

**The Experimental Migration of a Distributed Application to a
Multithreaded Environment.**

by

Thuan Q. Pham

B.S., Massachusetts Institute of Technology (1990)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

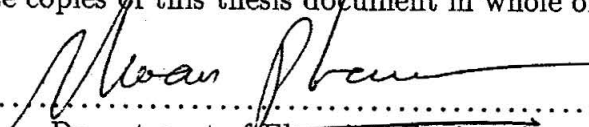
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1991

© Thuan Q. Pham, 1991

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

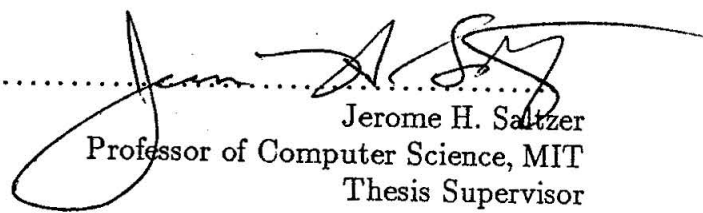
Signature of Author.....



Department of Electrical Engineering and Computer Science

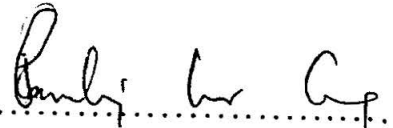
June 3, 1991

Certified by.....



Jerome H. Saltzer
Professor of Computer Science, MIT
Thesis Supervisor

Certified by.....



Pankaj Garg, Ph.D.
Hewlett-Packard Laboratories
Thesis Supervisor

Accepted by.....

Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Acknowledgements

The author would like to thank Dr. Pankaj K. Garg for the ideas that formulated this thesis work, and for the invaluable support, help, and guidance every step of the way.

Many thanks to professor Jerome H. Saltzer for the close supervision of the work at every stage; for the tough questions and the insightful comments that have kept everything on track; and for the perusal of this thesis' drafts.

My colleagues Debbie Caswell and Scott Marovich provided enormous help and expertise with the UNIX and MACH operating systems. Special thanks to Scott for all the debugging help and to Debbie for the careful readings of the thesis' early drafts.

Finally, I would like to thank my parents and my special friend Thảo for their love and support. A very special tribute to my dear mother, Mrs Mã Thị Kim-Liên, who made the ultimate sacrifice in bringing us children to this great country from the oppressive communist state of Vietnam, giving us a second chance to live and think in freedom. I love her with all my heart and affectionately dedicate this thesis to her. Thank you, mother, for giving me the past, the present, and the future.

Contents

1	Introduction	10
1.1	Thesis of experimentation	10
1.2	The experiment	12
1.3	Facility and equipment	15
2	System Architecture	16
2.1	Object editor	19
2.2	Workshop process	21
2.3	Interaction	22
3	System Re-architecture	23
3.1	From processes to single-thread tasks	23
3.2	Merging single-thread tasks to one multi-thread task	25
3.2.1	Running with threads	26
3.2.2	I/O contention	26
3.2.3	Problems with UNEXEC	27
3.3	Sharing memory	29
3.3.1	Object modeling	30
3.3.2	Garbage collection considerations	32
4	Performance Evaluation	34
4.1	Timing measurements	35
4.2	Test cases	37
4.3	Experimental results and analysis	38

4.3.1	Light-weight threads and shared memory	39
4.3.2	Bytes vs messages	41
5	Conclusion	44
5.1	Summary	41
5.2	Discussion and future considerations	45
5.2.1	Shared-memory architecture	45
5.2.2	System threads vs user threads	46
5.2.3	Multiprocessor implications	47

List of Figures

1-1	<i>Matisse</i> Architecture: A central shared information base is shared by all team members. Each person has an individual (active) information base which talks to various interactive tools.	14
2-1	<i>Matisse</i> : experimental subset	18
2-2	An example object	19
2-3	A sample display for a function object	20
2-4	Display attributes structure of a segment	20
2-5	Workshop object structure. a) "Object x" is a collection of object-slot-value triples with the same object ID. b) We can think of an object in the Workshop as having a set of slot-value pairs.	21
3-1	Re-architecture process: from disjoint single-thread tasks to multi-threads task with memory sharing capability. a) single-thread tasks, b) multi-thread task, c) multi-thread task with memory sharing.	24
3-2	resource contention: the communication channel <i>stdin</i> was "dup"ed and the copy was given to the Workshop process to avoid contention.	27
3-3	synchronization on thread exit. To ensure that the Editor thread exits cleanly, the Workshop thread waits for the join operation to complete before proceeding to call <i>unexec</i>	29
3-4	Data sharing: The new read primitive transparently reads the data directly from the Workshop heap and converting it to the format usable by the Editor, thus eliminating the need for IPC.	30

3-5	Read/write synchronization: If the write routine connects or disconnects the link to new/old object data in a last, atomic step, there won't be any conflict with the read routine looking at the same piece of data at the same time.	31
3-6	Memory management alternative: the object management mechanism can be implemented by an independent thread. Synchronization of object access between the Workshop and Editor threads can be handled by the Object Manager.	32
4-1	Timing measurement. This operation scenario contains 3 <i>Editor</i> transactions and 1 <i>Workshop</i> transaction. The total execution time is the sum of three (t_4-t_1) and one (t_3-t_2) , in both user time and system time. Note that the idle time is not charged to the execution time of either thread.	36

List of Tables

4.1	Performance Improvement, test scenario 1	39
4.2	Performance Improvement, test scenario 2	39
4.3	Performance Profile, test scenario 1	41
4.4	Performance Profile, test scenario 2	42

Chapter 1

Introduction

1.1 Thesis of experimentation

Light-weight computation threads in a multi-threaded operating system promise to provide low-overhead computation objects compared to their counterparts in conventional process-oriented operating systems. Traditional distributed applications using heavy-weight and disjoint computation processes could be merged as concurrent threads in a multi-threaded platform to take advantage of faster context switches and interprocess communication via shared memory. The hope is that there will be substantial performance improvement over existing implementations with single thread processes. An investigation of this hope is made by porting an existing distributed system from UNIX to MACII and merging some (single-thread) processes into one multi-thread task. This study addresses the benefits, the difficulties, and the trade-offs of such a mapping. We suggest some feasible architectures for migrating current distributed systems to multithreaded environments.

Before discussing this issue further, it is helpful to recall a few definitions of *processes*, *tasks*, and *threads*.

- A *process* is “a program in execution” [Sal66]. As the term is usually used, it includes a collection of system resources (i.e. memory image, register sets, open file descriptors, program counter, etc.) with a single thread of execution.

- A *task* is a collection of resources equivalent to a process without an execution thread, namely an execution environment in which threads may run [Ras86]. A *thread*, on the other hand, is the basic unit of computation, or simply a program counter with a register set and a context within a process. Each thread operates within the context of exactly one task, and many threads can co-exist within the same parent task, sharing all task resources [JRG+87a].

The usual process abstraction has too many things anchored to it to meet the needs of aggressively parallel applications; as a result, process creation and context switching result in high overhead on the part of the operating system, often using far more resources than one would like [ABB+86]. As a result, various efforts have been made by programmers to circumvent this problem, such as using coroutine packages to simulate and manage multiple contexts within a single process [ea85, JRG+87b]. However, these coroutine packages cannot take advantage of the operating system's scheduling services since the kernel has no knowledge of such coroutines or sub-processes. Thus, the question of alleviating expensive context switching is not completely addressed by user-created coroutines and sub-processes. Furthermore, since the processes do not share resources, distributed applications with large amount of data sharing must communicate via interprocess communication (IPC) mechanisms, which are costly both in time and resources.

The problems described above can be addressed by the operating systems supporting multithreading, with light-weight threads and shared resources. Being light-weight, creating and maintaining threads require lower operating system overhead than heavier processes. A thread, when created, has access to all the process information in the task. During a context switch, if a thread to be run belongs to the same parent task as the thread currently occupying the processor, only a few registers need to be saved, leaving most of the task's resources in place. In addition, the threads within a task are managed automatically by the operating system kernel. Since the computation threads share all resources within a task, including the memory address space, "inter-thread" communication can be done cheaply and efficiently via direct data sharing.

For the reasons mentioned, a systematic reduction of heavy-weight processes to light-weight threads, whenever possible, provides a substantial improvement in performance. However, the threads facility restricts the architecture of distributed systems. Further cost is incurred by the effort spent in porting existing software systems to a new, multi-threaded platform, which has some difficulties of its own. Clearly, performance improvements come at the expense of the lost generality to the process model: loosely-coupled processes can be run simultaneously in different machines; tightly-coupled (shared memory) threads must reside in the same machine. This issue is investigated by a UNIX-to-MACH port of an existing distributed application, meshing together some processes to threads within the same enclosing task. This study identifies the problems and benefits of such a re-architecture for future distributed systems in single and multithreaded environments.

