensive, and no user level implementation of process migration can compete with a kernel level implementation in terms of overhead. The principle reason for the user level implementations is the relatively easier software development and maintenance environment.

3.3 Kernel Level Process Migration

This section surveys process migration systems implemented at the kernel level, and concentrates on the process migration mechanism.
3.3.1 Locus

Locus [WPE+83] is a distributed operating system based on Unix that provides single processor Unix semantics in a distributed, multi-processor environment. Locus provides a network-wide, location independent name structure for its distributed file system [PWC+81]. As Locus follows the Unix design of placing many system resources under the file system, the network transparent file system facilitates process migration. Locus supports static process migration with a modified fork command, and supports dynamic process migration with a migrate command [Smi88]. Locus supports dynamic process migration using the Total Copy algorithm (refer to section 5.1). The origin site of a process keeps track of that process [PW85].

3.3.2 Demos/MP

Demos/MP [PM83] supports dynamic process migration using the Total Copy algorithm (refer to section 5.1). Demos/MP is a message based distributed operating system. Demos/MP places no process state in the various functional modules of the operating system. Demos/MP servers maintain resource state. The universal use of messages coupled with the high degree of locality with respect to state greatly facilitates the implementation of process migration in Demos/MP. Demos/MP sends messages over buffered, one-way channels, called Links. Demos/MP maintains Links during and after process migration by forwarding messages to the new machine. Routing messages through the old host is costly. The forwarding machine informs any message sender machine of the correct new destination machine address, which then updates the link.
3.3.3 MOS

MOS [BSW89] is a distributed operating system that is compatible with Unix Version 7 at the application level. MOS attempts to provide the illusion that the network of computers is a single system. MOS supports dynamic process migration. MOS reports being able to transfer the memory at 5.4ms per 1kB, or 540ms for a 100kB process. The authors did not report the overall process migration time. MOS runs on four PDP-11 computers connected by a 10M bit per second Local Area Network (LAN).

3.3.4 Process Suspend/Resume

On an AT&T Unix System V operating system Cagle [Cag86] implemented the ability to suspend a process to a file and at some later time resume the process from that file. The suspend command works even when the operating system has been rebooted while the process was suspended. Chen [Che86] implemented essentially the same capabilities for the Unix 4.2BSD operating system. They store information on file names in the form of complete file path names. While they did not perform process migration, the use of a network file system coupled with a file transfer capability would in principle allow some processes to be migrated slowly. Both systems find that they need kernel level modifications to obtain all of the necessary information.
3.3.5 Nest

The Nest [AE85] system runs on a collection of AT&T 3B2 computers connected by an AT&T 3BNET, which is compatible with 10M bit per second Ethernet. Some computers operate permanently as compute servers, and other machines become available when they are otherwise idle. Nest manages the machine resource pool. The `reexec` command submits work for remote execution. Nest automatically selects a host when the user does not specify a host. Nest supports static process migration. Nest preserves the migrated process view of the file system, parent–child relationship, process groups, and process signals. A Nest switch determines which file activity is redirected to the home machine. When a migrated process accesses system utilities, system libraries, and temporary files locally and both input and output files are redirected, the migrated process suffers a 6% execution performance penalty versus a 55.6% penalty for when the system redirects all accesses to the home machine [Ezz86].

3.3.6 Alonso and Kyrimis

Alonso and Kyrimis modified version 3.3 of the Sun Unix operating system to support dynamic process migration [AK88]. Their process migration facility supports neither processes that communicate with other processes using pipes or sockets, nor processes that depend on environment information, such as process identification. Their process migration facility uses a variant of the Total Copy algorithm (see section 5.1). At process migration time, the system checkpoints the process to three files. The first file contains a dump of the text and data regions. The second file contains the name of the host, the
current working directory, file information, and terminal flags. The third file contains the stack, registers, and process control information. The system transfers all of the data, and uses that data to restore the checkpointed process. The system requires 5.78 seconds to migrate a process to a new host. The system runs on Sun 2 work stations connected by a 10M bit per second Ethernet LAN.

### 3.3.7 Emerald

Emerald [JLHB88] is an object based programming language and distributed system. Emerald runs on top of the Ultrix operating system. Emerald is not a complete operating system. But since Emerald has processes, this section covers Emerald with the other process migration systems. Emerald supports the migration of all objects, including objects possessing a process. An Emerald object does not have its own address space, instead it is linked into the node’s address space. This means that object migration does not involve the transfer of an address space. Emerald migrates the data in the object. One disadvantage is that references have to be translated at migration time. An Emerald reference possesses additional information that support translation. An Emerald process carries out actions by invoking methods on various objects. Each invocation involves an Activation Record. Since Emerald does not place Activation Records on a stack, Emerald must collect the Activation Records at process migration time. Emerald code does not change, so code is readily replicated. Emerald migration often avoids migrating code, because a copy of the code is already on the destination machine. Emerald migrates a process containing 6 variables in 40ms [Jul89]. Emerald transfers the process
in a message of about 600 bytes, which includes object references, immediate data, a
stack segment, and general process control information. Emerald runs on MicroVAX II
machines connected by Ethernet. While Emerald process migration is fast compared to
other systems described in this section, Emerald process migration is clearly migrating
something quite different.

3.3.8 AIX/TCF

The AIX Transparent Computing Facility (TCF) [WM89] is a distributed operating
system that gives the illusion of a single machine Unix system. AIX/TCF supports
static process migration with the rexec, rfork, and run commands and dynamic process
migration with the migrate and SIGMIGRATE commands. AIX/TCF process migration
initially sends a remote tasking message containing part of the proc and user structures
along with open file information. A kernel server process forks a child process which
obtains environment, argument, stack, and data information. The child process demand
pages the program code if it is pure code, otherwise the child process copies the code.
The child process sends a fork done (FRKDNE) message to the source machine. The
source machine sends a message (EXECDNE) that contains any last signals and confirms
that the process has been destroyed on the source machine. Each machine keeps track of
all processes created there, even when the process may have migrated. Thus AIX/TCF
process migration does not support fault tolerance.
3.3.9 MOSIX

MOSIX [BW89] is a distributed operating system that is compatible with AT&T Unix Version 5.2. MOSIX gives the application the illusion that the application is running on a single processor system. MOSIX is a descendant of MOS. MOSIX supports dynamic process migration [BSW89]. MOSIX runs on National Semiconductor NS32532 processors connected by a ProNet-80, which is an 80M bit per second token-ring LAN. The memory transfer rate is 1.9ms per 1kB. They do not report overall process migration times. MOSIX achieves speedups of 3.86 with 4 processors and automatic process migration.

3.3.10 Mess

Mess [DEGM90] is a distributed operating system that is based on two premises. Mess treats all memory as one global virtual memory. Mess provides universal and transparent access to all resources, including processors. The presence of multiple processors and a network is invisible to applications. Process migration occurs as a byproduct of the unified Distributed Virtual Memory (DVM). Mach [MZDG93][MGZ93] also uses DVM for address space transfers. Malkawi et al studied load balancing in a simulation that models process migration as a process control transfer followed by memory transfer based on DVM [MA93]. The Mess publication [DEGM90] does not make clear the Mess implementation status.
3.3.11 Rhodos

Research Oriented Distributed Operating System (RHODOS) [GG1+91] is an experimental, message based, distributed operating system. RHODOS provides access and location transparency, as the design goal is to hide the distributed nature of the physical system. RHODOS supports preemptive load balancing through process migration [Nut94]. The Load Balancing Server decides when to migrate which process to which machine. The Load Balancing Server manages the collection of load information, and supports a variety of migration policies. The Process Migration Manager provides the process migration mechanism, which uses a variant of the Total Copy algorithm (see section 5.1). At process migration time, the Process Migration Manager freezes the migrating process. The Process Migration Manager transfers the entire state. Next the Inter-Process Communication Manager freezes the message ports, transfers them along with buffered messages, and redirects messages. The old host Process Migration Manager destroys the local migrating process. The new host Process Migration Manager resumes the migrating process.

3.3.12 Clouds

Clouds [DLAR91][DCM+90] is a distributed operating system. Clouds is based on its own object - thread model. A Clouds object is a large grained encapsulation of code and data. A Clouds object is passive, persistent, and exists within its own virtual memory space. A Clouds thread causes actions to occur by invoking object methods, which execute within the address space of the called object. In effect threads move from object to object in the

35
course of normal execution. The objects can reside on any processor. Thus threads easily migrate between processors. Clouds moves objects between processors according to its policy under a demand page mechanism. Clouds transfers an 8kB page in 12.3 ms. Clouds shows that the conventional process model is just one possible system model. Because the Clouds movement of data, code, and control among processors does not correlate directly with normal process migration, Clouds performance is also incomparable.

### 3.3.13 Galaxy

Galaxy [SMS+91] is a distributed operating system. Galaxy views all entities as objects, including files, devices, and nodes. Object access occurs in a location transparent manner, which facilitates process migration. Galaxy does not have any notion of an origin machine for a process.

Galaxy supports dynamic process migration [SPJ+91]. The Galaxy address space transfer algorithm combines elements from the *Pre-Copy* algorithm (see section 5.2) and the *Demand Page* algorithm (see section 5.3). At process migration time, process execution continues while Galaxy transfers memory resident pages that are not part of the working set. Then Galaxy freezes the migrating process. Galaxy transfers the process state. Galaxy transmits the pages in the working set and any pre–copied page that the process subsequently modified. At this point Galaxy resumes process execution on the new machine. In parallel with execution, Galaxy transfers the remaining pages.

All Galaxy objects, including processes, have a unique object ID. Galaxy keeps location information about an object in an ID Table Entry (IDTE). At process migration
time the IDTEs must be updated. The problem is similar to updating the Links in Demos/MP. Immediately prior to freezing a migrating process, Galaxy informs the message server. The message server forwards messages to the new host. When the migrating process is ready to begin execution on the new host, Galaxy sends a message to the ID manager at each host that has a copy of the migrating process ID. This protocol reduces the amount of message forwarding.

3.3.14 EMPS

The Eindhoven MultiProcessor System (EMPS) [vDvG92] supports dynamic process migration using the Total Copy algorithm (refer to section 5.1). EMPS delivers messages to Mailboxes, which are not part of a specific process. A process connects to a Mailbox via a Port. The Mailbox records all of the connected processes and their machine locations. This design provides an incomplete separation between message processing and process migration. If a process is not receiving or sending a message, Mailbox message processing continues unaffected during a process migration. Otherwise the system suspends Mailbox message processing for the duration of the process migration. The EMPS separation of process and Mailbox provides a partially effective means of separating process migration and message processing. However, the worst case message freeze time remains unchanged.

EMPS runs on Motorola MC68030 20MHz processors connected by the PHYLAN 2.5M bits per second LAN. The EMPS process migration time formula in milliseconds
is:

\[ TotalTime = 29.94 + 6.84 \times \text{Memory Size KiloBytes} + 16.49 \times \text{Number MailBoxes} \] (3.1)

EMPS has the fastest reported process migration times for very small processes. For example, EMPS migrates a 10kB process with 1 mailbox in 114.83 ms. EMPS performance does not scale well with increasing process size. At approximately the 50kB size process, Sprite begins to outperform EMPS [vDvG92, Fig. 3]. EMPS also reports process migration times within a machine, but those figures are not comparable to process migration times between machines.

3.3.15 BirliX

BirliX [HKLR92] is a distributed operating system that is based on an object model that emphasizes persistent objects. The team is the BirliX version of a process. The team consists of a collection of segments representing memory regions, threads, and access descriptors representing communication bindings. BirliX supports dynamic process migration [LHK93] using a version of the Total Copy algorithm (see section 5.1). BirliX checkpoints the migrating team at process migration time. BirliX sends a migration request message to the Type Manager on the destination machine. After the receipt of an acknowledgement, BirliX transfers the checkpointed information. After the arrival of all information, BirliX performs a recovery. A destination agent thread sends a message to the source machine team containing the destination agent identification. BirliX forwards
the identification to the client team that initiated the process migration, and destroys
the old team. A recent upgrade added the ability to tailor on a per object basis the stan-
dard BirliX migration mechanisms [Lux95]. The goals are to take advantage of specific
object properties to reduce migration costs and to support different migration policies.
BirliX migration supports the migration of many object types, including processes, files,
and dictionaries. BirliX takes approximately 500 ms to migrate a 100kB team with one
thread (based on a rough estimate from a performance graph [Lux95, p. 67]). BirliX
runs on a network of Sun-3/60 machines connected by a 10M bit per second Ethernet
LAN.

3.3.16 Amoeba

Amoeba [MvRT+90] is a distributed operating system based on objects and is heavily
oriented towards a client–server architecture. Amoeba supports static process migration.
Despite earlier reports about dynamic process migration and Amoeba, the actual imple-
mentation of dynamic process migration was first reported in 1994 [SZM94]. Amoeba
uses the Total Copy algorithm (see section 5.1). At process migration time, Amoeba
freezes the migrating processes. Amoeba rejects incoming messages with the “process
is migrating” reason. The Amoeba source Migration Server sends a message containing
control information and waits for acceptance. The destination Migration Server obtains
the memory of the migrating process via a series of RPC requests. Finally the source
Migration Server transmits an execution token, which causes the migrating process to
resume execution. The source machine responds to incoming messages with a “not here”
reply. Normal message recovery eventually updates the address of the migrating process and correctly delivers the messages affected by the process migration.

3.3.17 Mach

Mach [ABB+86] is a message based operating system. Milojicic et al have implemented dynamic task migration on Mach [MZDG93][MGZ93]. Their task migration operates at the Mach level, and is independent of the operating system emulations, such as Unix or VMS. Their task migration is even independent of Mach applications. This raises the question of whether it is appropriate to migrate Mach level tasks without migrating the higher level process. Milojicic et al report that migrating the task without migrating the process causes performance degradation after migration, because of the actions which must be forwarded to the home machine. Residual dependencies that occur when the system does not simultaneously migrate the higher level process prevent task migration from supporting fault tolerance.