1.2 The experiment

Matisse is a knowledge-based team programming environment under development at IIP Laboratories. *Matisse* offers automated support for communication and coordination efforts in team programming [AGS89]. Its architecture is illustrated in Figure 1-1. The core of each unit of *Matisse* includes an inference engine, an object editor, and a graphical object browser, all residing in the user's workstation. With the current model of implementation, each component within a unit of *Matisse* is a separate process. These processes are executed concurrently and must share information by passing large amounts of data back and forth between them. For example, when the user is using the editor to modify his program objects, the editor, the object browser, and the inference engine must communicate frequently, transferring numerous requests and very large amounts of data via sockets to update and reflect the most recent and correct system configuration. In a time-sharing computing environment, managing concurrent processes is expensive because of the context switches needed to distribute CPU time to each of the processes. Consequently, by merging some of these UNIX processes into threads within the same MACH task, we can obtain sig-

nificant performance improvement with light-weight threads and with inter-process communication via direct memory sharing.

In this experiment, it is sufficient to port and merge only the *Workshop* process and the *Editor* process. These processes are prime candidates to receive the benefits of the merge because they always reside on the same machine and must share a large collection of data via numerous large IPC messages. The locality of these processes enable them to be merged without any loss of generality or usefulness, and their interprocess communication can benefit greatly from direct memory sharing. Performing the port and the merge, and using the resulting multi-threaded applications provide us some insights about the costs, the benefits, and the feasibility of such re-architecture.

This experiment is divided into three stages. In the first phase, *Matisse* is ported from UNIX to MACH with its architecture essentially unaltered. This initial port maps each UNIX process to a MACH task with only one execution thread. A series of measurements are taken to assess the performance of the MACH implementation of *Matisse* and to serve as a baseline for future measurements. In the second phase, the two single-thread tasks are merged into a 2-thread task, but with the two threads still communicating and sharing data via sockets. Another round of performance measurement is taken here, and the results, when compared against the baseline, provide us some information about the performance of the system relating to having multi-threaded task versus single-threaded tasks. In the third and final stage of the experiment, large IPC messages between threads are eliminated and replaced with a memory-sharing protocol. At the completion of this phase, performance measurements are taken and compared against the first two.

In addition to providing performance measurements, the experiment gives us some understanding of the difficulty of this porting process and some insights into the troublesome areas, including the conditions and requirements that make the operation possible and optimal (data representation, locality, garbage collection, etc). These rules of thumb might be a helpful guide in identifying a suitable architecture, or re-architecture, of processes and threads for distributed software systems in the future.

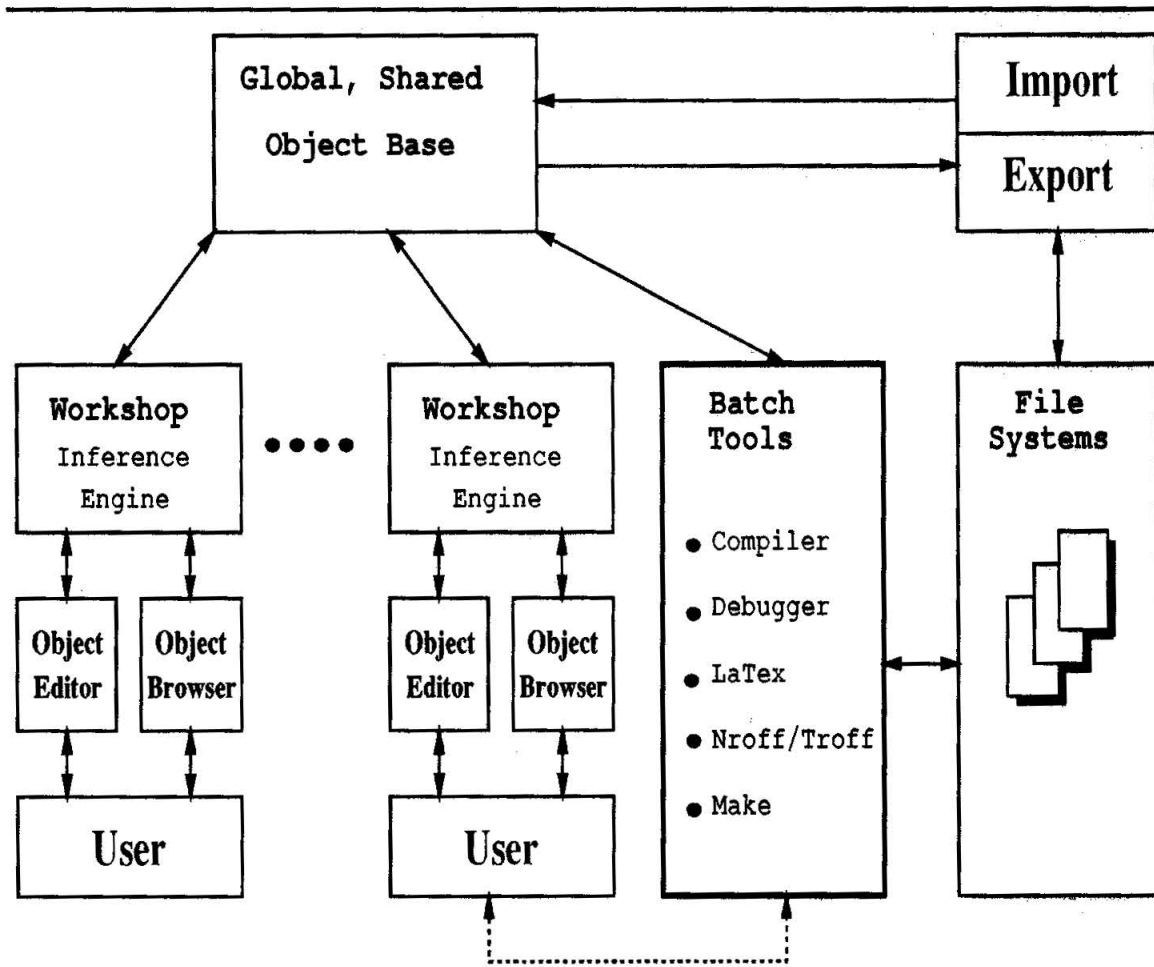


Figure 1-1: *Matisse* Architecture: A central shared information base is shared by all team members. Each person has an individual (active) information base which talks to various interactive tools.

Intuition and past research on related issues [ABB⁺86, FR86, JRG⁺87a, TR87] suggest that there is a substantial performance improvement to be gained from the successful reduction of heavy-weight processes to light-weight threads and inter-process communication via memory sharing techniques. However, such improvement does not come without costs and it is up to the designer to judge whether this approach is feasible for his or her specific applications.

Chapter 2 describes the architecture of *Matisse*. Chapter 3 discusses the issues and problems encountered during the port and the re-architecture. Chapter 4 presents the performance data at several stages of the experiment and the analysis of the costs and benefits learned from the experiment of multithreaded architecture. Chapter 5 offers the conclusion and thoughts about the experiments as well as listing the possible issues worth investigating in the future.

1.3 Facility and equipment

The experiment is carried out using the equipment and facilities of the Hewlett-Packard Laboratories. *Matisse* is currently operating on a single-processor workstation HP9000/370 running HP-UX 7.0. The MACH port operates on the HP9000/350 running HP's MACH 2.0 [CM89a]. MACH 2.5, although having a better scheduling algorithm than version 2.0 currently used, is not complete enough on the HP9000 workstation to be used as a platform for the experiment. All hardware and software needs are provided by the Hewlett-Packard Laboratories.

Chapter 2

System Architecture

As depicted in Figure 1-1, *Matisse* is a distributed application with 3 primary components: a persistent object base, an inference engine, and various interactive tools such as an object editor and an object browser.

The persistent *Object Base* is provided by an object-oriented database. The *Object Base* not only stores the programming objects generated by the users, but also records the evolution of objects by keeping the links between their successive versions. Furthermore, the object-oriented database provides the capabilities to support object inheritance, special object clustering, and object references [AGS89]. Since *Matisse* operates on top of an object platform, file systems are only an auxiliary part of the environment. In order to support the existing file-based software systems, and to utilize the existing file-based tools (e.g., compiler, debugger, text formatter, etc.), it is necessary to interface the object and file domains. A set of *import* [Pha90] and *export* [Gar91] tools is provided to serve this purpose.