Milojicic et al implemented two task migration systems. The Simple Migration Server (SMS) and the Optimized Migration Server (OMS) both primarily operate at the user level. However, they found that task migration could not be implemented without two kernel modifications. One kernel modification provides access to the task and thread kernel ports, and the other modification provides the ability to export the pager port. Mach task migration first suspends the task, and aborts all threads to clear associated kernel state. Mach task migration obtains the task and thread kernel ports. Next Mach logically transfers the address space. Then Mach transfers the threads, capabilities, and
other task state. Finally Mach restores the task and thread ports just prior to resuming
the task. Mach task migration normally demand pages the address space. The OMS
differs from SMS by using fewer messages to transfer the thread, capability, and other
task state information, and by supporting additional memory transfer policies: pre-copy,
flushing (similar to Sprite), and eager copy (similar to MOSIX). They dropped OMS
because SMS was “more robust” and OMS had limitations on what types of tasks OMS
could transfer. Since the developers dropped OMS in favor of SMS, this thesis uses SMS
as the standard-bearer of their work.

Their version of Mach supports NO Remote Memory Access (NORMA) Distributed
Shared Memory (DSM) using eXtended Memory Management (XMM), which is imple-
mented in the kernel [FBS89]. XMM supports demand paging. Their version of Mach
memory maps files, which allows DSM to provide file access support for migrated pro-
cesses. For performance reasons Milojicic et al are investigating adding a distributed file
system.

Mach task migration runs on 33MHz Intel 80486 processors connected via Ethernet.
Performance measurements show that on a typical size task SMS performs task migration
in 500ms and OMS performs task migration in 250ms. The SMS task migration time
formula in milliseconds is:

\[
Total Time = 150 + 48 \times NumberOfMemoryRegions \\
+ 22.8 \times NumberOfReceiveRights \\
+ 5.5 \times NumberOfSendRights
\]  

(3.2)
\[+5.5 \times \text{NumberOfSendOnceRights}\]
\[+58 \times \text{NumberOfThreads}\]

The OMS task migration time formula in milliseconds is:

\[
\text{TotalTime} = 50 + 2.4 \times \text{NumberOfMemoryRegions} + 7.9 \times \text{NumberOfReceiveRights} + 1.9 \times \text{NumberOfSendRights} + 1.1 \times \text{NumberOfSendOnceRights} + 5.4 \times \text{NumberOfThreads}
\] (3.3)

### 3.3.18 MDX

The Modular Distributed Computing System (MDX) extends migration capabilities to nearly all system objects, including processes [Sch95]. MDX uses location transparent naming. MDX uses a variant of the Total Copy algorithm (see section 5.1) to migrate processes. MDX runs on 33MHz Intel 80486 PC/AT computers connected by a 10M bit per second Ethernet. MDX migrates a 100kB process in roughly 360ms [Sch95, p. 80].

### 3.3.19 Charlotte

Charlotte is a message based, distributed operating system. Charlotte [AF89][FA89] supports dynamic process migration using a variant of the Total Copy algorithm (see
section 5.1), Charlotte processes communicate over connection oriented, duplex communication channels, which the system represents by capabilities called links. Charlotte buffers a message on the sender side until the destination actually accepts that message, and caches messages on the receiver side. Charlotte freezes communications during process migration only while the system marshals and transfers the process context. Charlotte delays message delivery during process migration. Prior to process migration, Starter processes collaborate to decide whether to migrate a process. At process migration time, the source and destination machines exchange a pair of messages on whether the destination will accept the migrating process. Then Charlotte transfers the address space. Next Charlotte updates the links. Finally Charlotte transfers the process control information.

Charlotte runs on VAX/11-750 machines connected by a Pronet token ring. Charlotte migrates a 100kB process with 6 links in approximately 750ms. The Charlotte process migration time formula in milliseconds is:

\[
TotalTime = 45 + 78 \times PreApprovalFlag + 12.2 \times SizeIn2kBBlocks + 9.9 \times LinkFlag + 1.7 \times NumberOfLinks
\]  

The \textit{PreApprovalFlag} is 0 when the process migration is already approved, and 1 otherwise. The \textit{LinkFlag} is 0 when all links are local, and 1 otherwise. The \textit{NumberOfLinks} is set to 1 when there are no links.
3.3.20 V

V [Che88] is a message based, distributed operating system. V runs on collections of disk-less workstations and servers. Because processes already accessed all files across the network, migrating a process to a new machine did not cause significant changes for the file system. The V process uses only inter-process communication primitives (messages) to access anything outside the process address space. Updating the communication links serves to reconnect a process after migration. V process migration relies on error recovery to re-establish communication links after migration. The transfer time of an entire address space for many processes is longer than the message timeout periods, which means program failures could occur as a result of a process migration using the Total Copy algorithm. Theimer developed the Pre-Copy algorithm [TLC85][The86] to overcome the message freeze problem. Section 5.2 describes the Pre-Copy algorithm in detail. V did not possess virtual memory at the time, though it does now [Stu88]. V is a milestone in process migration by showing that the address space of a process does not have to be transferred in one operation.

V runs on Sun 10MHz 68010 processors connected by a 10M bit per second Ethernet LAN. V migrates a 100kB process in approximately 680ms. The V process migration time formula in milliseconds is:

\[ TotalTime = 80 + 6 \times MemorySizeKiloBytes \]  

(3.5)
3.3.21 Accent

Accent [Gos92, sec. 14.2] is a message based, distributed operating system. Zayas implemented the first process migration system using *copy on reference* to transfer the address space [Zay87b][Zay87a]. At process migration time Accent moves the migrating process into two messages. The *Core* message contains the Perq computer microengine, the kernel stack, the process control block, and the set of port rights. Accent physically transfers the contents of the *Core* message to the destination machine. The Real and Imaginary Memory Address Space (RIMAS) message contains a logical representation of the virtual address space. The process is restored on the destination machine using the *Core* and RIMAS messages. When a memory page is requested, the destination machine obtains it from the source machine. The *copy on reference* technique averages 58% fewer byte transfers and a 47% reduction in message processing time. The Accent process migration algorithm is a milestone in process migration performance. However, Accent loses the ability to support fault tolerance, because of the residual dependency for memory.

Accent runs on Perq work stations connected by a 10M bit per second Ethernet LAN. Accent migrates a 100kB process in approximately 13,000ms. The Accent process migration time formula in milliseconds is:

\[
\text{TotalTime} = 1180 + 115 \times \text{Memory Size KiloBytes} \quad (3.6)
\]
3.3.22 Sprite

Sprite [OCD88] is a network operating system with a distributed file system. Sprite process migration [DO87][Dou89][DO89][Dou90][DO91] takes advantage of the Sprite network file server to gain much of the benefits of demand paged memory transfer while simultaneously eliminating the long term memory residual dependencies. The integration of file server support into process migration is another milestone in the development of process migration. Sprite does not completely eliminate residual dependencies, because Sprite forwards a small set of kernel service requests to the home machine of a migrating process. Most Sprite kernel service requests are serviced by the current host machine’s kernel. At process migration time Sprite transfers all of the migrating process state except the cached file blocks and address space directly to the destination machine. Sprite flushes the cached file blocks and memory pages that differ from the backing store copy to the Sprite file server. Sprite then resumes execution of the migrating process on the destination machine. The migrating process begins on the destination machine without any memory pages or cached file blocks. Sprite demand pages memory and file data on the new host.

Sprite runs on SPARCstation 1 work stations connected by a 10M bit per second Ethernet LAN. The average Sprite process migration time is about 330ms [DO91, p. 775], which according to the formula corresponds roughly to a 100kB process with about 6 open files. The Sprite process migration time formula in milliseconds is:

\[
TotalTime = 76 + 9.4 \times NumberOpenFiles
\]  

(3.7)
3.3.23 Summary

The many existing dynamic process migration systems show that the increased difficulty of a kernel level implementation is not a serious obstacle. Despite access to all of the state of a migrating process, none of the existing kernel level dynamic process migration systems supports the migration of all application processes. However, the kernel level systems support dynamic process migration for a wider class of application processes.

The kernel level implementations perform dynamic process migration in far less time than the user level implementations. Table 3.1 reports the process migration time for a 100kB process, the algorithm, the computer, the LAN, and the year results were published for several systems [TLC85] [Zay87a] [AF89] [DO91] [vDvG92] [MZDG93] [MGZ93] [Lux95, rough estimate p. 67] [Sch95, p. 80]. All of the process migration systems report the time the migrating program is resumed as the process migration latency time. However, the algorithms differ considerably in the amount of state that they transfer. The Total Copy and Pre-Copy algorithms transfer all of the state. The Demand Page and File Server algorithms transfer all of the state other than the address space and file cache blocks. The Freeze Free algorithm transfers the process control state, execution state, the current code page, the current stack page, and the current heap page. The Demand Page and File Server algorithms must demand page the current code page, the current
<table>
<thead>
<tr>
<th>Operating System</th>
<th>Computer</th>
<th>LAN Name - Mb/s</th>
<th>Year</th>
<th>Sample Time ms</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Sun 10mHz 68010</td>
<td>Ethernet 10</td>
<td>1985</td>
<td>680</td>
<td>Pre-Copy</td>
</tr>
<tr>
<td>Accent</td>
<td>Perq workstations</td>
<td>Ethernet 10</td>
<td>1987</td>
<td>13,000</td>
<td>Demand Page</td>
</tr>
<tr>
<td>Charlotte</td>
<td>VAX 11/750</td>
<td>Pronet</td>
<td>1989</td>
<td>750</td>
<td>Total Copy</td>
</tr>
<tr>
<td>Sprite</td>
<td>SPARCstation1</td>
<td>Ethernet 10</td>
<td>1991</td>
<td>330</td>
<td>File Server</td>
</tr>
<tr>
<td>EMPS</td>
<td>MC68030</td>
<td>Phylan 2.5</td>
<td>1992</td>
<td>730</td>
<td>Total Copy</td>
</tr>
<tr>
<td>Mach SMS</td>
<td>33mHz Intel80486</td>
<td>Ethernet 10</td>
<td>1993</td>
<td>500</td>
<td>Demand Page</td>
</tr>
<tr>
<td>Mach OMS</td>
<td>33mHz Intel80486</td>
<td>Ethernet 10</td>
<td>1993</td>
<td>250</td>
<td>Demand Page</td>
</tr>
<tr>
<td>BerliX</td>
<td>Sun3/60</td>
<td>Ethernet 10</td>
<td>1995</td>
<td>500</td>
<td>Total Copy</td>
</tr>
<tr>
<td>MDX</td>
<td>33mHz Intel80486</td>
<td>Ethernet 10</td>
<td>1995</td>
<td>360</td>
<td>Total Copy</td>
</tr>
<tr>
<td>Choices</td>
<td>SPARCstation 2</td>
<td>Ethernet 10</td>
<td>1995</td>
<td>14</td>
<td>Freeze Free</td>
</tr>
</tbody>
</table>

Table 3.1: Process Migration Systems and Example Times

heap page, and the current stack page before the vast majority of migrating processes can get useful execution underway. For example, adding three page transfers at 12ms each [DO91, p. 776] and three requests at 1ms each to Sprite’s migration time of 330ms yields an effective process migration latency time of about 369ms.

Figure 3.1 graphically shows the process migration latency times for kernel level process migration systems on a 100kB process. The times are not adjusted for hardware differences. Figure 3.1 shows that for about a decade process migration latency times have remained quite large despite the use of a wide variety of hardware.

3.4 Summary

This chapter began with a brief comparison of the similarities and differences of programming language based migration and process migration. The chapter surveyed ten user level process migration systems. The inability to obtain all of the process state and
the performance handicap caused by the repeated crossings of the kernel - application boundary severely handicap user level process migration systems. This thesis drops user level process migration systems from further consideration because of these handicaps. The chapter surveyed 21 kernel level process migration systems. They have a fundamental performance advantage over user level process migration systems and have access to all of the process state, which means they can migrate a greater variety of processes. Still existing kernel level systems suffer from too much overhead and the complications of the message freeze time. Some process migration systems support processor fault tolerance.
System | Process Migration Latency Times
--- | ---
V | 13 Seconds
Accent | 
Charlotte | 
Sprite | 
EMPS | 
Mach OMS | 
Mach SMS | 
BerliX | 
MDX | 
Choices | 

Seconds

Figure 3.1: Comparison Process Migration Latency Times
Chapter 4

Mechanism

The mechanism that migrates a running process to a new machine is the focus of this thesis. This chapter identifies the major issues facing all process migration mechanisms. The process migration mechanism decomposes into a protocol controlling information transfer and the transfer of the process control state, execution state, address space, communication state, and file state. A separate section covers each of these major areas.

4.1 Process Migration Protocol

The process migration protocol defines the steps taken by two machines to agree to a process migration. The protocol establishes the transfer order for the state components of a migrating process. Two major factors influence protocol design. The new host may reject a process migration. The protocol should not preclude the old host from recovering the migrating process when the new host does not accept the migrating process. Improved
network communications offer improved data bandwidth, while still being limited to
approximately the same round trip times [Par94, p. 313]. Performance goals call for
minimizing both round trip messages and the total number of messages.

4.2 Process Control

The process control state comprises all of the information used by the operating system
to manage the process. The process control information includes priority, state, process
identification, and parent process identification. The most important characteristic is
that once the system marshals the process control information, the process migration
system must freeze the migrating process.

4.3 Execution State

The execution state includes the processor state, which represents the current execution
of the process. The execution state includes the general purpose registers, floating point
registers, stack pointer, program counter, and status registers. The execution state is
heavily machine dependent.

4.4 Address Space

The address space incorporates all of the virtual memory belonging to the process. The
address space is the largest component of the state of a process. Transferring the entire
address space as a single unit results in large transfer times for even relatively small applications. If a process migration goal is to support load balancing, then transferring the address space as a unit is not desirable.

4.5 Messages

The message state of a process consists of buffered messages and control information about the link. The management of the communication links during process migration and the re-establishment of the communication links after process migration are the major challenges of migrating the message state of a process.

4.6 Files

The file state of a process consists of the file descriptors and the cached file blocks. The possible existence of temporary files, which are automatically deleted when the file is closed, forces a general purpose process migration mechanism to transfer the files in an open state. Large file caches exist to reduce the work load on the file server. A long delay results if the process migration mechanism waits on an entire file cache flush to either the file server or new host.
4.7 Summary

This chapter has covered the major issues facing all process migration systems in the areas of protocol controlling the information transfer, and the transfer of the process control state, address space, communication state, and file state. The next chapter examines how existing process migration algorithms address these issues.
Chapter 5

Existing Process Migration Algorithms

Computer scientists have developed four primary process migration algorithms. This chapter presents the process migration algorithms according to the chronological order of invention: Total Copy, Pre-Copy, Demand Page, and File Server algorithms. The chronological order correlates directly with the order of increasing sophistication. The study of the existing process migration algorithms serves both to show what has been accomplished and where current limitations exist.