The second major component of *Matisse* is the inference engine called the *Workshop*. The *Workshop* is a rule-based inference engine and serves also as a local cache of program objects in a user's workstation. While its main objective is to perform the pattern matchings on program objects that are needed to support the associative queries, the *Workshop* process also uses its rule base to intelligently manage the software development process – from automatic constraint checking to invoking external tools on an event-driven basis [AGS89]. Since the *Workshop* process uses declarative

rules rather than conventional, hard-coded, procedures, the user has the flexibility to customize the set of rules according to his needs, development models, and policies.

The third and most visible component of *Matisse* is the set of user interactive tools used to edit and display program objects. The two most notable tools are the *Object Editor* and the *Object Browser* (also known simply as *Editor* and *Browser*). The *Editor* is an *Emacs* based editing and browsing interface [GYA90a], capable of displaying programming objects in independent sections of the display buffer. This capability enables powerful ways of displaying objects, such as grouping together all the objects of a certain characteristic in the same text buffer for editing. Moreover, by clicking the mouse on the selected regions of the text buffer, the *Editor* can be used simply as a navigation tool through the network of locally cached objects. On the other hand, a more graphically-oriented *Browser* provides a powerful way to navigate through the local object store using X widgets. With a few simple mouse clicks, one can select to view, create, or delete the program objects, their links, and their attributes.

For the purpose of this experiment, we concentrate on a subset of *Matisse* involving the *Object Base*, the *Workshop*, and the *Editor* (Figure 2-1). More specifically, we ported the *Workshop* process and the *Editor* process to MACH tasks, then merged them into a multithreaded task and allowed them to share the same pool of object data. Although we won't be modifying the *Object Base* implementation, its presence in the architecture is necessary since it is an integral part of *Matisse*. Both the old processes and the new task ultimately connect to the *Object Base* in order to function.

A decision was made to merge the *Workshop* and the *Editor* because they are prime candidates to benefit from such a re-architecture. The two processes share large amounts of data by communicating constantly with one another via sockets. By merging these two processes into a multithreaded task and enabling them to share data directly, we have an improvement in performance resulting from light context switches and from not having to spend heavy resources on interprocess communication. The following sections describe in detail the working internals of the *Editor* and the *Workshop* processes.

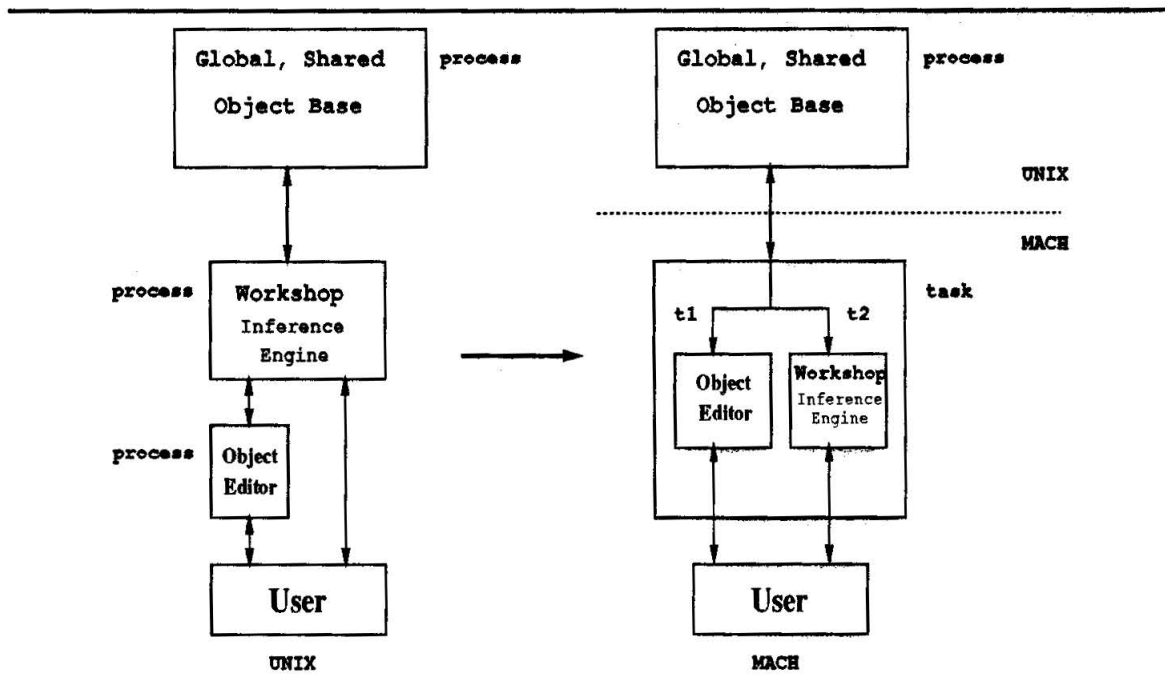


Figure 2-1: *Matisse*: experimental subset

```
Object1
Has-class: c-function
Has Name: printf
Author: Tom Harry
Job: Demo
Element-of: demo-objects.c
Has-source:
printf(arg)
  char *arg ;
{
  while (*arg++)
    putc(*arg) ;
}
```

Figure 2-2: An example object

2.1 Object editor

The *Editor* is essentially an *Emacs-based Software Object Editor* [GYA90b] that has the capability to display program objects with different sizes and having different facets in the same text buffers. This special capability is made possible by superimposing a structure on top of the linear structure of *Emacs* [GYA90b], effectively dividing the text stream into sections. Each section can then have its own display attributes such as color, visibility, name, etc. An “object”, to the editor, is simply a collection of segments in the display buffer. Figure 2-2 and Figure 2-3 show some of the ways an object can be displayed according to the user’s desire.

Like *Emacs*, the user can interact with the *Editor* via the keyboard and the mouse. With the extension of sections, the *Editor* has more sophisticated editing features such as designating a certain section to be “hidden”, or to be “read-only”, in order to prevent unauthorized and unwanted modifications, for example. Furthermore, sections allows us the convenient way of issuing commands via mouse clicks in the text buffer. By issuing mouse clicks in certain customizable sections of the text buffer, the user can either navigate through the local object cache, or invoke actions to be

```
/* <<printf (Demo) >> */
printf(arg)
char *arg ;
{
while (*arg++)
    putc(*arg) ;
}
```

Figure 2-3: A sample display for a function object

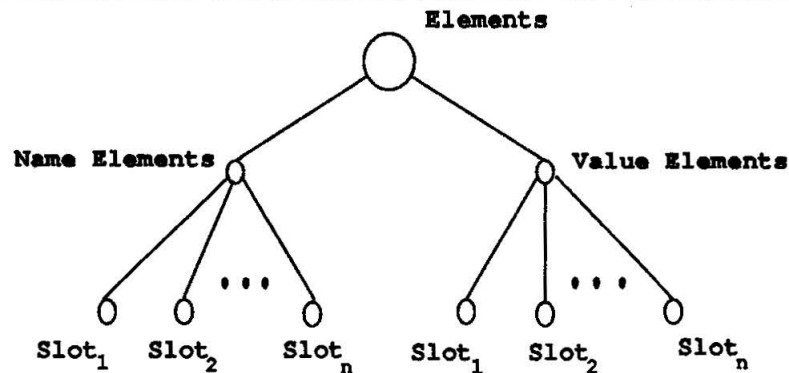


Figure 2-4: Display attributes structure of a segment

performed.

To support such a display capability, the *Editor* must maintain some information about the nature of the objects being displayed. This information is stored in a tree of attribute nodes as illustrated in figure 2-4. Each of the slot nodes contain information about the object (e.g. object class, inheritance, text string, display attributes, etc.). Much of the information stored here is also present in the *Workshop*, and requires frequent updates and synchronizations of the two object bases as the program objects in them are created, used, and modified. Here, the potential for a performance gain from the multi-thread merge is great because we would not have to spend resources in managing redundant information, and especially not having to spend the effort in synchronizing the two sets of data.

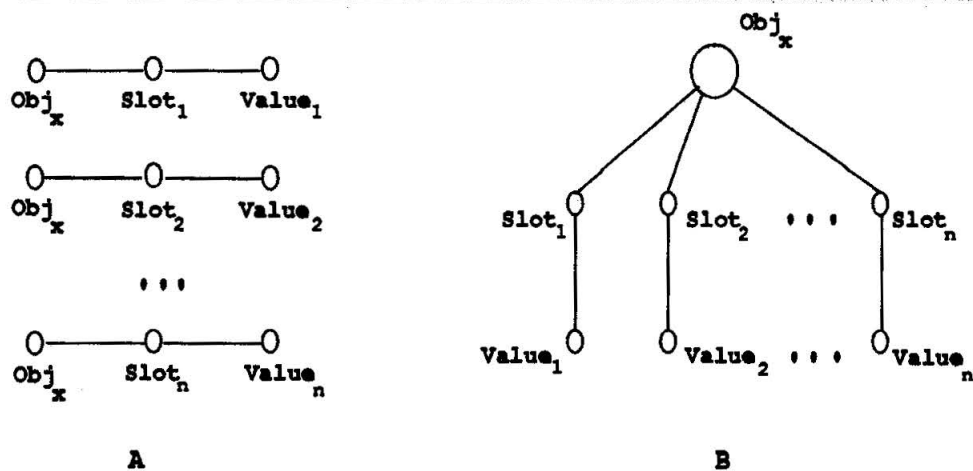


Figure 2-5: Workshop object structure. a) “Object x” is a collection of object-slot-value triples with the same object ID. b) We can think of an object in the Workshop as having a set of slot-value pairs.

2.2 Workshop process

The other major component of the system is the *Workshop*, which is a combination of a rule-based inference engine and a heap of storage for locally cached objects. The *Workshop* not only provides a shared object base of information, but also supports rule-based reasonings for managing the software development process such as constraint checking, or processing a user’s requests.

The *Workshop*, also known as *Eclipse: a C-based Inference Engine Embedded in a Lisp Interpreter* [BS90], is a product of fusing together XLISP [Bet89] and CLIPS [Art88], with some added features. XLISP is a small, fast, and powerful LISP interpreter written in C, and CLIPS is a C-based forward chaining inference engine. Together, they form a flexible, customizable, C-based inferencing system with object storage capabilities. The *Workshop* can thus administer the software development activities by using the set of user-defined rules to intelligently manage data in the object base.

Data in the em Workshop is stored simultaneously in two different formats: as facts to satisfy the rule base’s pattern matching requirements, and as a pool of *object-slot-value* triples to facilitate efficient access to objects when direct lookup is needed. Each

fact corresponds to an *object-slot-value* triple, and a program object is a collection of *object-slot-value* triples that have the same object ID (i.e. the first element). Figure 2-5 illustrates the object structure within the *Workshop* process. Although slightly different in format, these objects overlap a great deal with those in the *Editor*. A merge of the two object bases would reduce the cost of having to maintain redundant information. The redundancy here does not provide us any advantage in a sense of data replication.

2.3 Interaction

Together, the *Editor* and *Workshop* cooperate to support users' transactions, with the *Editor* handles most of the I/O while the *Workshop* performs most of the computations. In the Current implementation, the *Editor* accepts a user's input from the keyboard and mouse, then passes data via sockets to the *Workshop* for processing. When the *Workshop* finishes the requested job, it sends the data back, also via sockets, to the waiting *Editor*. At this time, the *Editor* uses the data to update its display screen and to synchronize its object pool with that of the *Workshop* by replacing old entries with the new data.

By merging the two processes as threads and combining their data storages in shared memory, we have a performance gain not only from interprocess communication via memory sharing, but also from not having to maintain unnecessary copies of data.

Chapter 3

System Re-architecture

As mentioned in chapter 1, the experiment is divided into three stages. We first performed a “straight port” of the Editor and the Workshop, making each process a singlethreaded task in the new environment. The second stage involved merging these two singlethreaded tasks into one task with two threads while leaving their interprocess communication mechanisms undisturbed. Finally, in the third stage, we replaced most of the IPC messages between the two threads with some primitive routines that allow direct access of data in memory between threads, and a new communication protocol using these primitives. Figure 3-1 graphically describes this experiment, showing the shared memory and illustrating also the process, task, and thread boundaries. The division of the experiment into these three stages is natural, since each stage has its own set of problems that, for the most part, is distinct and unrelated to the others. The following sections describe each phase of the project, discuss the difficulties that were encountered, and render our solutions.

3.1 From processes to single-thread tasks

The first step of porting *Matisse* from a single-threaded to a multi-threaded environment is necessary to establish a stable foundation on which the experiments with light-weight threads and shared memory can be performed. For this portion of the experiment, our only concern was to reconstruct the system on the new multithreaded

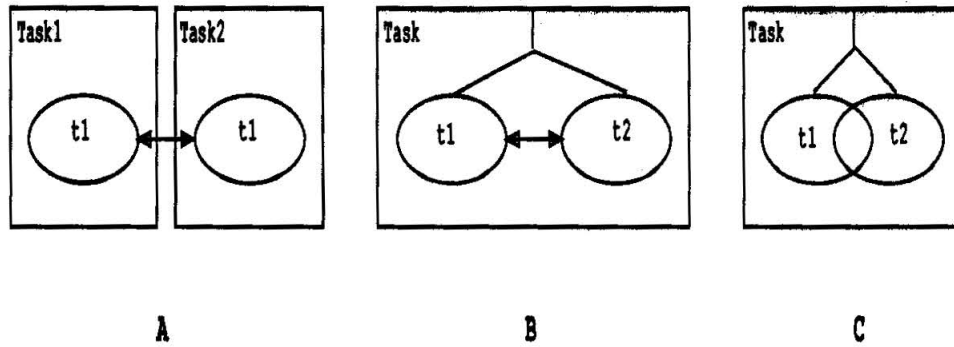


Figure 3-1: Re-architecture process: from disjoint single-thread tasks to multi-threads task with memory sharing capability. a) single-thread tasks. b) multi-thread task. c) multi-thread task with memory sharing.

platform, making each former single-thread process a new singlethread task. This porting effort exposed the dissimilarities between the two operating systems, both in features and implementation. Furthermore, this version of the system serves as a baseline against which subsequent versions are measured and compared.

Although this is the simplest and most straight-forward step of the experiment, it is, however, not necessarily easy, depending on the portability of the existing system. This step can be as simple as a recompilation, or as arduous as a major rewrite. For example, our Workshop port went particularly smoothly, requiring little beyond setting up the directories, the Makefile, and the recompilation. On the other hand, our Editor, a modified Emacs, was particularly hard to port since its complex and peculiar code exploited many system-specific features, and exposed numerous system differences.

While porting, we encountered the usual problem of having different names for [equivalent] systems calls and different directory hierarchy between the file systems. This problem can be rectified by meticulously checking, identifying, and tracking down the right functions and files in the new environment.

A slightly harder problem we encountered was the difference of implementation and system behaviors between the operating systems. One notable example was the

incompatible side-effects of the functions *regcmp/ regex*: one set of function stored the data to be processed in an internal static structure, and thus could not be called recursively, while the other took the argument from the program stack, contained no static internal state, and could be used recursively by the program. When facing this type of discrepancy, where the source of the problem is embedded deep inside the system libraries, it was not possible for us to re-implement these libraries to suit only our needs since there are other existing programs that may already depend on these system behaviors. Our solution was to define the macros, functions, and data structures necessary to use over the existing resources.

The problems described above are just a few of the many that could arise in every port effort. There is no hard and fast rule for telling in what form and flavor the problems will come under, but their cause is most often some system dependencies or assumptions by the application program that happen to expose the dissimilarities between the two operating systems. These problems are straight-forward and can be resolved with some debugging. Our port of *Matisse* was determined “complete” when we have achieved identical system behavior, in both look and feel, of six typical usage scenarios.

3.2 Merging single-thread tasks to one multi-thread task

Once the initial port was completed, we proceeded to merge the two single-thread tasks into one 2-thread task. In order to measure the effects of light-weight threads on system performance, and to effectively deal with the issues arising from the merge, we left the interprocess communication mechanism unchanged. The performance measurements taken at this stage, when compared with the base line, gave us some insights about the advantages to be gained from converting an existing single-threaded application to a multi-threaded one. Furthermore, the merge also presented us some interesting problems regarding the use of common system resources and synchronization between threads, particularly with the termination of threads and restarting

them before and after *unexec*¹. The following sections describe in turn the merge process, the problems encountered, and their solutions.

3.2.1 Running with threads

In order to merge the singlethreaded tasks, we first resolved duplicate file names and global variable names of the previously disjoint programs, including the *main* functions which had to be renamed. A new *main* function was implemented to set up global, shared resources such as thread IDs, mutex and condition variables. This *main* function forked the *Editor* thread and then ran the last *Workshop* code as part of the main thread. In addition, since the thread interface allowed only one argument to be passed to a thread at creation time, the main function must parse the command-line arguments, package them in to appropriate structures, and hand them to the appropriate threads when calling the thread creation routine.