The actual systems which first implemented each of the principle process migration algorithms differ in terms of their actual process components. For example, Sprite application processes communicate through the file system instead of via messages [DOKT91, p. 364]. The published reports vary considerably in the level of detail presented on the respective process migration algorithms. Therefore this thesis presents generalized ver-
sions of the principle process migration algorithms that use the same process components standardized at the same level of granularity. The generalized versions assume that a distributed file system is present.

5.1 Total Copy Algorithm

The Total Copy algorithm is both the first and the most widely used process migration algorithm. Locus [Smi88], Demos/MP [PM83], AIX/TCF [WM89], Rhodos [GGI+91], EMPS [vDvG92], Amoeba [SZM94], MDX [Sch95], and Charlotte [AF89] use variations of the Total Copy algorithm. The Total Copy algorithm suspends a migrating process, transfers all state information to the new host, and then resumes the migrating process.

One generalized variant of the Total Copy algorithm follows. The old host first suspends the migrating process. The old host transmits a process migration approval request message. The new host responds with an approval or rejection message. The old host marshals and transfers the process control and execution state. The old host ships the communication links and any buffered messages. The old host transfers the file descriptors and dirty file cache blocks. The old host discards the clean file cache blocks. The old host ships all of the code, heap, and stack pages. Upon completion the old host transmits a message telling the new host to resume execution of the migrated process. Figure 5.1 graphically displays the data exchange sequence for the Total Copy algorithm.

As the Total Copy algorithm transfers all of the process state prior to resuming execution
on the new host, there are many valid orderings for transferring the components of the process state.

The *Total Copy* algorithm has one major flaw, and that is the slow transfer time. Even relatively small processes have address spaces that stretch over a 100kB, and large processes extend into the millions of bytes. The transfer time is particularly important to processes that communicate with other processes. During the transfer period the migrating process is unable to accept messages. Communication systems use timeout periods to detect failed message receivers. The long process migration transfer time would appear to the communication system as a failed receiver, and result in a communication failure. The long transfer times add overhead to the use of process migration for load balancing, and thus reduce its utility. The *Total Copy* algorithm eliminates residual dependencies, and thus can support processor fault tolerance.

### 5.2 *Pre–Copy* Algorithm

Theimer invented the *Pre–Copy* algorithm for the V operating system in order to overcome the communication failures that occur in the *Total Copy* algorithm [TLC85][The86]. This is important in V, because of its message based design. The Mach OMS process migration system implemented the *Pre–Copy* algorithm as one of its transfer strategy options [MZDG93][MGZ93].

The *Pre–Copy* algorithm transfers address space to the new host in parallel with continued execution of the migrating process on the old host. The migrating process
modifies address space pages by its continued execution. The Pre-Copy algorithm transfers modified pages until the number of modified pages falls below a small threshold value. The Pre-Copy algorithm then suspends the migrating process and transfers all remaining modified pages and the other process state components. The Pre-Copy algorithm freezes message processing during the suspension of the migrating process. This approach dramatically reduces the message freeze time, because of the large reduction in process state transferred during the message freeze time.

A generalized version of the Pre-Copy algorithm follows. At process migration time the migrating process continues execution. The old host transmits a process migration approval request message. The new host responds with an approval or rejection message. The old host ships all of the code, heap, and stack pages. Next the old host compares a predetermined limit against the number of pages modified by the continuing execution of the migrating process. If the number of modified pages exceeds the limit, the old host ships all of the modified heap and stack pages. Code pages are not normally modified. However, should the migrating process modify any code pages, the old host would ship the modified code pages. The old host repeats the cycle of shipping modified pages until the number of modified pages is less than some predetermined limit. At that time the old host suspends the migrating process. The old host marshals and transfers the process control and execution states. The old host ships the communication links and any buffered messages. The old host transfers the file descriptors and dirty file cache blocks. The old host discards the clean file cache blocks. The old host ships the remaining dirty heap and dirty stack pages. Upon completion the old host transmits a message telling the
new host to resume execution of the migrated process. Figure 5.2 graphically displays the data exchange sequence for the Pre-Copy algorithm.

The Pre-Copy algorithm reduces the message freeze time such that it eliminates process migration caused communication failures. The message freeze time is still substantial. Every modified page transfer represents the transfer of a page that the Pre-Copy algorithm has already transferred, and constitutes additional work not present in the Total Copy algorithm. Because of the additional work the Pre-Copy algorithm requires more total time than the Total Copy algorithm. The increased overhead detracts from the ability of process migration to support load balancing.

5.3 Demand Page Algorithm

Zayas implemented process migration in Accent using the copy on reference method [Zay87a] to transfer the address space. Lazy copying [Zay87b] refers to the same method. Copy on reference applies the much older time shared operating system concept of Demand Paging to process migration. At process migration time the address space remains on the old host. The new host demand pages the address space data as it is required. AIX/TCF demand pages unmodified code pages. Distributed Virtual Memory (DVM) accomplishes essentially the same form of address space transfer. The Mach OMS and SMS both use DVM for address space transfer [MZDG93][MGZ93]. Mess also uses DVM for address space transfer [DEGM90].
One generalized version of the Demand Page algorithm follows. This generalized version differs from the original Accent version in that Accent accesses the file system via messages [Zay87b, p. 18] and Zayas make no mention of a file cache. The old host first suspends the migrating process. The old host transmits a process migration approval request message. The new host responds with an approval or rejection message. The old host marshals and transfers the process control and execution state along with information about the address space. The system does not transfer the actual code, heap, and stack pages at this time. The old host ships the communication links and any buffered messages. The old host transfers the file descriptors and dirty file cache blocks. The old host discards the clean file cache blocks. Next the old host transmits a message telling the new host to resume execution of the migrated process. The migrating process page faults immediately because the migrating process does not have the needed code page. The new host transmits a message requesting the specific code page, and the old host responds by transmitting the needed page. Very shortly the new host page faults on both a stack page and a heap page. Almost all realistic programs require at least one code, one heap, and one stack page for meaningful execution. Figure 5.3 graphically displays the data exchange sequence for the Demand Page algorithm.

The Demand Page algorithm dramatically reduces the amount of data shipped during the process migration latency period. The Demand Page algorithm generally has a shorter message freeze time than the Pre-Copy algorithm. The Demand Page algorithm eliminates the transfer of address space pages that are never used after migration. Zayas found that on average this approach transfers 58% fewer bytes and reduces message
processing times by 47% [Zay87a, p. 23]. The reductions in both the process migration latency period and the overall amount of data transferred by a process migration contribute to reducing process migration overhead, and boost its utility for load balancing. The major disadvantage of the Demand Page algorithm is that the old host must maintain the address space until the migrating process completes execution. Should the process migrate again, another machine must maintain address space information. In the worst case scenario, a page fault occurring after a process migrates multiple times causes a search for the missing page on every machine involved in the past process migrations. The address space residual dependency prevents the Demand Page algorithm from supporting fault tolerance.

5.4 File Server Algorithm

All of the previous process migration algorithms involved exactly two machines: the old host and the new host. Dougis expanded the process migration possibilities by adding the file server machine as a third machine. The goal of Sprite’s process migration mechanism is to achieve most of the efficiency of the Demand Page algorithm without the address space residual dependency. The Sprite file system, which is a distributed file system, manages all files, including temporary files for backing store. After suspending the migrating process, Sprite flushes the dirty memory pages and dirty file cache blocks to the file server. The new host demand pages the file blocks and memory pages from the file server. Since Sprite applications are always dependent on the Sprite file
server, this approach does not add a new machine dependency. Sprite process migration
[DO87][Dou89][DO89][Dou90][DO91] swaps the cost of the file server flush for the elim-
ination of the address space residual dependency. Sprite does not eliminate all residual
dependencies on the old host machine, because the system forwards some system calls to
the home machine for execution, such as the system time call.

As Douglois did not publish a name for his process migration algorithm and the use of
the file server represents its most important contribution, this thesis uses the name File
Server algorithm for the generic algorithm. Sprite differs from the File Server algorithm
because Sprite applications communicate through the file system instead of via messages.

A generic version of the File Server algorithm follows. The system first suspends the
migrating process. The old host transmits a process migration approval request message.
The new host responds with an approval or rejection message. The old host marshals
and transfers the process control and execution state along with information about the
address space to the new host. The actual code, heap, and stack pages are not transmitted
at this time. The old host ships the communication links and any buffered messages to
the new host. The old host transfers the file descriptors to the new host. Then the old
host flushes the dirty stack pages, dirty heap pages, and dirty file cache blocks to the
file server. The old host discards the clean pages and clean file cache blocks. Next the
old host transmits a message telling the new host to resume execution of the migrated
process. The migrating process page faults immediately because the migrating process
does not have the needed code page. The new host transmits a message requesting the
specific code page, and the old host responds by transmitting the needed page. Very
shortly the new host page faults on both a stack page and a heap page. Almost all realistic programs require at least one code, one heap, and one stack page for meaningful execution. Figure 5.4 graphically displays the data exchange sequence for the File Server algorithm.

Sprite takes up to several seconds to migrate a process with many dirty pages and file blocks and several open files [Dou89, p. 59]. Sprite does not worry about message freeze time, because Sprite applications do not use messages and the file system based communications has been implemented in a way that is not time sensitive. In the conclusion to a study by the system developers comparing Sprite and Amoeba, the system developers report that the message based communications of Amoeba is simpler and has better performance than Sprite’s file based communications [DOKT91]. Therefore file based communication will not replace message based communication. The File Server algorithm has potentially long message freeze times, and thus the File Server algorithm will cause communication failures in a message based system.

The File Server algorithm transfers dirty memory pages and dirty file cache blocks first to the file server and later to the new host. The data transfers to the file server represent overhead not present in the other algorithms, as well as a further burden on the use of process migration to support load balancing. The actual process migration latency performance of Sprite ranks at the top among older systems (excluding Mach OMS) for a 100kB size process, as shown in table 3.1.
5.5 Summary

This chapter has examined the primary process migration algorithms. Despite being the first and most widely used algorithm, the Total Copy algorithm has unacceptably long process migration latency times and message freeze times. The Pre-Copy algorithm reduces message freeze time below where communication failures occur. Yet, the Pre-Copy algorithm achieves that goal at the cost of longer process migration latency times and a greater amount of total work. The Demand Page algorithm does not physically transfer the address space as part of the process migration latency operation, and thereby shortens dramatically both the process migration latency time and the message freeze time. The Demand Page algorithm demand pages needed pages from the old host. The address space dependency on the old host can exist indefinitely, and the Demand Page algorithm loses the ability to support processor fault tolerance. The File Server algorithm adds a file server support. At process migration time, the File Server algorithm flushes modified data to the file server and then acts like the Demand Page algorithm. The File Server algorithm trades off a flush operation for the elimination of residual dependencies. While the normal case has acceptable process migration latency and message freeze times, the worst cases are unacceptable. The next chapter explores how to improve on the existing algorithms.
Total Copy Algorithm

Old Host | New Host
---|---
Request Migration Approval
Approved Migration Request
Process Control and Execution State
Communication Links
File Info
Code Pages
Stack Pages
Heap Pages
Resume Process Command
Useful Execution

Figure 5.1: Total Copy Algorithm
Figure 5.2: Pre-Copy Algorithm
Demand Page Algorithm

Old Host

Request Migration Approval

Approved Migration Request

Process Control and Execution State

Communication Links

File Info

Resume Process Command

Request Code Page

Code Page

Request Stack Page

Stack Page

Request Heap Page

Heap Page

New Host

Useful Execution

Figure 5.3: Demand Page Algorithm
Figure 5.4: File Server Algorithm
Chapter 6

Performance Improvement

Opportunities

This chapter identifies performance bottlenecks in existing process migration algorithms and shows how to overcome these limitations. Faster processor and network speeds reduce both the process migration times and process execution times. Therefore this thesis places the emphasis on algorithmic changes for improvements in process migration. Separate sections identify performance improvement changes in the protocol controlling information transfer, the address space transfer, the message management, the file management, the object allocation, and the process modularity.
6.1 Protocol

Existing systems send a message from the old host to the new host requesting approval for a process migration, and then wait for the response. A better approach eliminates the approval request message. The Freeze Free algorithm uses the process control and execution state message to both transfer state and by implication signal the start of a process migration to the new host. The new host still responds with approval or rejection. Meanwhile the old host assumes acceptance and continues the shipment of data. The Freeze Free algorithm approach eliminates waiting time.

The new host can occasionally reject a process migration or simply fail. The Freeze Free algorithm protects against data loss by not discarding data on the old host until the new host confirms receipt. If the old host does not receive a positive confirmation, the old host migrates the process elsewhere or simply executes the process locally when resources become available.

Newer communication gear is steadily appearing with increased data bandwidth. While switching circuits are reducing switching times, the laws of physics limit both the time an electron flows through a wire and the time a photon passes through an optical fiber. Thus round trip times in physically distributed systems will not decrease much, while the bandwidth increases dramatically. These facts justify moving to a protocol that does not rely upon request and response messages. The Freeze Free algorithm blasts the process migration information, and uses request messages only when some unit of infor-
mation is needed out of order. The blast protocol does not require acknowledgements for each message [Mul93, p. 233], which reduces communication overhead.

6.2 Address Space

Existing process migration algorithms either transfer the entire address space or rely on demand paging. Transferring the entire address space results in unacceptably long process migration latency periods. Demand paging the address space requires request and response messaging for each page, which results in a waiting period per page. Zayas examined transferring the working set of pages at process migration time and then demand paging the remaining pages. The Zayas performance measurements show that the system achieves better performance by demand paging all of the pages [Zay87b]. As network and disk access speeds differ, it is not surprising that demand paging across a network differs somewhat from demand paging from disk.