3.2.2 I/O contention

After bringing up the multi-threaded system, we noticed that it behaved unpredictably on a random set of user inputs. Upon investigating the matter, we uncovered a problem of resource contention between the two concurrent threads. In our case, the two threads both used the file descriptor associated with the standard input stream and listened on this channel for user's input². Facing this problem, we either had to modify the *Editor*'s code extensively to have it look for input in the real channel of the editing window, or to remove the *Workshop*'s dependence on *stdin*. We chose the latter option because the modification of code required was much simpler. Our solution was to dup the open file descriptor *stdin* and give this copy to the *Workshop*. As a result, the two processes no longer contend for the same file descriptor. This

¹*Unexec* is a utility that dumps an image of a running program into a file in order to restart it later at that point.

²Although the *Editor* has its own editing window and does not use the shell window that the other thread uses for input (which is really *stdin*), its code is written in such a way that the core of the *Editor* expects user inputs from *stdin* by having some routines map the channel associated with the editing window to *stdin*. The two threads' dependence on the input stream created a confusion that led to system failure. See Figure 3-2.

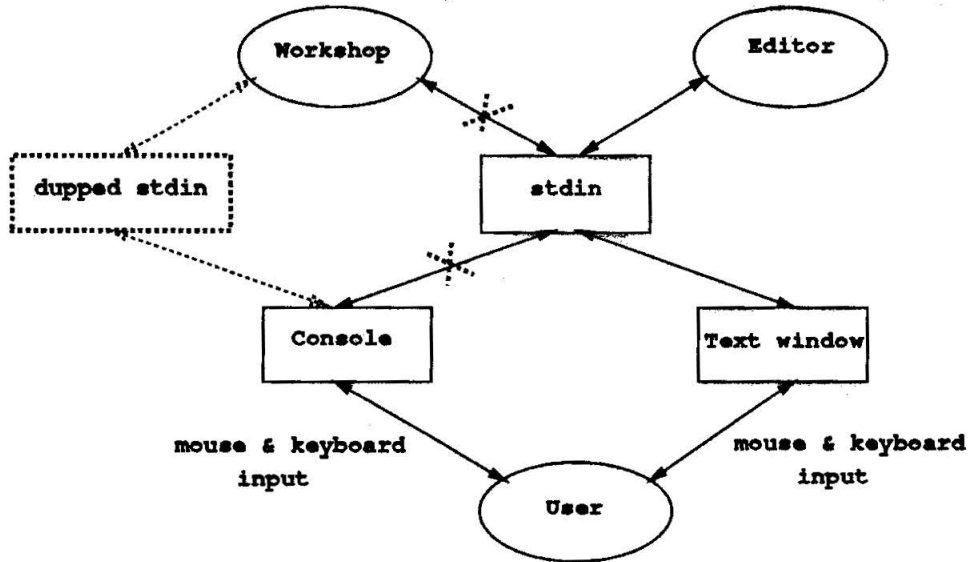


Figure 3-2: resource contention: the communication channel *stdin* was “dup”ed and the copy was given to the Workshop process to avoid contention.

problem is graphically described by Figure 3-2.

3.2.3 Problems with UNEXEC

Matisse takes a rather long time to start up because it has to load in the necessary files (such as the rules for the inference engine) needed for operation. Because of this lengthy process, it is convenient and useful to terminate a session of work by freezing the image of the program and dumping out to an *a.out* format file capable of being restarted by the operating system. This is accomplished by a mechanism called *unexec*. However, making *unexec* work for a multi-threaded task requires that we properly start up the threads, coordinate their terminations, and clean up after they exit.

First of all, in order for the program to be restarted after an *unexec*, we must call the thread initialization routine to set up the internal system resources needed for operation since the individual thread states were not saved by *unexec*. Since the effects of calling the thread initialization routine are not idempotent, in many systems, this

thread initialization routine is automatically called by the start up code, and only once. Our *C threads* [CD88] package, for example, saved a static flag to indicate whether or not the initialization routine was called and is used to avoid initializing more than once. This static flag prevented the initialization routine from being called when we restarted from the *unexec* image. To resolve this problem, we modified the thread library and made the flag available as a global variable so that the program can set it and force the C start up code to call the initialization routine at the next start up.

Even if the thread initialization routine were being called every time, we still were not able to restart the dumped image of the program if it was not cleaned up properly before the *unexec* dump. The problem here was that the thread initialization routine did not set up new internal data structures to support the startup threads if these structures were present in the dumped image from the previous run. Worse yet, it did not re-initialize these structures corresponding to the states of the newly started threads, and left the stale data in these structures to corrupt the threads, which led to system crashes. To remedy this problem, we extended the thread-exit code to free all such data structures, thus forced the thread initialization code to create new ones each time the program was restarted from an *unexec*.

Finally, we must ensure that the thread calling the *unexec* code be the last one to exit. This requirement ensures that all other threads have cleanly exited and can be safely restarted. This was accomplished by making this last thread wait for the others to terminate via a *join* operation. In our application, when the user issues a termination command, the Editor tells the Workshop to terminate before calling the thread exit routine to terminate itself (Figure 3-3). The Workshop, on the other hand, waits for the join operation to succeed before calling *unexec* to dump out the program. When the join operation succeeds, we are guaranteed that the Editor thread has exited cleanly (i.e. the thread-exit routine had time to clean up all the internal data structures), and it is safe to proceed with the *unexec* dump.

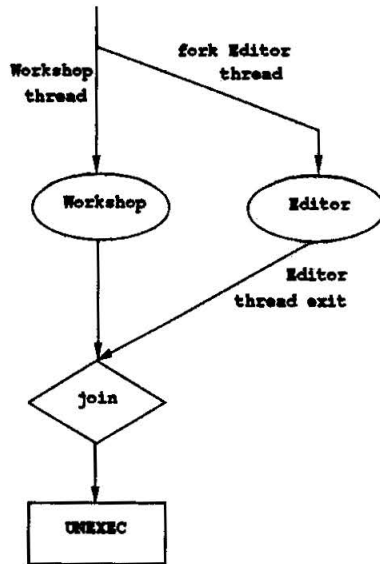


Figure 3-3: synchronization on thread exit. To ensure that the Editor thread exits cleanly, the Workshop thread waits for the join operation to complete before proceeding to call *unexec*

3.3 Sharing memory

This third and final stage of the experiment involved the removal of large IPC messages between the Workshop and Editor threads. In its place, we implemented a communication protocol using direct memory lookup, specifically keeping all data in the heap of the Workshop thread and allowing the Editor to read them as needed. The object lookup is facilitated by a thin layer of routines that read the needed data from the Workshop's heap and convert it to the format usable by the Editor. This layer of code is completely transparent to both the Workshop and the Editor. To protect the integrity of the data during the critical time when data are read and converted, the execution of this code and the Workshop's garbage collector must be mutually exclusive³. This is accomplished by a simple locking scheme described in section 3.3.2.

³The Workshop's update operations and the garbage collector are already mutually exclusive since they are part of a single thread of control. Read-Write synchronization is discussed in section 3.3.1.

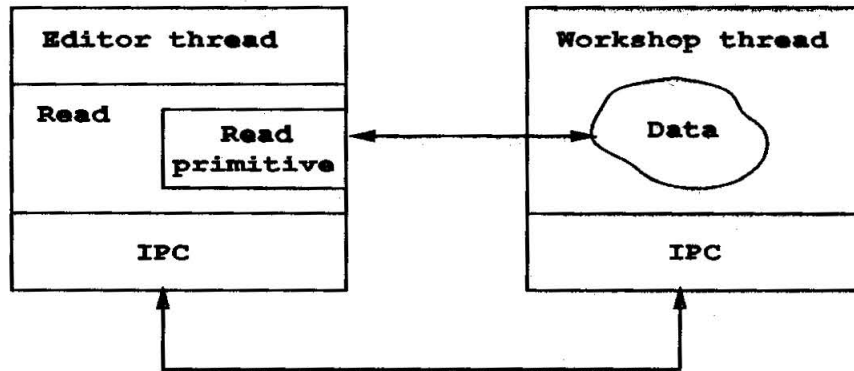


Figure 3-4: Data sharing: The new read primitive transparently reads the data directly from the Workshop heap and converting it to the format usable by the Editor, thus eliminating the need for IPC.

3.3.1 Object modeling

Having merged the threads, the Workshop and Editor now shared the same address space and can access each others' data directly. At this time, we no longer needed to keep two separate copies of data in both the Workshop and the Editor. Since the Workshop process does most of the computation with objects, our decision was to let the Workshop thread manage all the data. Furthermore, since the two threads use the data in different formats, we implemented the memory-read primitives to perform a conversion from the Workshop format to that of the Editor. In effect, this read operation is transparent to the Editor, but the data comes directly from the Workshop's storage. Figure 3-4 illustrates the design of the new read mechanism, which bypasses the interprocess communication channel and enables data to be shared between the Workshop and the Editor.

With the Workshop modifying objects and the Editor reading them to update the display, we must ensure that these two operations do not contend with one another. For this, a simple locking scheme implemented at the object-base level (i.e. one mutex locks the entire object base) or at the finer object level (i.e. one mutex per object) would ensure the mutual exclusion of reading and writing on the same object. However, certain systems do allow us to forgo the expense of some locking if we

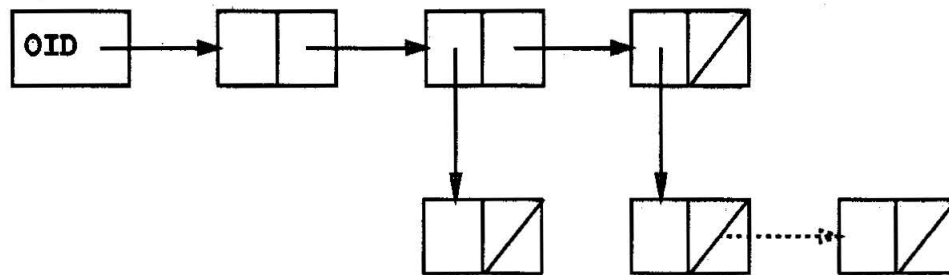


Figure 3-5: Read/write synchronization: If the write routine connects or disconnects the link to new/old object data in a last, atomic step, there won't be any conflict with the read routine looking at the same piece of data at the same time.

carefully take advantage of its semantics. For example, in *Matisse*, the Workshop first modifies the object, then sends a notice to the Editor about the changed status. The Editor then reads the object to update its display. With this semantic, the Editor never reads an object for new information unless the Workshop has finished changing it and sent its notification. Prior to receiving the notification, any Editor look up of the object (due to a screen refresh, for example) would be interested in the old object. In the rare event where the Editor's object lookup on an object is performed in the small time frame when the Workshop just modified an object, and the notification of the changes is being sent to the editor, what the Editor gets back from the read operation is the new version of the object, not the old one it is seeking. This "error" is perfectly harmless, since the Editor will receive the notification of that object being modified soon, and would update its display with the new information anyway. To ensure correctness, we must take care that when adding or removing object attributes, the update functions add the link to new information or cut the link from data to be deleted as one last step (Figure 3-5). Prior to this step, any read done by the Editor will see the old object; after this step, the Editor's read will get the modified object.

It is worth mentioning here that other implementations for this type of data sharing is possible, such as providing a separate thread to manage the entire shared object base, including synchronization primitives and garbage collection (Figure 3-

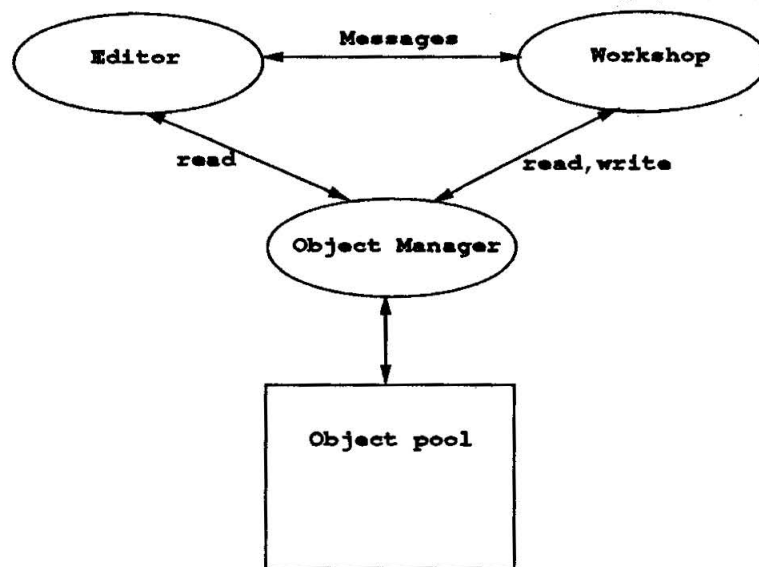


Figure 3-6: Memory management alternative: the object management mechanism can be implemented by an independent thread. Synchronization of object access between the Workshop and Editor threads can be handled by the Object Manager.

6). However, due to the architecture of *Matisse*, it was much simpler, and just as effective, for us to work with the scheme described above. Using a general data sharing method would have required us to rewrite many memory management operations and the garbage collector, which are already present as part of the Workshop. Designers of future systems should carefully consider exploiting the existing architecture before resorting to this general scheme.

3.3.2 Garbage collection considerations

Having the two threads sharing the same heap of data, we had to consider carefully the memory management issues, specifically the garbage collector's operation, to protect the system from object movements due to the garbage collector during the critical read/write regions. In this version of *Matisse*, the Editor accesses Workshop's heap of data directly and must be protected from the Workshop's garbage collector during a critical time when it is holding pointers to Workshop's object and performing the

reading and conversion between object formats. If this provision is not made, it is possible that the Workshop's garbage collector is called to perform its job and relocates the data being pointed to by the Editor amidst its read operation, which results in a segmentation fault as the Editor attempts to read the data from the pointer that is no longer valid. This mutual exclusion between the Workshop garbage collector and the Editor's read primitive can be achieved by a simple mutex lock to be seized by either of these two entities as they try to get to the data. Locking can be done at a finer object level, but the infrequent Editor's look-ups make it costly and infeasible to implement a mutex for each object.

This simple locking scheme is sufficient if there is never a need for obtaining the mutex lock again during the critical section where the mutex lock is already held by either the garbage collector or the read primitive. However, in our system, there is a problem when the memory of the object heap runs low during the critical section of the Editor's read operation. In this case, the Editor thread would block by running the garbage collector, which would block waiting for the mutex lock to be released by the Editor. This deadlock can be resolved by requiring the Editor to yield the mutex lock to the garbage collector and redo its read operation later⁴. A very simple heuristic which also worked for us was to let the garbage collector abort when the mutex lock is held by the Editor thread in its critical segment, and to rely on the Workshop to call the garbage collector soon after the Editor releases the mutex lock⁵.

Thus, with the effort of porting *Matisse* to the new multithreaded environment, of merging the singlethreaded tasks to a multithread task, and of replacing most of the IPC byte transfers with the new memory-sharing protocol, we have produced a new system which is identical in appearance to the original one, but with much better speed and responsiveness. The performance improvement is discussed in the following chapter.

⁴The *redo* can be done simply by having the read primitive release the lock, call the garbage collector, and then call itself with the original arguments.

⁵This works in *Matisse*, because the Editor thread seizes the mutex lock very infrequently, and the Workshop thread does all of the information processing which almost always notices the need for garbage collection before the memory is depleted.

Chapter 4

Performance Evaluation

This chapter reports and analyzes the performance data collected with versions of *Matisse* from each stage of the experiment, illustrating the benefits and performance improvements resulting from having light-weight threads and memory-sharing inter-process communication (IPC).

As described by the previous chapters, *Matisse* evolved through three stages. Version 1 is a “straight port”, featuring a one-to-one mapping of UNIX processes to MACH singlethreaded tasks. Version 2 is the merge of these singlethreaded tasks into one multithreaded task with the IPC mechanism unaltered. And finally, version 3 is the multithreaded version similar to version 2, but most of the IPC messages have been replaced by direct read/write of data in shared memory between the threads. However, it was also necessary to implement version 3 in two steps: in the first we only reduced the number of bytes being transferred (Version 3a), and in the second we reduced both the number of IPC messages and bytes (Version 3b). It became apparent during the experiment that reducing the number of IPC messages sent would provide us even better performance improvement than by reducing the number of IPC bytes alone. The rationale for this two-steps implementation will be discussed later in this chapter. Also, we made no attempt to compare the performance of the ported system to the original one running on the UNIX platform since there are simply too many system differences between UNIX and MACH to have a meaningful comparison. Therefore, version 1 of *Matisse* serves as the baseline with which all performance

measurements are compared.