All of the existing process migration algorithms have overlooked one other possibility. At process migration time, the system can always determine the immediately needed code and stack pages, and can usually determine the needed heap page. At any moment the program counter holds the address that the computer accesses for the next instruction. Similarly the stack pointer holds the address of the top of the stack, which is at the top of the current frame. The Freeze Free algorithm determines the heap page by examining the instruction stream for instructions that load or store registers. It is possible for a pathological program to possess complex branching across large distances. On
those rare occasions when the Freeze Free algorithm cannot easily determine the heap address, the Freeze Free algorithm simply resorts to demand paging all of the heap pages. Since shipping the wrong page does not cause a failure and also does not incur a large performance penalty, the Freeze Free algorithm uses a heuristic to determine the heap address. Since the Freeze Free algorithm only needs to determine the correct page, the Freeze Free algorithm takes advantage of program locality. The Freeze Free algorithm uses the first heap address and ignore branches. The heuristic is fast, simple, and effective. The heuristic requires 12 µseconds to determine the heap address for the test program. The most important advantage of this approach is that a program can perform useful execution with the correct code, the correct stack, and the correct heap page. On the other hand, no program can accomplish much until it does have these three pages.

Under this approach, the system does not have to transfer the remaining address space pages during the process migration latency period. The Freeze Free algorithm flushes the modified pages to the file server using a blast protocol. The blast occurs in parallel with the execution of the migrated process on the new host. Distributed file systems will become a standard component of distributed systems. Until then, in the absence of a file server, the Freeze Free algorithm blasts the data to the new host.

This approach eliminates address space dependencies on the old host when the overall process migration completes. Machine fault tolerance is an important goal of dynamic process migration.

The stack has the special property that the memory past the current top of the stack is not currently being used by the process. The Freeze Free algorithm discards the bytes
on the stack page past the current top of the stack. The process migration system reduces the current stack page to 528 bytes for the test application process, which represents an 87.5% savings on a 4kB page size system.

6.3 Messages

All existing process migration algorithms freeze message processing for some portion of the process migration latency period. The message freeze time on most existing systems lasts from process suspension until process resumption. The message freeze time complicates the communication subsystem, because the message freeze requires the communication subsystem to either buffer messages, or drop messages during the freeze and rely on error recovery.

Most systems do not separate the process state from the state of the communication links. EMPS [vDvG92] places the communication link in a Mailbox separate from the process. When the migrating process is not accessing communication functions, EMPS migrates a process without freezing message processing. However, EMPS freezes message processing during the process migration latency period if the process has any active message activity, such as a wait for message operation. Since the EMPS message freeze time in the worst case extends from process suspension to resumption, EMPS faces the same complications for the communication subsystem that other systems face.

The migrating process does not generate any outbound messages, because the process migration system suspends the migrating process during this period. However, other
programs remain active and can send messages to the migrating process. Upon message receipt, the communication subsystem modifies the process state to record the message receipt. Message receipt in the old systems affects more than just the message queue. The process migration system may have already marshalled the other affected data structures. So the process migration system freezes message processing until it has reconstructed the entire process state on the new host. The crux of the problem is the entanglement of the process state and the communication link state.

The Freeze Free algorithm design removes all communication link control state from the module containing the process control state. Conversely the Freeze Free algorithm design removes all process control state from the module containing the communication link control state. The Freeze Free algorithm design allocates a memory region within the address space of the process solely for the purpose of containing all of the communication state. These steps serve to isolate the process and communication link from one another. The Freeze Free algorithm design permits message receipt to proceed in parallel with process migration. The communication subsystem buffers each incoming message in the communication memory region and records each message in the communication message queue, which also resides in the communication memory region. A migrating process does not wait for a message arrival, so the communication subsystem does nothing to a process when a message arrives during process migration. When a message arrives and the message queue is full, the same error occurs regardless as to whether a process migration is underway. Process migration message freeze time is no longer dependent on the overall process migration.
There remains the issue of message processing support during the migration of the communication link. Just prior to shipping the communication link, the process migration system sets a flag which causes the rejection of all incoming messages. The process migration system immediately transfers the communication link, which in the current design fits in one message. The process migration system queues the communication link transfer message for transmission before the communication subsystem can reject an incoming message. The old host rejects messages arriving after the communication link transfer. The rejection notice contains the reason \textit{process migrated} and provides the new location. The communication subsystem that originated the message updates its link information and retransmits the message to the new host.

In rare situations, a retransmitted message arrives at the new host prior to the communication link transfer message. The new host communication subsystem blocks messages until the communication link arrives. The test system uses a communication subsystem built on a \textit{Choices} port [Lia95] of the $x$-Kernel [HP91]. $x$-Kernel uses one thread to process each message through the message protocol stack, and maintains a pool of threads. If one message thread blocks, overall message processing is unaffected. Message originators only receive rejection notices with new location information after the transmission of the communication link message. Thus the communication link transfer message is already in transit, and the new host wait is short. Therefore the message block time is short enough that the system treats it in the same manner as a page fault that fetches from disk the page containing the message queue. This approach effectively eliminates message freeze time.
The migrating process resumes execution without waiting for the communication link state. The migrating process could attempt to access messages prior to the arrival of the communication links, and the new host would block the migrating process. The process migration system automatically transfers the communication link information after shipping the stack pages. So the communication link information will arrive shortly without any need for a request message.

6.4 File Information

Distributed file systems, such as the Sprite file system [NWO88], cache file blocks to enhance performance. At process migration time, a process may have a large number of dirty file blocks in the file system cache. A simple approach transfers all of the cached dirty file blocks before resuming the migrating process. Unfortunately the simple approach yields long process migration latency periods. When many open files and dirty file blocks exist, Sprite reports process migration times measured in seconds [Dou89, p. 59]. Long process migration times hinder effective load balancing.

A better approach to handling the file cache is clearly needed. Eliminating the file cache is not a viable approach, because that would degrade application run time performance.

An examination of application file access patterns provides valuable clues to an effective solution. In 1985 Ousterhout et al analyzed the file system access patterns for Unix 4.2 BSD and found that file accesses were heavily sequential [OCH+85, Tab. 5]. In 1991
Baker et al investigated access patterns of the distributed file system in Sprite. They reported “If anything the sequentiality of access has increased over the last six years: our traces show that about 78% of all read-only accesses were sequential whole-file transfers, versus about 70% in the BSD study, and that more than 90% of all data was transferred sequentially, versus less than 70% in the BSD study” [BHK+91, p. 202].

These results have major significance for process migration. This means that fully written file blocks are unlikely to be written again. Therefore the process should not have to wait for the vast majority of the dirty file blocks to be flushed. The unchanged file blocks are already present on the file server. The Freeze Free algorithm design simply discards the unmodified file blocks on the old host at process migration time, and flushes the modified file blocks to the file server. The Freeze Free algorithm flushes the dirty file blocks after the migrating process resumes execution on the new host. Should the new host request a file block that is still present on the old host, the distributed file system orders the old host to transfer the block directly to the new host. Given the actual measured file access patterns, this situation does not occur for the overwhelming majority of file blocks.

The Freeze Free algorithm design applies the same separation that it applied to the communication link state and the process control state to the file descriptor state and the process control state. The file descriptor information contains no information about the process control state, and vice versa. The new design places the file descriptor information in its own memory region in the address space of the process.
The migrating process resumes execution without waiting for the file descriptor state. Should the migrating process attempt to access files prior to the arrival of the file descriptor information, the migrating process blocks as the system automatically ships the file descriptor state after the communication links and it will arrive shortly.

6.5 Object Allocation

Message based communication subsystems do not allocate a message sized chunk of memory every time a message arrives. Message based communication subsystems deposit incoming messages into already allocated buffers. None of the publications on existing process migration systems referenced in this thesis mention the use of any preallocated data structures. For process migration the Freeze Free algorithm preallocates a set of data structures and partially initializes the data structures. This technique moves the memory allocation and some of the initialization out of the process migration latency period. The preallocation and partial initialization occur at system startup and at times after a process migration completes. The motivation is that this allows process migration to take advantage of processor cycles at less critical time periods. The implemented process migration system preallocates data structures for one process, one domain, and six memory regions (command line arguments, code, heap, user stack, kernel stack, and message queue). Chapter 8 provides measurements showing substantial performance gains. At process migration time, the Freeze Free algorithm uses migrated information to complete initialization.
6.6 Process Modularity

Currently operating systems typically scatter process related information in many different locations throughout the operating system’s data structures. This violates the concept of modularity. The information dispersal makes the extraction and insertion of a task’s control information more difficult and takes longer. Zayas measured the time Accent expends extracting a process for eight sample programs [Zay87b, p. 74]. Dividing Zayas’ measurements of the overall migration time without host lookup by his measurements of the excision time produced the percentage time Accent spends in excision. Accent process migration spends 11% – 24% in excision, with an average of 17.5% in excision. The Zayas figures show that information dispersal is expensive. The typical process migration insertion operation scatters the process related information on the new host, resulting in another costly operation.

The Freeze Free algorithm design reorganizes the operating system data structures so that information about an object is contained strictly within that object. For example, the Process object contains all process control information, and the ApplicationMsgQueue contains all information about that particular application message queue. Conversely the Freeze Free algorithm design also takes information out of objects where it does not belong. The new ProcessManager can find Process objects, but the ProcessManager does not contain information about specific Process objects. The result is that the combined excision and insertion costs for process control state, execution state, and memory region information now take 506µs, which is 3.6% of the
process migration latency period (see section 8.3 for a breakdown by category of the process migration latency period activities). The excision and insertion costs are now a much smaller percentage of a far smaller latency period.

### 6.7 Summary

This chapter has identified six major impediments to process migration performance and provided techniques to overcome these existing limitations. The Freeze Free algorithm takes advantage of specific changes in protocol, address space transfer, message management, file management, object allocation, and process modularity.
Chapter 7

Freeze Free Algorithm

This chapter describes the new Freeze Free algorithm for process migration. The Freeze Free algorithm starts after the operating system orders the process migration subsystem to migrate a specific process from its old host machine to a specific new host machine. The Freeze Free algorithm targets a distributed system environment containing a distributed file system and applications that can communicate via messages. The Freeze Free algorithm supports process migration among homogeneous machines running under the same operating system. The Freeze Free algorithm is independent of the programming language and operates at the executable module level. The basic Freeze Free algorithm migrates a process with exactly one thread of control.

The chapter begins with a section describing the Freeze Free algorithm steps. The chapter follows with sections covering process migration latency, message handling, and both page faults and flush operations.
7.1 Algorithm Steps

At process migration time, the Freeze Free algorithm suspends the migrating process. The old host marshals the execution and process control state into a message and transmits it to the new host. The new host initializes an empty Process object, and responds with an acceptance or rejection message. Meanwhile without waiting the old host determines the current code page and ships it. The old host searches for the current heap page using a heuristic. If the old host determines the current heap page, the old host ships it. Next the old host determines the current stack page, and after discarding the bytes past the top of stack, ships it. When the new host has received and processed the first code page, the first heap page (optional), and the first stack page, the new host resumes the migrating process. The old host continues by shipping any remaining stack pages, followed by the communication links, and then the file descriptor information to the new host. Next the old host flushes to the file server the dirty heap pages, and then the dirty file cache pages. If an additional physical communication link exists between the old host and the file server that is separate from the communication link between the old host and the new host, then the old host flushes dirty pages to the file server right after process suspension in parallel with transfers to the new host. After all data has been transferred off the old host, the old host sends a flush complete message to the new host. That completes the overall process migration. Figure 7.1 graphically portrays the steps of the Freeze Free algorithm.
7.2 Process Migration Latency

The process migration latency period extends from process suspension till process resumption, which occurs after the new host receives the first stack page. The Freeze Free algorithm performs the same steps regardless of process address space size, number of open files, state of the message queue, or the number of dirty pages and file cache blocks. Three factors affect performance: the number of bytes truncated from the stack page, the presence of floating point, and whether the heap page is found. Thus the Freeze Free algorithm migrates even large processes quickly.

7.3 Message Handling

Until the old host ships the communication links, the old host receives messages. The new host holds any messages arriving while the communication links are in transit. After the communication links arrive on the new host, the new host accepts messages. The Freeze Free algorithm never stops message receipt, and message receipt is only delayed while the communication links are in transit. During the time the system suspends the migrating process, the migrating process does not wait for a message. Any message arriving during the process migration latency period becomes available after the migrating process resumes execution on the new host.
7.4 Page Faults and Flushing

After process migration, the migrating process knows which dirty pages reside on the old host. When a page fault occurs after process migration, the distributed file system knows whether the missing page resides on the old host or the file server, and the distributed file system requests the page directly from the appropriate machine. The flush complete message tells the new host that all pages are now available from the file server. At this point no dependencies on the old host exist, unless the old host is also the file server. The time for flushing is directly correlated with the number of dirty pages present on the old host.

7.5 Summary

This chapter has described the new Freeze Free process migration algorithm. The Freeze Free algorithm reduces the information sent during the process migration latency period to a minimum. A general purpose process migration system can not transfer less than the process control state, execution state, current code page, current heap page, and current stack page. The Freeze Free algorithm effectively eliminates message freeze time. The Freeze Free algorithm supports processor fault tolerance by quickly eliminating residual dependencies on the old host machine. Overall the Freeze Free algorithm is a significant advance over previous process migration algorithms.
Freeze Free Algorithm

File Server

Old Host

Process Control and Execution State

New Host

Code Page

Migration Accepted

Heap Page

Stack Page

Remaining Stack Page(s)

Useful Execution

Communication Links

File Info

Dirty Heap Page(s)

Dirty File Cache Page(s)

Flush Complete

Figure 7.1: Freeze Free Algorithm
Chapter 8

Performance Measurements

This chapter describes the experiments carried out to validate the process migration ideas, and reports the actual performance measurements. The chapter begins with a description of the process migration test system, and then covers a series of experiments. First, the experiments prove that the Freeze Free Algorithm works. Second, the experiments show that the Freeze Free Algorithm achieves a process migration latency time under 14ms. That time is an order of magnitude shorter than all previous process migration systems, and that time achieves the short process migration latency time identified in section 2.6.3. Third, the chapter reports performance measurements of the process migration steps, and analyzes the major costs. Fourth, the chapter reports performance measurements of the cross network demand page operations, and analyzes the major costs. Fifth, experiments demonstrate the utility of process migration for load sharing and the effects of process migration overhead on load sharing speedup.
8.1 Test System

Any general algorithm must be tailored to the actual target environment. This section describes the environment and actual process migration system implementation.