4.1 Timing measurements

To effectively illustrate the performance improvement from the rearchitecture described in the previous chapter, two scenarios with significant IPC overhead were chosen as benchmark tests for each version of the system as it evolved from two singlethreaded tasks to one multithreaded task with shared-memory IPC. The IPC overhead of these scenarios, mostly involving passing data back and forth between the Workshop and the Editor, is estimated at 40%. We use our *estimate* here because the actual execution time, comprised of *user time*¹ and *system time*², cannot be precisely measured across thread boundaries. The task of breaking a message into data packets, sending the packets across the wire, and reassembling the data stream, starts in one thread and ends in another. The system timing utilities available to us could not be used to measure system execution time across thread boundaries. Thus, in order to coarsely measure the percentage of IPC overhead in a transaction, we computed, from time stamps, the *real* time it takes to send a message from the sending thread to the receiving thread. Since we can only measure this using *real* time, the number we get is, of course, slightly larger than the actual real execution time of the two entities due to some other system threads (such as the scheduler, the window manager) having the CPU in between. Other than the essential system threads, the experimental threads are the only user threads running in an extremely lightly loaded system. The CPU cost of the system threads, being constant across all measurements, does not affect the qualitative analysis of the performance profile, and amounts only to a small offset factor in the quantitative analysis. Along with the timing measurements, a pair of counters is also implemented to count the number of messages and the number of bytes being sent via the communication sockets.

The execution time of each scenario is measured by obtaining the total running

¹*user time* is the total amount of time the system spends executing in user mode

²*system time* is the total amount of time the system spends executing on behalf of the thread or process.

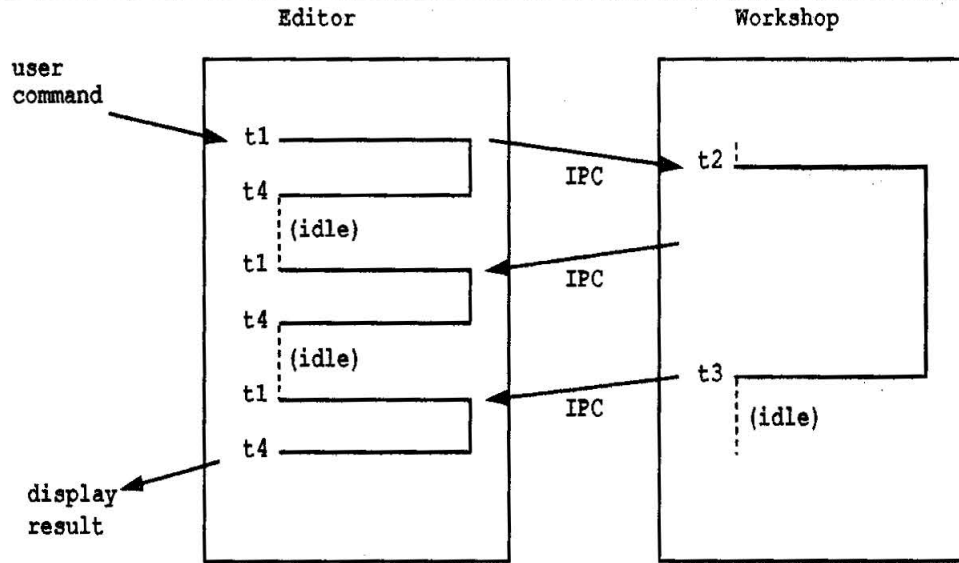


Figure 4-1: Timing measurement. This operation scenario contains 3 *Editor* transactions and 1 *Workshop* transaction. The total execution time is the sum of three $(t4-t1)$ and one $(t3-t2)$, in both user time and system time. Note that the idle time is not charged to the execution time of either thread.

time of all threads (or *tasks*, in version 1) between its start and end points. The acquisition of *user time* and *system time* is done by calling the system utility *getrusage*. With *getrusage*, we have a timing granularity of 1 microsecond, which is adequate for measuring transaction time lasting in the order of tens of seconds. The following paragraph and Figure 4-1 illustrate the timing measurement for a typical transaction.

Initially, both the Editor and the Workshop stay idle in their respective *select* loops waiting for some triggering event either from the user or from each other. This idle time is not charged to the executing time of the threads in our measurements. Immediately after the user issues a command (via the Editor) to start a transaction, the Editor takes a snapshot of its running time, say $t1$, by calling *getrusage*. Since the communication is asynchronous, after sending data to the Workshop, the Editor returns to its waiting loop to wait for another event. However, before returning to the waiting loop, it makes another call to *getrusage*, $t4$, which is used later to compute the Editor's recent execution time for the transaction. The Workshop on the other

hand, makes a call to *getrusage*, t_2 , immediately upon receiving the data and then proceeds to read data from the socket. After assembling the message, the Workshop carries out the requested job, and interactively sends the results back to the Editor as they become available. At the end of the job, the Workshop once again samples the time, t_3 , and returns to its waiting loop. The differences between t_4 and t_1 , t_3 and t_2 are the Editor's and Workshop's execution time for the transaction. By summing the execution times, we obtain the total execution time of the system for the transaction. Note also that each of our test scenarios involves a number of transactions from both the Editor and the Workshop, so the execution time of the scenario is the sum of execution times of all the transactions involved.

4.2 Test cases

The first test case is the startup sequence of actions that takes place as each user logs into *Matisse*. This scenario has a high percentage of IPC activities because the Editor and the Workshop must communicate with each other extensively to set up the environment for the user. The setup process involves the Editor getting the numerous program objects from the Workshop and initializing the display screen. This interprocess communication is done via sockets in version 1 and 2, and via direct memory sharing in version 3 of the system.

The second test case involves another IPC-intensive sequence of actions: modifying and saving a program object. In order to save a text object, the Editor first sends the modified object to the Workshop where it is validated, updated, stored, and sent back to the Editor to be displayed. In addition, the Workshop uses its rule base to determine and update the necessary changes in the system configuration.

Although the IPC overhead in both test cases is high (about 40%), they are slightly different in composition. The first test case involves numerous small IPC messages, while the second test case is comprised of fewer, but larger, IPC messages. This difference plays a key role in explaining the amount of system performance improvement and will be discussed in the following sections.

4.3 Experimental results and analysis

Before discussing the experimental results, it is necessary to label the various versions and components of the system for comparison. First, the differences between versions of *Matisse* are highlighted to be later used in the performance analysis.

- Version *V1* comprises the *singlethreaded* Workshop and Editor tasks, each communicating with the other by sending messages via sockets.
- Version *V2* is a *multithreaded* task with concurrent Workshop and Editor threads, each still communicating with the other by sending messages via sockets.
- Version *V3a* is a *multithreaded* task with concurrent Workshop and Editor threads, and most of the bytes formerly communicated over sockets are now communicated by a shared-memory read/write protocol. This version of the system reduces significantly the number of IPC bytes being sent via sockets.
- Version *V3b* is a *multithreaded* task with concurrent Workshop and Editor threads, and most of the IPC messages have been replaced by a shared-memory read/write protocol. This version of the system reduces significantly the number of *both* the IPC bytes and IPC messages being sent via sockets.

The followings are symbols and labels used in the discussion on performance analysis:

- Scenario 1 and Scenario 2 are the first and second test cases described in the previous section, respectively.
- *CPU Time* is the sum of *user time* and *system time* of all relevant threads.
- The percentage change of either CPU time or IPC bytes is computed according to the formula $(v2 - v1)/v1 * 100$. The negative sign indicates a reduction of cost, or, equivalently, improvement of performance.

	<i>Change in IPC bytes</i>	<i>Change in CPU Time</i>
$V1 \rightarrow V2$	-3.80%	-5.30%
$V2 \rightarrow V3a$	-31.25%	-11.11%
$V1 \rightarrow V3a$	-33.87%	-15.82%

Table 4.1: Performance Improvement, test scenario 1

	<i>Change in IPC bytes</i>	<i>Change in CPU Time</i>
$V1 \rightarrow V2$	-1.87%	-5.05%
$V2 \rightarrow V3a$	-40.86%	-15.38%
$V1 \rightarrow V3a$	-41.97%	-17.11%

Table 4.2: Performance Improvement, test scenario 2

4.3.1 Light-weight threads and shared memory

The first set of experiments pits versions $V1$, $V2$, and $V3a$ together to compare the performance improvements resulting from the migration of a distributed application from two singlethreaded tasks to a multithreaded task and then to a multithreaded task with shared memory interprocess communication. Table 4.1 and Table 4.2 illustrate the percentage reduction in IPC bytes and CPU time between every two versions compared.