8.1.1 Hardware

The implementation platform is a pair of SPARC Station 2 [Tec90] work stations with a 64kB cache and 16MB main memory. The Lance AMD 7990 supports the connection to a 10M bit per second Ethernet LAN.

8.1.2 Operating System

Choices[CI93] release 5.19.94 is the host operating system. Choices is an object oriented operating system [CIRM93] built using C++[Str91] and a very small amount of assembly language. Choices models system resources and management entities as a collection of objects that represent policies, algorithms, mechanisms, and data representations [CIJ+91]. Objects are organized into frameworks, which are combined to produce the overall system.

8.1.3 Communication Subsystem

Choices supports several quite different communication subsystems. The process migration system uses the Choices port [Lia95] of the x-Kernel [HP91] system. An environment with migrating processes needs a message protocol with location transparent
addressing. The FLIP [KRST93] protocol provides location transparent addressing to Amoeba. Hutchinson and Peterson designed \( z \)-Kernel to facilitate the addition of new protocols. The process migration system implementation includes the implementation of a new protocol \texttt{ProcessMsg} that addresses messages by process identifiers on top of UDP using \( z \)-Kernel. The \texttt{ProcessMsg} message protocol works for message communications within a single processor and across the network. The \texttt{ProcessMsg} message also delivers signals to processes. For example, the system delivers the \textit{wakeUpProcess} signal using \texttt{ProcessMsg} messages, and the kernel performs the appropriate action on the blocked process.

The communication subsystem copies the \texttt{ProcessMsg} message data only once on transmission from kernel threads. The communication subsystem delivers the address of the \texttt{ProcessMsg} message data to the kernel thread without copying. Table 8.1 reports the round trip transfer time needed for one kernel thread to kernel thread \texttt{ProcessMsg} message across the Ethernet LAN. The various message data sizes correspond to the actual data sizes of messages used in process migration. The \textit{Final Copy Time} in table 8.1 shows the actual data copy times encountered by the \texttt{MigrateIn} daemon. The combined process control and execution state message is 352 bytes, and the \texttt{MigrateIn} daemon copies the process control and execution state information to a kernel resident data structure. The other messages deliver page data that requires the addition of a memory management unit address translation during the copy operation. The cost of adding one address translation accounts for the large time difference between copying 352 bytes and
<table>
<thead>
<tr>
<th>Size (Bytes)</th>
<th>Round Trip No Final Copy Time μsec</th>
<th>Final Copy Time μsec</th>
<th>Round Trip With Final Copy Time μsec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,798</td>
<td></td>
<td></td>
</tr>
<tr>
<td>352</td>
<td>2,358</td>
<td>66</td>
<td>2,424</td>
</tr>
<tr>
<td>528</td>
<td>2,524</td>
<td>553</td>
<td>3,077</td>
</tr>
<tr>
<td>4,096</td>
<td>8,970</td>
<td>1,786</td>
<td>10,756</td>
</tr>
<tr>
<td>8,192</td>
<td>16,047</td>
<td>2,390</td>
<td>18,437</td>
</tr>
<tr>
<td>16,384</td>
<td>30,205</td>
<td>4,638</td>
<td>34,843</td>
</tr>
</tbody>
</table>

Table 8.1: ProcessMsg Message Transfer Performance

528 bytes. ProcessMsg messages for application processes require an additional copy on transmit and another on receipt.

8.1.4 File System

The Choices file system [Mad92] supports a variety of file formats with a kernel level implementation. The Choices file system is not a distributed file system with location transparency. Another individual is working on providing file data across the network using a variety of flexible caching strategies. The new process migration system does not support migrating programs with open files at this time for two reasons. First, a working distributed file system with a unified name space and location transparency was not available. Secondly, the design has moved the file state transfer out of the critical path, which is the process migration latency period.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Time µsec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Thread Context Switch</td>
<td>46.1</td>
</tr>
<tr>
<td>lock acquire and release</td>
<td>.174</td>
</tr>
<tr>
<td>semaphore P + V</td>
<td>3.40</td>
</tr>
</tbody>
</table>

**Table 8.2:** Process Subsystem Performance

### 8.1.5 Process Subsystem

The *Choices* process [RJC88] is a single thread of control that executes in one virtual memory address space. *Choices* supports multiple processes in one virtual address space. The process migration implementation migrates a process that has exclusive access to its virtual memory address space.

The *Freeze Free* algorithm design disentangles the *Process* object from other system objects and localizes process related information within the *Process* object. The implementation effort included upgrading the process scheduler both to support process priorities and globally enforce a uniform scheduling policy. The implementation effort also included upgrading for performance reasons the process locking from a single heavyweight lock to a set of lightweight locks appropriate for different situations. The end result is almost a complete replacement and major performance improvements, which are reported in [RLC95]. The context switch, the combined acquire and release lock, and the combined non-blocking semaphore *P* and *V* kernel level performance times appear in table 8.2.
8.1.6 Application Operating System Interface

*Choices* provides an object oriented application interface to the operating system, which uses the ObjectProxy [Rus91] to represent kernel objects and extend inheritance to application programs. The ObjectProxy contains the kernel address of the kernel object that is represented by the ObjectProxy. The specific kernel address may already be used on the new host. The system cannot use the existing ObjectProxy without expensive translations. Extending inheritance across the kernel – application boundary also extends compilation dependencies. For these reasons the process migration system does not migrate ObjectProxies. The system dependencies on ObjectProxies are quite extensive. The process migration system recreates the StandardSystemInterface, StandardOutput, and StandardInput proxies after process migration. In other areas the new design eliminates the ObjectProxy. For example, the new design supports access to Process objects through the use of location independent process identifiers. This experience leads to a recommendation against the use of location dependent identifiers in distributed systems.

8.1.7 Process Migration Subsystem

The principal components of the Process Migration Subsystem are two daemons. The MigrateOut daemon is a kernel thread supporting outbound process migration, and the MigrateIn daemon is a kernel thread supporting inbound process migration. The two daemons use the new ProcessMsg message protocol for communications. A Process object does not exist in complete isolation. The Process object requires several support-
ing objects. For example, the Domain object identifies the memory regions supporting the process, and the ProcessorContext object contains the execution state of a process. In keeping with the object oriented design methodology, the Freeze Free algorithm design adds methods to the supporting objects for extracting and inserting their own state data. This approach facilitated the process migration implementation. Sprite uses callbacks into modules to encapsulate data, and reported that this approach was easier to implement, maintain, and port than an earlier approach that performed these functions from a central location [Dou89, p. 67].

8.2 Process Migration Performance

This section first reports the overall process migration latency performance, and then describes the costs of the process migration steps on the old host and the new host.

8.2.1 Process Migration Latency

The first experiment demonstrates very short process migration latency times using an application that sends and receives messages. The process migration system performs the process migration latency period in 13.89ms using a 4kB page size. The system design makes page size an option. The process migration system performs the process migration latency period in 20.819ms using an 8kB page size and in 36.894ms using a 16kB page size. Only three items cause a variance in the process migration latency time. Floating point registers are not used by all processes. Occasionally it is possible that
the heuristic will fail to find a heap page address, and then the system demand pages all heap pages. The degree to which the system can truncate the top stack page varies. Therefore these times apply to processes of any size with only time variations for the factors just identified.

8.2.2 Migrate Out Steps

The first step to improving process migration is learning what it does. This section details the steps performed by the MigrateOut daemon during the process migration latency period.

The timing begins once the SystemInterface object receives the application process migration request. During the Start Daemon step the system blocks the application process and context switches to the MigrateOut daemon. The MigrateOut daemon initializes during the Init step. The MigrateOut daemon adds the application stack to the kernel Domain during the Add Stack step. The MigrateOut daemon extracts the process control information during the Get Process step, the memory region information during the Get Domain step, and the register state during the Get Regs step. The MigrateOut daemon processes the first message through the communication protocol stack during the 1st Msg step. The MigrateOut daemon adds the application heap to the kernel Domain during the Add Heap step, finds the current application heap address during the Get Heap Addr step, and processes the first heap page message through the communication protocol stack during the Heap Page Msg step. The MigrateOut daemon
processes the first stack page through the communication protocol stack during the *Stack Msg* step, and removes the stack from the kernel *Domain* during the *Remove Stack* step.

The *MigrateIn* daemon processes the first message and responds with an acceptance message. The *MigrateIn* daemon processes the code, heap, and stack page information and responds with a stop timer message. The *MigrateIn* then enqueues the migrating application process for execution.

The *MigrateOut* daemon waits for the accept migration message during the *Accept Msg* step. The *MigrateOut* daemon waits for the stop timer message during the *Stop Msg* *Wait* step. The system uses the stop timer message only during the process migration latency period timing tests to signal the old host that the new host has resumed the migrating process. Because the stop timer message is not part of the actual process migration system, subtracting the cost of the delivering the stop timer message from the stop message arrival time yields the actual process migration latency time. The process migration performance numbers for 4kB page size appear in table 8.3, for 8kB page size they appear in table 8.4, and for 16kB page size they appear in table 8.5.

### 8.2.3 Migrate In Steps

Messages from the *MigrateOut* daemon drive the operations of the *MigrateIn* daemon on the new host. The *MigrateIn* daemon receives four messages during the process migration latency period.

Upon the receipt of the first process migration message, the *MigrateIn* daemon prepares for a new process migration. During the *Data Copy* step, the *MigrateIn* copies the
<table>
<thead>
<tr>
<th>Step</th>
<th>Time $\mu$s</th>
<th>Cumulative Time $\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Daemon</td>
<td>139</td>
<td>139</td>
</tr>
<tr>
<td>Init</td>
<td>5</td>
<td>144</td>
</tr>
<tr>
<td>Add Stack</td>
<td>110</td>
<td>254</td>
</tr>
<tr>
<td>Get Process</td>
<td>21</td>
<td>275</td>
</tr>
<tr>
<td>Get Domain</td>
<td>21</td>
<td>290</td>
</tr>
<tr>
<td>Get Regs</td>
<td>425</td>
<td>721</td>
</tr>
<tr>
<td>1st Msg</td>
<td>468</td>
<td>1,189</td>
</tr>
<tr>
<td>Add Code</td>
<td>111</td>
<td>1,300</td>
</tr>
<tr>
<td>Code Page Msg</td>
<td>1,713</td>
<td>3,013</td>
</tr>
<tr>
<td>Add Heap</td>
<td>110</td>
<td>3,123</td>
</tr>
<tr>
<td>Get Heap Addr</td>
<td>11</td>
<td>3,134</td>
</tr>
<tr>
<td>Heap Page Msg</td>
<td>1,762</td>
<td>4,896</td>
</tr>
<tr>
<td>Stack Msg</td>
<td>398</td>
<td>5,294</td>
</tr>
<tr>
<td>Remove Stack</td>
<td>448</td>
<td>5,743</td>
</tr>
<tr>
<td>Enqueue</td>
<td>1</td>
<td>5,744</td>
</tr>
<tr>
<td>Accept Msg</td>
<td>4,565</td>
<td>10,309</td>
</tr>
<tr>
<td>Stop Msg Wait</td>
<td>4,438</td>
<td>14,747</td>
</tr>
<tr>
<td>Stop Msg Cost</td>
<td>-857</td>
<td></td>
</tr>
<tr>
<td>Total Time</td>
<td></td>
<td>13,890</td>
</tr>
</tbody>
</table>

**Table 8.3:** Migrate Out 4kB Page
<table>
<thead>
<tr>
<th>Step</th>
<th>Time $\mu$s</th>
<th>Cumulative Time $\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Daemon</td>
<td>147</td>
<td>147</td>
</tr>
<tr>
<td>Init</td>
<td>5</td>
<td>152</td>
</tr>
<tr>
<td>Add Stack</td>
<td>108</td>
<td>260</td>
</tr>
<tr>
<td>Get Process</td>
<td>23</td>
<td>283</td>
</tr>
<tr>
<td>Get Domain</td>
<td>20</td>
<td>303</td>
</tr>
<tr>
<td>Get Regs</td>
<td>407</td>
<td>710</td>
</tr>
<tr>
<td>1st Msg</td>
<td>468</td>
<td>1,178</td>
</tr>
<tr>
<td>Add Code</td>
<td>110</td>
<td>1,288</td>
</tr>
<tr>
<td>Code Page Msg</td>
<td>2,820</td>
<td>4,108</td>
</tr>
<tr>
<td>Add Heap</td>
<td>112</td>
<td>4,220</td>
</tr>
<tr>
<td>Get Heap Addr</td>
<td>12</td>
<td>4,232</td>
</tr>
<tr>
<td>Heap Page Msg</td>
<td>2,884</td>
<td>7,116</td>
</tr>
<tr>
<td>Stack Msg</td>
<td>394</td>
<td>7,510</td>
</tr>
<tr>
<td>Remove Stack</td>
<td>444</td>
<td>7,954</td>
</tr>
<tr>
<td>Enqueue</td>
<td>1</td>
<td>7,955</td>
</tr>
<tr>
<td>Accept Msg</td>
<td>9,206</td>
<td>17,161</td>
</tr>
<tr>
<td>Stop Msg Wait</td>
<td>4,511</td>
<td>21,672</td>
</tr>
<tr>
<td>Stop Msg Cost</td>
<td>-853</td>
<td></td>
</tr>
<tr>
<td>Total Time</td>
<td></td>
<td>20,819</td>
</tr>
</tbody>
</table>

**Table 8.4:** Migrate Out 8kB Page
<table>
<thead>
<tr>
<th>Step</th>
<th>Time $\mu$s</th>
<th>Cumulative Time $\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Daemon</td>
<td>135</td>
<td>135</td>
</tr>
<tr>
<td>Init</td>
<td>5</td>
<td>140</td>
</tr>
<tr>
<td>Add Stack</td>
<td>109</td>
<td>249</td>
</tr>
<tr>
<td>Get Process</td>
<td>23</td>
<td>272</td>
</tr>
<tr>
<td>Get Domain</td>
<td>20</td>
<td>292</td>
</tr>
<tr>
<td>Get Regs</td>
<td>410</td>
<td>702</td>
</tr>
<tr>
<td>1st Msg</td>
<td>469</td>
<td>1,171</td>
</tr>
<tr>
<td>Add Code</td>
<td>112</td>
<td>1,283</td>
</tr>
<tr>
<td>Code Page Msg</td>
<td>5,459</td>
<td>6,742</td>
</tr>
<tr>
<td>Add Heap</td>
<td>124</td>
<td>6,866</td>
</tr>
<tr>
<td>Get Heap Addr</td>
<td>12</td>
<td>6,878</td>
</tr>
<tr>
<td>Heap Page Msg</td>
<td>5,516</td>
<td>12,394</td>
</tr>
<tr>
<td>Stack Msg</td>
<td>369</td>
<td>12,763</td>
</tr>
<tr>
<td>Remove Stack</td>
<td>417</td>
<td>13,180</td>
</tr>
<tr>
<td>Enqueue</td>
<td>1</td>
<td>13,181</td>
</tr>
<tr>
<td>Accept Msg</td>
<td>17,658</td>
<td>30,839</td>
</tr>
<tr>
<td>Stop Msg Wait</td>
<td>6,921</td>
<td>37,760</td>
</tr>
<tr>
<td>Stop Msg Cost</td>
<td>-866</td>
<td></td>
</tr>
<tr>
<td><strong>Total Time</strong></td>
<td></td>
<td><strong>36,894</strong></td>
</tr>
</tbody>
</table>

**Table 8.5:** Migrate Out 16kB Page
first message data into a local buffer. The **MigrateIn** daemon adds the application code memory region to the application **Domain** during the **Add Code User** step, the application heap memory region during the **Add Heap User** step, and the application command line argument memory region during the **Add Args** step. Next the **MigrateIn** daemon initializes the process control information in the **Process** object during the **Init Process** step. Finally, the **MigrateIn** daemon queues an accept message for transmission during the **Accept Msg** step.

Upon the receipt of the code message, the **MigrateIn** daemon first adds the application code memory region to the kernel **Domain** during the **Add Code Kernel** step. Then the **MigrateIn** daemon loads the code page during the **Copy Code Page** step.

Upon the receipt of the heap message, the **MigrateIn** daemon first adds the application heap memory region to the kernel **Domain** during the **Add Heap Kernel** step. Then the **MigrateIn** daemon loads the heap page during the **Copy Heap Page** step.

Upon the receipt of the stack message, the **MigrateIn** daemon first adds the application stack memory region to the kernel **Domain** during the **Add Stack Kernel** step. Then the **MigrateIn** daemon loads the stack page during the **Copy Stack Page** step.

The **MigrateIn** daemon performance measurements for 4kB page size appear in table 8.6, for 8kB page size they appear in table 8.7, and for 16kB page size they appear in table 8.8.

The **MigrateIn** daemon maintains a pool of objects for use at process migration time. The **MigrateIn** daemon creates one empty **Domain** object, one empty **Process** object, four backing store objects for memory regions, one empty user stack, and one kernel
<table>
<thead>
<tr>
<th>Step</th>
<th>Time $\mu$s</th>
<th>Cumulative Time $\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Copy</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>Add Code User</td>
<td>181</td>
<td>233</td>
</tr>
<tr>
<td>Add Heap User</td>
<td>76</td>
<td>309</td>
</tr>
<tr>
<td>Add Args</td>
<td>81</td>
<td>390</td>
</tr>
<tr>
<td>Init Process</td>
<td>3,314</td>
<td>3,704</td>
</tr>
<tr>
<td>Load Regs</td>
<td>18</td>
<td>3,722</td>
</tr>
<tr>
<td>Accept Msg</td>
<td>479</td>
<td>4,201</td>
</tr>
<tr>
<td>Code Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Code Kernel</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>Copy Code Page</td>
<td>1,802</td>
<td>1,925</td>
</tr>
<tr>
<td>Heap Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Heap Kernel</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>Copy Heap Page</td>
<td>1,683</td>
<td>1,787</td>
</tr>
<tr>
<td>Stack Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Stack Kernel</td>
<td>116</td>
<td>116</td>
</tr>
<tr>
<td>Copy Stack Page</td>
<td>543</td>
<td>659</td>
</tr>
<tr>
<td>Remove Stack Kernel</td>
<td>408</td>
<td>1,067</td>
</tr>
</tbody>
</table>

**Table 8.6:** Migrate In Messages 4kB Page
<table>
<thead>
<tr>
<th>Step</th>
<th>Time $\mu$s</th>
<th>Cumulative Time $\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Copy</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>Add Code User</td>
<td>179</td>
<td>230</td>
</tr>
<tr>
<td>Add Heap User</td>
<td>80</td>
<td>310</td>
</tr>
<tr>
<td>Add Args</td>
<td>84</td>
<td>394</td>
</tr>
<tr>
<td>Init Process</td>
<td>3,320</td>
<td>3,714</td>
</tr>
<tr>
<td>Load Regs</td>
<td>18</td>
<td>3,732</td>
</tr>
<tr>
<td>Accept Msg</td>
<td>473</td>
<td>4,205</td>
</tr>
<tr>
<td>Code Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Code Kernel</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>Copy Code Page</td>
<td>2,351</td>
<td>2,474</td>
</tr>
<tr>
<td>Heap Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Heap Kernel</td>
<td>108</td>
<td>108</td>
</tr>
<tr>
<td>Copy Heap Page</td>
<td>2,273</td>
<td>2,381</td>
</tr>
<tr>
<td>Stack Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Stack Kernel</td>
<td>121</td>
<td>121</td>
</tr>
<tr>
<td>Copy Stack Page</td>
<td>550</td>
<td>671</td>
</tr>
<tr>
<td>Remove Stack Kernel</td>
<td>405</td>
<td>1,076</td>
</tr>
</tbody>
</table>

Table 8.7: Migrate In Messages 8kB Page
<table>
<thead>
<tr>
<th>Step</th>
<th>Time $\mu$s</th>
<th>Cumulative Time $\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Copy</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>Add Code User</td>
<td>179</td>
<td>230</td>
</tr>
<tr>
<td>Add Heap User</td>
<td>80</td>
<td>310</td>
</tr>
<tr>
<td>Add Args</td>
<td>83</td>
<td>393</td>
</tr>
<tr>
<td>Init Process</td>
<td>3,319</td>
<td>3,712</td>
</tr>
<tr>
<td>Load Regs</td>
<td>18</td>
<td>3,730</td>
</tr>
<tr>
<td>Accept Msg</td>
<td>472</td>
<td>4,202</td>
</tr>
<tr>
<td>Code Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Code Kernel</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>Copy Code Page</td>
<td>4,618</td>
<td>4,743</td>
</tr>
<tr>
<td>Heap Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Heap Kernel</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>Copy Heap Page</td>
<td>4,491</td>
<td>4,614</td>
</tr>
<tr>
<td>Stack Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Stack Kernel</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>Copy Stack Page</td>
<td>553</td>
<td>679</td>
</tr>
<tr>
<td>Remove Stack Kernel</td>
<td>407</td>
<td>1,086</td>
</tr>
</tbody>
</table>

**Table 8.8**: Migrate In Messages 16kB Page
<table>
<thead>
<tr>
<th>Step</th>
<th>Time µs</th>
<th>Cumulative Time µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Domain</td>
<td>7,557</td>
<td>7,557</td>
</tr>
<tr>
<td>1 Process</td>
<td>90</td>
<td>7,647</td>
</tr>
<tr>
<td>4 Back Store</td>
<td>3,003</td>
<td>10,650</td>
</tr>
<tr>
<td>User Stack</td>
<td>56</td>
<td>10,706</td>
</tr>
<tr>
<td>Kernel Stack</td>
<td>673</td>
<td>11,379</td>
</tr>
</tbody>
</table>

**Table 8.9: Migrate In Object Pool**

stack. The MigrateIn daemon adds the kernel stack to the kernel domain. The time required to create these objects appears in table 8.9.

### 8.3 Process Migration Latency Period Analysis

This section first identifies the operations of the process migration latency period, and assigns the operation costs to major categories. Then the section examines the impact of four areas on the process migration latency period, and identifies performance improvement possibilities.

#### 8.3.1 Latency Period Critical Path

Process migration uses three principle resources: the old host processor, the new host processor, and the Ethernet LAN. Any one of these resources can potentially be a bottleneck. Figure 8.1 graphically displays the resource usage of these three resources over time by process migration with a 4kB page size. The old host is busy for 5,743 µs, the new host is busy for 11,421 µs, and Ethernet is busy 7,821 µs during the process migration latency period.
The process migration latency period divides naturally into four steps: process control and execution state, first code page, first heap page, and first stack page. The time measurements for the old host come from table 8.3. Adding the appropriate `ProcessMsg` header, UDP header, IP headers, Ethernet headers and trailers, and Ethernet stop gaps to the actual data length, and then dividing by 10M bits per second yields the Ethernet transmission time. The calculation does not account for possible Ethernet packet collisions. Router software periodically transmits routing messages, which can conflict slightly with process migration for Ethernet. Directly measuring the time spent by the x-Kernel processing an inbound message was not possible. A 4kB `ProcessMsg` message travels in 3 Ethernet messages. x-Kernel processes each inbound Ethernet message with a separate thread. A 4kB `ProcessMsg` requires at least 4 kernel thread context switch, which cost 46.1μs each. The processing of an inbound message up through the message protocol stack is on the same order of magnitude as processing an outbound message. The use of context switch and message protocol costs as a guide resulted in an estimate of 654μs for each message arriving on the new host. Subtracting both the outbound message protocol processing time and the transmission time from the round trip time yields a measure of the receive message protocol processing cost that is in close agreement with the prior estimate. The combination of the receive message processing cost and the `MigrateIn` daemon process migration costs from table 8.6 yield the the total time cost of each step on the new host. The net result is a rough, but useful picture of where the process migration system spends time during the process migration latency period.
Process Migration Latency Period Resource Usage

<table>
<thead>
<tr>
<th>Time</th>
<th>Old Host</th>
<th>Ethernet</th>
<th>New Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>State</td>
<td>State</td>
<td></td>
</tr>
<tr>
<td>1,189</td>
<td>Code</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>3,013</td>
<td>Heap</td>
<td>Heap</td>
<td></td>
</tr>
<tr>
<td>4,896</td>
<td>Stack</td>
<td>Stack</td>
<td></td>
</tr>
<tr>
<td>5,743</td>
<td>Ack Msg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,477</td>
<td></td>
<td>Begin Execution</td>
<td>13,890</td>
</tr>
</tbody>
</table>

**Figure 8.1:** Process Migration Resource Usage (time in $\mu$s)

Figure 8.1 clearly shows that the new host is the principal bottleneck in the current process migration system, and that the Ethernet LAN is a close second.

Table 8.10 reports the percentage of time on the critical path spent by each major activity category.

### 8.3.2 Extraction and Insertion

The extraction on the old host and the insertion on the new host of the process control state, the execution state, and the memory region information takes 506$\mu$s and 3.6% of
<table>
<thead>
<tr>
<th>Activity Category</th>
<th>Time $\mu$s</th>
<th>Per Cent of Critical Path Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Extract/Insert</td>
<td>506</td>
<td>3.6</td>
</tr>
<tr>
<td>Domain Add/Remove</td>
<td>1,199</td>
<td>8.6</td>
</tr>
<tr>
<td>Add Address Translations</td>
<td>1,308</td>
<td>9.4</td>
</tr>
<tr>
<td>Init Process</td>
<td>3,293</td>
<td>23.7</td>
</tr>
<tr>
<td>Data Copy</td>
<td>2,852</td>
<td>20.5</td>
</tr>
<tr>
<td>Msg Protocol Work</td>
<td>3,018</td>
<td>21.7</td>
</tr>
<tr>
<td>Transmission Time</td>
<td>1,337</td>
<td>9.6</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>377</td>
<td>2.9</td>
</tr>
</tbody>
</table>

**Table 8.10:** Major Activities During Critical Path

the process migration latency period. Accent excision alone on average takes 17.5% of the process migration latency period (see section 6.6). So clearly localizing information yields a noticeable performance improvement.

### 8.3.3 Virtual Memory

Adding and removing memory regions from address space domains on both hosts takes $1,199\mu$s and 8.6% of the process migration latency period. Adding memory management address translations on the new host takes $1,308\mu$s and 9.4% of the process migration latency period. These two virtual memory management operations take $2,507\mu$s and 18.0% of the process migration latency period. The `Process` object initialization takes $3,293\mu$s and 23.7% of the process migration latency period. The biggest part of the `Process` object initialization is the preparation of a zero filled memory region for the application stack. This is primarily a virtual memory operation. These three categories combined take 41.7% of the process migration latency period. Any further improvements
in process migration performance will require serious examination of the virtual memory support operations.

While the Choices virtual memory system is operational, there are substantial opportunities for improvement. One change to the operation that adds a memory region to the address space domain saved 3ms by using a better search heuristic.

A mismatch exists in the virtual memory subsystem between the class hierarchy and the functional dependencies. Another Choices project added the MemoryObjectView class to the bottom of the existing virtual memory class hierarchy to support mapping a selectable number of pages in a memory region. The parent class MemoryObject represents a memory region. Currently a MemoryObjectView must be created as part of adding a MemoryObject to a Domain, and destroyed upon the removal of the MemoryObject from a Domain. The system creates seven MemoryObjectViews and destroys one during the process migration latency period. A single Choices object creation with its needed memory allocation requires 50–90µs. Changing the class hierarchy to make class MemoryObject a subclass of MemoryObjectView would both eliminate these unnecessary object creations and harmonize the class hierarchy and functional dependency relationships.

Further opportunities for virtual memory performance enhancements exist. Others are already rewriting the virtual memory subsystem [CRT95].

8.3.4 Data Copying

Copying the data to and from message buffers on the old and new host takes 2,852µs and 20.5% of the process migration latency time. Current limitations require prior to
transmission a copy to contiguous memory of the various message headers (ProcessMsg, UDP, IP, and Ethernet), message data, and message trailer. A smarter network interface device could eliminate this copy on transmission by directly shipping headers, data, and trailer from different memory locations.

Almost all of the process migration related messages contain one page of data. Currently on receipt the MigrateIn daemon must copy the page data both to page align the data and to strip the message protocol wrapper. This leads to a recommendation for the development of a special protocol for page size transfers. A smart interface device would separate the message wrapper and deposit the page data into a physical page. The data would be delivered by remapping the physical page. This approach would eliminate the data copy on message receipt. Distributed Virtual Memory and File Transfers are among the other system services that would benefit from such a protocol.

### 8.3.5 Communication Cost

The system waiting time to transmit bits on the Ethernet LAN is 1.337µs and takes 9.6% of the process migration latency time. The bulk of the transmission time is hidden by the higher costs of creating and initializing objects on the new host. This has interesting implications. Faster network speeds will not yield large improvements. Replacing the 10M bit per second Ethernet LAN with a 100M bit per second ATM network while leaving everything else unchanged, would only yield an 8.6% reduction in the process migration latency time. Naturally the transmission time will become more important when savings occur in other areas. The message protocol handling takes 3.018µs and 21.7% of the
<table>
<thead>
<tr>
<th>Action</th>
<th>4kB Page</th>
<th>8kB Page</th>
<th>16kB Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Request</td>
<td>899</td>
<td>899</td>
<td>899</td>
</tr>
<tr>
<td>Page Transfer</td>
<td>4,485</td>
<td>8,023</td>
<td>15,102</td>
</tr>
<tr>
<td>Page Copy</td>
<td>1,802</td>
<td>2,351</td>
<td>4,618</td>
</tr>
<tr>
<td>Total Time</td>
<td>7,186</td>
<td>11,273</td>
<td>20,619</td>
</tr>
</tbody>
</table>

Table 8.11: Page Fault Overhead in μs

process migration latency time. The combined transmission time and message protocol handling take 4,355 μs and 31.3% percent of the process migration latency time.

8.4 Demand Page Analysis

Page faults on non-resident pages incur substantial overhead on the new host. First, a message is sent requesting the missing page, which takes 899 μs. The other host spends 11 μs selecting the correct page. Next the system takes 4,485 μs transporting the 4kB page to the new host. The new host spends 1,802 μs copying the message 4kB page data to the actual 4kB page. The new host then continues as if the page were simply missing the needed address translations. The 4kB page fault across the network adds 7,186 μs. Table 8.11 reports the times for 4kB, 8kB, and 16kB page sizes. During the performance tests the old host was idle, so no domain change occurs when servicing a page request. Domain changes would add overhead to the cross network page fault.

An alternative view of page fault overhead is the number of page faults that add one second of overhead. Table 8.12 shows that 139 – 4kB page faults, 89 – 8kB page faults, or 48 – 16kB pages faults add roughly one second of overhead.
<table>
<thead>
<tr>
<th>Action</th>
<th>4kB Page</th>
<th>8kB Page</th>
<th>16kB Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Faults/Second</td>
<td>139</td>
<td>89</td>
<td>48</td>
</tr>
</tbody>
</table>

**Table 8.12**: Page Faults Per Second

<table>
<thead>
<tr>
<th>Activity Category</th>
<th>Time $\mu$s</th>
<th>Per Cent of Critical Path Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Address Translation</td>
<td>436</td>
<td>6</td>
</tr>
<tr>
<td>Data Copy</td>
<td>1,366</td>
<td>19</td>
</tr>
<tr>
<td>Msg Protocol Work</td>
<td>1,939</td>
<td>27</td>
</tr>
<tr>
<td>Transmission Time</td>
<td>3,445</td>
<td>48</td>
</tr>
</tbody>
</table>

**Table 8.13**: Major Activities During Demand Page

Table 8.13 shows for a page fault across the network the absolute times and percentages spent in four major activities. The relative importance differs dramatically from the process migration latency period. The network transmission time dominates, and faster network speeds will be effective. The earlier recommendation about how to eliminate data copying (see section 8.3.4) applies here, and eliminating data copying would improve performance by 19%.

The messages containing a page are all larger than the maximum size supported by Ethernet, and are fragmented into multiple messages on transmit and reassembled on receipt. The z-Kernel assigns one thread to process each message fragment. The receive side has at a minimum a context switch to each z-Kernel thread plus a context switch back to the **MigrateIn** daemon. The system requires a minimum of four context switches for a 4kB page.
This leads to a design change recommendation for $x$-Kernel. The new design places incoming messages on a queue. A single thread processes sequentially all messages on the queue. The new design activates a new thread when the currently running thread blocks. The new design reduces the number of context switches to two. Traces of $x$-Kernel activity show that on Choices the $x$-Kernel message threads run sequentially, with only rare exceptions. The cost of one context switch is 46.1μs. The process migration system transfers 31 – 4kB pages for my example application. This proposed change would save approximately 2.86ms for this particular process migration. While the communication subsystem performance is respectable, further opportunities for performance enhancement exist, such as in line code expansion in the manner used by the Unix 4.3BSD system [LC95].

8.5 Load Sharing

The goal of this thesis is to demonstrate a better process migration mechanism. Load sharing is one important use of process migration. An experiment demonstrates the effectiveness of process migration. A parent process creates four identical child processes, and then measures the total execution time. Each child process executes a variant of the Lawrence Livermore Loops test for five seconds without performing input or output. The processes do not share anything. The system allows only one active process migration at a time. The old host also acts as the server for page data. Choices does not allow a program to be stopped at arbitrary points, because of resource management limitations. Thus the
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Time sec</th>
<th>Speedup</th>
<th>Migrations</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Migration</td>
<td>20.435</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>4kB Migration</td>
<td>13.352</td>
<td>1.53</td>
<td>2.5</td>
</tr>
<tr>
<td>8kB Migration</td>
<td>12.920</td>
<td>1.58</td>
<td>2.0</td>
</tr>
<tr>
<td>16kB Migration</td>
<td>12.624</td>
<td>1.62</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 8.14: Load Sharing Experiment Results

child processes periodically poll the system asking to migrate to another machine. The polling adds scheduling delay.

The application process execution takes 20.435s with process migration turned off. The remainder of the experiments turned on process migration. The 4kB page system migrates on average 2.5 processes, and executes the applications in 13.35s, for a speedup of 1.53. The 8kB page system migrates on average 2 processes, and executes the applications in 12.92s, for a speedup of 1.58. The 16kB page system migrates on average 3 processes, and executes the applications in 12.62s, for a speedup of 1.62. Table 8.14 summarizes the results. The 16kB system yields a 2.5% better speedup than the 8kB system, which in turn yields a 3.3% better speedup than the 4kB system. Figure 8.2 graphs the relationship between page size and speedup.

Next a series of experiments show the effect of additional overhead on overall performance. The experiments ran again for each page size with additional artificial delays ranging from .5 to 8 seconds. The experiment ran a buffer copy operation during the delay period to simulate an artificial load. Figure 8.3 charts the effect of additional overhead on speedup for 4kB systems with 2 process migrations. Figure 8.4 does the
Figure 8.2: Page Size Impact on Performance
same for 8kB systems with 2 process migrations, while figure 8.5 handles 16kB systems with 3 process migrations.

Both the process migration latency period and demand paging across the network add overhead to the execution of an application on a new host. The following formula defines the total cost overhead of process migration:

\[
TotalProcessMigrationOverhead = CostProcessMigrationLatency + NumberOfPageFaults \times CostNetworkPageFault
\]  

The total process migration overhead cost for the 4kB page size test system is in milliseconds:

\[
TotalProcessMigrationOverhead_{4kB} = 13.9 + 7.2 \times NumberOfPageFaults
\]  

The sample programs generates 29 page faults when executed on a 4kB page size system. According to the above formula, the sample program incurs a total process migration overhead of 228ms, of which 208.8 result from network page faults.

The following formula calculates the theoretical load sharing system speedup when using process migration for a two processor system with a workload split into two equal numbers of identical processes:

\[
Speedup = \frac{ExecutionTimeWithoutMigration}{AllOverhead + 0.5 \times ExecutionTimeWithoutMigration}
\]  

113
Overhead vs Performance

Figure 8.3: Added Overhead versus Performance with 4kB Pages
Figure 8.4: Added Overhead versus Performance with 8kB Pages
Figure 8.5: Added Overhead versus Performance with 16kB Pages
The combined process migration overhead and additional overhead values appear in figure 8.6 plotted against speedup. Theoretically complete overhead values appear in the same figure plotted against speedup. Figure 8.6 clearly shows that actual measured performance tracks the theoretical curve, and that there is significant additional overhead. The additional overhead lies in the scheduler and its policy. The application processes poll to request process migration. The polling introduces a delay between when the system should migrate and when the system actually begins a process migration. The new host lies idle while an application waits for page faults across the network. Migrating two applications would allow the processor to work on one application while the other waits for a page transfer. The third delay occurs on the old host. Choices schedules processes according to priority, but the scheduler does not preempt a process for a newly ready process with a higher priority. The scheduler allows the currently running process to complete its time slice. The process servicing demand page requests has a higher priority than application code, but it waits anyway, because of the lack of preemption. Priority based process preemption was not needed for testing the migration of a single process. However, the multiple application test shows that priority based process preemption is essential for effective load balancing. The delay for a demand page request repeats many times (29 times in this example), and the delays combine.

Figure 8.6 shows that the relationship between speedup and overhead is non-linear. For example, a speedup of 1.5 requires an overhead of 1.67s (33%), while a speedup of 1.8 requires an overhead under .56s (11%) for the 5 second application execution time. The actual process migration overhead of 223ms is 4.5% of the 5 second application
Figure 8.6: Combined Overhead versus Performance
execution time. Small overhead is essential for effective load balancing, and a constant unit reduction in overhead yields increasing units of speedup until overhead reaches zero. These results agree with the general direction of Mirchandaney et al. [MTS89], but these results show that overhead should be far less than 1/2 the execution time for good speedups on a two processor system.

8.6 Summary

The performance measurements on my Freeze Free process migration system confirm that the Freeze Free algorithm achieves targetted goals.

The Freeze Free algorithm completes the process migration latency period in 13.89ms for a 4kB page system, 20.82ms for a 8kB page system, and 36.89ms for a 16kB page system. These performance figures show that the system accomplishes process migration with minimal overhead. Virtual memory management operations dominate the cost in the process migration latency period.

The analysis of demand page overhead shows that the demand page cost now dominates the total overhead of process migration, with a cost of 208.8ms versus 13.89ms for the latency overhead in the sample program. Communication costs dominate the total demand page cost.

The experiments measuring load sharing speedup versus increasing overhead provide concrete evidence that minimizing overhead is absolutely essential to effective speedup.
Scheduling overhead adds to process migration latency and demand page overheads. Process migration requires priority based scheduling with priority based process preemption to support demand paging in a load sharing environment. Otherwise demand page delays become excessive.
Chapter 9

Algorithm Comparison

This chapter compares the Demand Page and File Server process migration algorithms against the Freeze Free algorithm. The chapter concentrates on the algorithms, and seeks to factor out the implementation differences. The comparison uses the same step performance times from chapter 8 for all three algorithms. The Total Copy and Pre-Copy algorithms have severe performance limitations compared to the Demand Page and File Server algorithms, and so the thesis drops Total Copy and Pre-Copy algorithms from further coverage. The chapter also describes how DVM could be used to support the Freeze Free algorithm.

9.1 Normalized Performance

It is unfair to compare process migration systems running on different platforms solely on the basis of absolute time. Both processor and network speeds are important perfor-
mance factors. The Demand Page algorithm implementation in Accent, the File Server algorithm implementation in Sprite, and the Freeze Free algorithm implementation in Choices all use a 10M bit per second Ethernet LAN. So raw communication speed is held constant. However processor speeds vary significantly. Therefore this chapter analyzes what the performance would be if the Demand Page and File Server algorithms performed their own steps as described in chapter 5 at the same speed as the actual Freeze Free algorithm implementation as reported in chapter 8. The original Demand Page algorithm implementation in Accent does not mention any file cache flush operation [Zay87a]. So the comparison assumes that the Demand Page algorithm memory maps files. Otherwise the Demand Page algorithm would flush the dirty cache blocks like the File Server algorithm.

The process migration step times come from several experiment measurements. The Freeze Free process migration system does not engage in request and response message pairs. Adding a little overhead to the round trip time for a null message (1.8ms) produces an estimate of 2ms for the Confirm With New Host step. The Control/Exec State step lasts from when the MigrateOut daemon begins until the MigrateIn daemon completes the insertion of the process control and execution state. The system overlaps work on the old host and the new host, and the transmission of data over the LAN. The comparison uses the time the new host completes work for the code, the heap, and the stack on the new host for the Freeze Free algorithm. The other algorithms demand page the code, the heap, and the stack page. The flush operation processes the first message through the message protocol stack and then time is dominated by the actual transmission. The
comparison assumes that up to four file descriptors fit in one page. Table 9.1 shows the costs of the steps involved in the Demand Page, File Server, and Freeze Free algorithms. Thus table 9.1 effectively provides normalized process migration latency costs formulas for the three algorithms.

<table>
<thead>
<tr>
<th>Step</th>
<th>Demand Page (ms)</th>
<th>File Server (ms)</th>
<th>Freeze Free (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confirm with new host</td>
<td>2.0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>Control/Exec State</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
</tr>
<tr>
<td>Communication Link(s)</td>
<td>1.8+3.5*L</td>
<td>1.8+3.5*L</td>
<td></td>
</tr>
<tr>
<td>File Control</td>
<td>1.8+9*F</td>
<td>1.8+9*F</td>
<td></td>
</tr>
<tr>
<td>Current Code Page</td>
<td>7.2</td>
<td>7.2</td>
<td>2.6</td>
</tr>
<tr>
<td>Current Heap Page</td>
<td>7.2</td>
<td>7.2</td>
<td>3.3</td>
</tr>
<tr>
<td>Current Stack Page</td>
<td>7.2</td>
<td>7.2</td>
<td>1.6</td>
</tr>
<tr>
<td>Resume Process Command</td>
<td>.9</td>
<td>.9</td>
<td></td>
</tr>
<tr>
<td>Dirty Stack Pages</td>
<td>1.8+3.5*S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dirty Heap Pages</td>
<td>1.8+3.5*H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dirty File Cache Blocks</td>
<td>1.8+3.5*B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L = Number communications links  
F = Number files  
S = Number additional stack pages  
H = Number dirty heap pages  
B = Number dirty file cache blocks

**Table 9.1**: Normalized Comparison of Process Migration Latency

Next is a comparison of the process migration latency times of a sample program using the Demand Page, File Server, and Freeze Free algorithms. The sample program has 25 dirty heap pages, 10 dirty file cache blocks, 3 open files, one communication link, and a stack of 528 bytes. The process migration latency time is 40.7ms for the Demand Page algorithm, 165ms for the File Server algorithm, and 13.9ms for the Freeze Free algorithm. The message freeze times are 21.1ms for the Demand Page algorithm, and 143.4ms for
the File Server algorithm. The Freeze Free algorithm does not have a message freeze period. Table 9.2 summarizes these results.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Total Time (ms)</th>
<th>Msg Freeze Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand Page</td>
<td>40.7</td>
<td>21.1</td>
</tr>
<tr>
<td>File Server</td>
<td>165.0</td>
<td>143.4</td>
</tr>
<tr>
<td>Freeze Free</td>
<td>13.9</td>
<td>0</td>
</tr>
</tbody>
</table>

Example Process
Code 100kB
Dirty Heap 25 x 4kB
Dirty File Cache Blocks 10 x 4kB
Open Files 3
Communication Link 1
Stack 528 bytes

Table 9.2: Comparative Process Migration Latency and Message Freeze Times

The Freeze Free algorithm blasts to the new host the communication link information and file descriptor information. Unless the migrating process accesses the communication link or files almost immediately, the three algorithms have virtually the same execution times after the process migration latency period.

9.2 Demand Page Algorithm

The Demand Page algorithm process migration latency time is 2.9 times longer than that of the Freeze Free algorithm. The message freeze time is short, but long enough to encounter many messages. The principle disadvantage of the Demand Page algorithm is its long term residual dependency on the old host, which eliminates its use for fault tolerance.
9.3 File Server Algorithm

The File Server algorithm eliminates the residual dependency at the cost of longer message freeze and latency times. The worst case message freeze and latency times are arbitrarily long, and can stretch into seconds. Communication failures can occur during process migrations, because of the long message freeze times. Load sharing suffers from the potentially long flush times prior to process resumption on the new host. However, when only limited numbers of dirty heap and dirty file cache pages exist, the File Server algorithm provides an acceptable tradeoff between fault tolerance and performance. The Freeze Free algorithm effectively eliminates message freeze times, has a very short process migration latency period, and provides fault tolerance by quickly eliminating residual dependencies on the old host.

9.4 Distributed Virtual Memory

Distributed Virtual Memory (DVM) offers an enticing means of transferring address space. When a process migration system uses page based DVM for address space transfers, the DVM approach has precisely the same performance advantages and disadvantages as the Demand Page algorithm. The DVM approach simply saves the process migration implementor the need to write code to transfer memory. Page based DVM combined with an option to transfer selected pages at times other than page faults, would enable DVM to provide the page transfer strategy used for the Freeze Free algorithm.
The use of page based DVM to transfer kernel data structures would suffer from a granularity mismatch that would cause the transfer of many unneeded bytes. An object based DVM [Tan95, chap. 6] could support kernel data structure transfers, but would suffer from extra messages. Object based DVM combined with an option to ship selected objects together in a single transfer would eliminate the extra overhead. An object based DVM approach is only feasible if the relevant objects are both modular and self contained.

Sprite [NWO88] demonstrated the value of sharing physical memory pages between the virtual memory system and the file cache. Several systems require a page transfer mechanism. This leads to a recommendation that the process migration system, the distributed file system, and the DVM system share a page transfer mechanism.

9.5 Summary

The Freeze Free algorithm outperforms both the Demand Page and File Server algorithms even after normalizing step speeds for both process migration latency times and message freeze times. The Freeze Free algorithm achieves performance without sacrificing processor fault tolerance.
Chapter 10

Conclusion

This thesis research seeks to demonstrate ways to reduce the overhead of process migration in existing systems by an order of magnitude. The new Freeze Free algorithm for process migration dramatically reduces overhead through the use of six techniques described in chapter 6. This chapter summarizes the major accomplishments in process migration and identifies promising future avenues for further improvements.

10.1 Process Migration Latency Time

A prevalent myth is that process migration latency is so large that process migration is not generally cost effective. Mullender states “Process migration remains an expensive operation, however — so expensive, that I do not know of distributed systems that use it for load-balancing purposes” [Mul93, p. 401]. This thesis research shatters this myth by achieving an order of magnitude improvement over all prior process migration systems.
The Freeze Free process migration system achieves process migration latency times of 13.89ms for a 4kB page system, 20.82ms for 8kB page system, and 36.89ms for a 16kB page system. By way of comparison, Sprite requires 330ms.

The Freeze Free algorithm reduces the number of messages sent during the process migration latency period to four messages containing the combined process control and execution state, the current code page, the current heap page, and the current stack page. A process migration system that supports the migration of a wide class of application processes can not further reduce this information. The Freeze Free algorithm allows for variations in both the page size and the process control and execution state information.

Several other changes contribute significantly. The Freeze Free algorithm eliminates all request and response message pair dependencies from the process migration latency period. The Freeze Free algorithm uses a pool of partially initialized objects to shift expensive object creation operations out of the process migration latency period. The Freeze Free algorithm design systematically reorganizes system objects so that they are self-contained, which means that the design places all state information about an object within that object. This reduces excision and insertion costs.

### 10.2 Message Freeze Time

Existing process migration systems freeze message processing during part of process migration. The root cause of the extended message freeze time is the entanglement of the process state and the message queue state. The Freeze Free algorithm design
separates the process state and message queue state. This permits the message queue to continue receiving messages while the process migrates. The Freeze Free algorithm never freezes message processing. The message queue accepts messages when it is on either the old or new host. When the message queue is in transit, messages are temporarily held in suspension on the new host. The suspension lasts only as long as required to transfer the message queue, which is short enough that it can be treated like a page fault to disk.

10.3 Modularity

Many existing operating systems widely scatter process related information. That approach makes extracting and inserting process related information expensive. That approach also entangles the state of the process and the state of the module holding the process state information. The net result is that an action on either the process or the module holding information about the process must account for effects of the actions on both entities. This manifests itself at process migration time in the form of extended freeze times for the process and its supporting modules. One example is the message processing freeze time. The new design receives messages in parallel with process migration. The file descriptor information provides another example. The new design enables the old host file cache flush to proceed in parallel with the execution of the migrating process on the new host. This leads to the general recommendation that the system decouple the process state from its supporting modules.
10.4 Load Sharing vs Overhead

The effectiveness of load sharing in a distributed system is inversely proportional to the overhead incurred in implementing load sharing. The process migration latency operations, the multiple demand pages across the network, and the scheduling activities add overhead. The experiments measuring the effects of overhead on load sharing, as shown in figure 8.6, graphically portray the negative impact of overhead on performance. Small overheads are needed for high speedups. For example, an 11% overhead results in a 1.8 speedup on a two processor system, while 100% overhead results in no speedup.

The large reduction in the cost of the process migration latency period reduces its role in load sharing overhead. For the sample test process the process migration latency operations cost 13.89ms, while the demand pages across the network cost 208.8ms, which is an order of magnitude larger. Future process migration improvement efforts will have to place a greater emphasis on the cross network demand page operations.

The ProcessManager schedules processes according to priority, but does not preempt processes for newly ready higher priority processes. Thus the high priority process serving page requests waits for the time slice expiration of a low priority, compute bound application process. This adds delays to each page request, and combines to produce an enormous overhead. Priority based process preemption is necessary for effective load sharing.
10.5 Fault Tolerance

The Freeze Free algorithm supports fault tolerance by quickly moving all process related information off the old host. The dependency time period is directly and linearly dependent on the number of dirty pages and file cache blocks present on the old host. The dependency period is generally much longer than the process migration latency period.

10.6 Opportunities for Further Improvement

While the Freeze Free algorithm achieves significant performance improvements, the performance analysis uncovered several promising areas for future improvement.

10.6.1 Higher Bandwidth Network

The replacement of the 10M bit per second Ethernet with a 100M bit per second ATM network would dramatically reduce transmission times. On the sample program, the higher bandwidth would reduce 91.1ms out of 223ms for a 40.9\% reduction.

10.6.2 Page Transfer Protocol

The development of a protocol for transferring pages coupled with smarter network interface devices could eliminate data copying. On the sample program, the elimination of data copying would eliminate 42.5ms out of 223ms for a 19\% reduction.
10.6.3 Communication Protocol Handling

On the sample program, the communication protocol handling consumes 59.2ms out of 223ms, which is 26.5% of the process migration overhead. Several options exist for further reducing communication protocol handling costs. A new network supporting larger packet sizes could eliminate the fragmentation costs. The x-Kernel package has a very deep level of procedure nesting (about 30) for my ProcessMsg protocol. Replacing procedure calls with in line code expansions is one option. Another option is redesigning the protocol stack to reduce layers. The bottom line is that even small improvements in this area will affect process migration because of its significant contribution to overhead.

10.6.4 Virtual Memory

On the sample program, the virtual memory and process initialization operations consume 18.4ms out of 223ms, which is 8.3% of the process migration overhead. These two operations consume 41.7% of the process migration latency time. So virtual memory operations becomes more important with smaller processes. Adding address translations alone consumes 14ms, so improving address translations is a clear target for future work.

10.7 Opportunities for Added Capabilities

The process migration system would benefit from several additional features. The Freeze Free algorithm can readily be extended to support the migration of processes with multiple threads. The integration of DVM and process migration would enable the migration
of processes with shared memory regions. The current system combines a system library into the executable load module. The system could be changed to use a memory resident system library for applications. The process migration system would not have to migrate the library, because the library would be present on all machines.

10.8 Related Areas

The high overheads of earlier process migration systems severely handicapped effective load sharing. Now that the Freeze Free algorithm dramatically reduces the overhead of process migration, it is time to reexamine distributed system process scheduling. Someone should systematically compare the effectiveness of the top dozen scheduling policies for distributed systems, much as Leutenegger systematically compared scheduling for multiprocessors [Leu90]. The scheduling overhead should also be measured.

Process migration should not be added to a system by itself. Process migration should be added in conjunction with distributed system scheduling, automatic task placement, and monitoring facilities. Parallel make and similar utilities are essential for widespread acceptance and use of process migration.

10.9 Summary

In conclusion, the new Freeze Free process migration algorithm reduces process migration latency times by an order of magnitude to 13.89ms for 4kB page systems, 20.82ms for 8kB page systems, and 36.89ms for 16kB page systems. The thesis identifies where additional
changes could produce further major savings. The Freeze Free algorithm effectively eliminates message freeze times. The Freeze Free algorithm supports fault tolerance by rapidly eliminating dependencies on the old host. And thesis experiments graphically show the reductions in load sharing system speedup caused by increasing overhead.

The high cost of previous process migration systems was the principal barrier to the widespread use of process migration for load sharing and fault tolerance. The Freeze Free algorithm improvements in process migration open the way to the full utilization of the computational power in distributed systems.
Bibliography


[GGI+91] Gerrity, Goscinski, Indulska, Toomey, and Zhu. Can we study design issues of distributed operating systems in a generalized way? In *Symposium on*


Vita

Ellard T. Roush was born 5 December 1953 in San Mateo, California. He earned a B.S. in Mathematics from the University of Washington, and graduated Cum Laude in 1976. He earned a M.S. in Computer Science: Computer Engineering from Stanford University in 1977. He earned a M.B.A. from Campbell University in 1980. Mr. Roush is nearing completion of the Ph.D. in Computer Science at the University of Illinois at Urbana-Champaign. He is specializing in the area of distributed operating systems, as evidenced by this thesis.

As a Ph.D. graduate student at the University of Illinois from 1992 to 1995, Mr. Roush created a new algorithm that overcomes major obstacles to effective Process Migration (moving running programs between computers in a network environment for improved performance and reliability). He implemented a new algorithm that performs over 20 times faster than world’s previous best and effectively eliminates message freeze time. He replaced the operating system process subsystem, resulting in significant kernel performance improvement, as in process locking which is 125.8 times faster on Virtual Choices and 5.35 times faster on SS1 Choices. A TCP communications application improved data rates by 8-22% for transmission and 15-24% for reception. Choices is a multi-threaded, single/multi-processor operating system implemented in “C++”.

As the Project Technical Advisor at Sterling Software from 1986 to 1990, Mr. Roush performed technical consulting for European site personnel and central software team on the MAXI project. He designed and developed software, maintained large fielded systems (500,000+ lines of code), and installed systems. MAXI is a “C” and assembly language software package that at each of many world wide sites ties together multiple minicomputers (VAX and PDP-11) utilizing ICC. The system connects to several communication networks (SCINET, AUTODIN, and OPSCOMM) to transmit, receive, and process information that is often time sensitive.

He initiated efforts and developed a plan to use standard window (GUI) technologies for work station access to MAXI through a Client/Server architecture. A government contract resulted, and an X-Windows (Motif) based package was fielded for Sun and DEC work station access to MAXI.

He regularly field tested new software releases and developed the many changes needed for the operational sites. Real world 24 hour missions required near constant availability. When on-site software staff encountered difficulty, he provided cures. Solutions spanned operating system kernel code, such as inter-task communication and device handlers; local and remote communications; real-time applications; user interfaces; and microprocessor
devices. One trouble shooting example; cured erratic terminal reliability by identifying and overcoming in microprocessor code a slight physical device timing mismatch.

For another project, he delivered reliable software involving TCP/IP communications across an Ethernet LAN connecting Macro-11 and FORTRAN software under IAS on PDP-11’s, “C” software under MVS on an IBM 4381, and “C” software under UNIX on work stations.

As a Senior System Consultant at Inco, Incorporated from 1981 to 1986, Mr. Roush managed entire MAXI system at HQ USAFE/OSC, Ramstein, Germany. The initially delivered system suffered down times exceeding 30%. He quickly provided numerous software fixes, which significantly reduced outages to under 1% for a period exceeding a year of 24 hour operations. He delivered new software, such as the Profile Maintenance Subsystem; and performed system tuning, maintenance, and training.

On a previous project, he developed a security accreditation plan for a system involving numerous computer systems and communication links.

He developed several “C” Unix software packages, including one utilizing Ingres, for the 66th MI Group, Augsburg, Germany.

As a Captain in the U.S. Army, with service from 1977 to 1981, he managed the section responsible for the operating systems on DEC PDP-11 and IBM 360 computers. He led the group delivering the FORTRAN based Army Standard Intelligence Plotting System version 1.7 graphics package for PDP-11, IBM 360/370, and Honeywell 6000 series computers. He provided technical guidance on Army computer software projects.

He implemented a Macro-11 device handler supporting terminals, printers, and SU1652’s connected to a BR1569 32-line interface on PDP-11/70’s. Forms mode and graphics were supported.