As seen in Table 4.1 and Table 4.2, the improvement between $V1$ and $V2$ is indicative of the light-weight thread issues. Unfortunately, only a slight improvement is observed here because the scheduler of our MACH 2.0 does not provide light-weight threads with much advantage over conventional processes or tasks, and crossing the kernel boundary is expensive (as much as for processes). With a smarter scheduler such as one in MACH 2.5³, we would expect to win in the event that two consecutive threads occupying a processor execute within the same task, and therefore, share the same address space. For example, our system (HP9000 series 350) uses a virtual cache which can hold information for one address space at a time. A smarter scheduler

³Although MACH 2.5 exists for the HP9000 machines, it was not readily available for this experiment.

could notice that the next thread running on the processor uses the same address space as the previous one, and avoid any unnecessary cache flush. Thus, by allowing the next thread to use valid data in the cache rather than causing expensive cache misses (after an unnecessary cache flush), an intelligent scheduler can cut the cost of a context switch on a virtual cache machine [CMS9b].

The tables also show that the performance improvement between *V2* and *V3a*, however, is much more significant. By shifting from socket-based IPC to shared-memory IPC, we reduced the number of bytes to be sent via sockets by 31% and improved the overall system speed by 11%. Since roughly 40% of the original system overhead is IPC related, we have effectively slashed the IPC overhead by approximately 25%.

The comparison between *V1* and *V3a* indicates the total performance improvement from having both the benefits of light-weight threads and memory-sharing IPC. This comparison is less specific than the previous two comparisons since it does not distinguish the contribution of each component of interest, but does provide the total effect of the performance improvement. Note also that the cumulative figure presented in this comparison is not the simple addition of the previous two comparisons, but slightly less. This is correct since the comparison between *V1* and *V3a* is *the improvement of version V3a with respect to version V1*, not *the improvement of version V2 with respect to version V1* and *the improvement of version V3a with respect to version v2*.

In addition, second order effects exist which indirectly improve system performance. These beneficial factors occurred naturally as part of our re-architecture of *Matisse* and required no additional work. For example, with the data sharing in version *V3a*, the Editor no longer has to keep its local copy of the data, saving 400K of memory at run time. This version is leaner than version *V2*, takes up less memory, and can thus run faster. Being smaller in size, the merged version has less paging overhead, takes much less time to load into memory, and is less likely to be swapped out during a context switch.

	<i>Change IPC bytes</i>	<i>Change IPC msgs</i>	<i>Change CPU Time</i>
<i>V1</i> → <i>V2</i>	-3.80%	0%	-5.30%
<i>V2</i> → <i>V3a</i>	-31.25%	0%	-11.11%
<i>V1</i> → <i>V3a</i>	-33.87%	0%	-15.82%
<i>V3a</i> → <i>V3b</i>	-80.88%	-80.50%	-28.50%
<i>V1</i> → <i>V3b</i>	-87.35%	-80.50%	-39.81%

Table 4.3: Performance Profile, test scenario 1

4.3.2 Bytes vs messages

Although the use of shared memory for interprocess communication already provide us with some performance improvement, it was discovered that we can do much better than this by not only reducing the number of bytes being sent between threads, but also the number of IPC messages. This section examines the result of two slightly different variations of version *V3*'s memory sharing techniques: version *V3a* which reduces the IPC bytes, and version *V3b* which reduces the number of IPC messages as well. But in order to understand the causes and benefits of this design decision, we must first examine the system's interprocess communication activities.

The original IPC in *Matisse* was done by processes sending requests and receiving data via sockets. In *Matisse*, the requests and data are sent back and forth in *messages*. Each *message* contains a [OID,Slot,Value] triplet. The OID is the object's unique ID, and the Slot-Value pair is part of the object's content. An *object* is represented by a set of triplets with the same OID (see Figure 2-5).

Typically, the Editor would send to the Workshop a data request for an [OID,Slot], or send a notification regarding a particular [OID,Slot,Value] that has been modified. The Workshop, upon receiving either the request or the notification, performs the necessary search and update functions, respectively, and then sends the entire object back to the Editor. Depending on the nature of the request and notifications, tens of messages and thousands of bytes are sent back and forth between the processes for each request or notification.

Versions *V1* and *V2* of *Matisse* use the above method of interprocess communi-

	<i>Change IPC bytes</i>	<i>Change IPC msgs</i>	<i>Change CPU Time</i>
<i>V1→V2</i>	-1.87%	0%	-5.05%
<i>V2→V3a</i>	-40.86%	0%	-15.38%
<i>V1→V3a</i>	-41.97%	0%	-17.11%
<i>V3a→V3b</i>	-85.23%	-82.95%	-20.44%
<i>V1→V3b</i>	-91.42%	-82.95%	-34.05%

Table 4.4: Performance Profile, test scenario 2

cation. Version *V3a*, however, eliminates most of the byte transfer by allowing the Workshop to only send back the small messages containing the fixed length [OID,Slot] pairs of data, while making the Editor perform the lookup and copy of large *Value* data fields directly from the Workshop’s memory. As illustrated by Table 4.1 and Table 4.2, the amount of data being sent across sockets is reduced by one-third, boosting overall system performance by 16%. Consider the fact that 40% of version *V1*’s overhead is IPC, a reduction of 15.8% total system time is equivalent to roughly 39.5% reduction of the original IPC overhead, knowing that all other activities remain unchanged.

However, we can do much better in Version *V3b* by reducing not only the number of IPC messages being sent between the two threads, but also the number of IPC messages in transit. In this version, the Workshop only sends *one* message, containing a single [OID], to the Editor for each object that is modified. The Editor thus has a greater responsibility to look up the Slot and Value attributes of the desired object. In contrast to version *V3a*, where there were still tens of [OID,Slot] messages sent (same OID, but different Slots) for each modified object, version *V3b* has only one message sent for each modified object. Here, we reduced both the IPC bytes and messages (over 80% from *V3a*) and obtained significant improvements. Since most of the IPC messages are small, and the cost of sending messages up to a certain size is constant, the benefit is not fully realized if we just reduced the IPC bytes. For example, in our system, the cost is the same for messages up to 8K bytes in size, so it is not very beneficial to send just small data packets. This explains why

the large reduction in IPC byte transport in version *V3a* did not yield comparable performance improvement. In test scenario 1 where the communication activities involve many small messages, a reduction of 80.88% of IPC bytes and 80.50% of IPC messages reduced the overall CPU time by 28.50%. On the other hand, in the test scenario 2 where the communication pattern is comprised of fewer but larger messages, a comparable reduction of 85.23% of IPC bytes and 82.95% of IPC messages led to a much smaller reduction of 20.44% overall CPU time.

The performance data in Table 3 and Table 4 show that, in version *V3b*, we have reduced the byte traffic by as much as 85% and CPU time by 28% over version *V3a*. Comparing this performance with the original version *V1*, we have achieved approximately 40% reduction in CPU time in running our set of test cases. This reduction means we have virtually eliminated the original system overhead related to IPC of 40%. Obviously, there is still a trickle of IPC messages present in the system since we have not completely abandoned the practice, but the performance improvement resulting from any secondary effects have covered this cost. In addition, the performance gained from these secondary effects also covered the cost incurred from the shared-memory read/write protocol.

Chapter 5

Conclusion

5.1 Summary

As stated in chapter 1, this project did not attempt to prescribe the proper development of new applications for a multithreaded platform, but rather our goal was to establish a useful framework for future users to accomplish successful migration of existing applications in order to obtain better performance with minimal efforts. It was our projection and successful finding that existing singlethreaded applications with a major overhead of message-based interprocess communication can benefit greatly from the light-weight threads and the shared-memory interprocess communication mechanism offered by a multithreaded environment.

We also introduced *Matisse* in the first chapter as the target of our experiment and discussed the reasons why it is a suitable candidate: its high overhead of message-based IPC and the fact that the processes can be merged without any loss of functionality. To the end user, the re-architected *Matisse* looks and behaves just like the former one, only faster and more responsive.

Chapter 2 discussed the system architecture of *Matisse* in detail. It also identified the subsystems to be ported and illustrated the coordination between them. The chapter focused especially on the similar internal data representation of each component and the communication pattern that was the main target of the re-architecture. By providing some details about the operation and coordination model of *Matisse*'s

components — the Workshop and the Editor processes — this chapter provided the readers with the background information needed to understand the architectural re-design discussed in chapter 3.

Chapter 3 presented the migration process taking the application from the singlethreaded UNIX environment to the multithreaded MACII environment. We discussed the three phases of the migration process in great detail from the initial port to the merging of singlethreaded tasks to the replacement of message-based IPC with shared-memory IPC. Throughout the discussion, we enumerated the problems and difficulties faced during each phase of the process, examined their causes, and reported our solutions. The problems we faced ranged from the well known *thread synchronization* problem to the more peculiar *uncrcc* routines. It is hoped that the reader will find this section useful in considering and attempting to perform future ports of applications from singlethreaded to multithreaded environments.

Chapter 4 reported the performance analysis and discussed our findings. We presented the techniques utilized to obtain the experimental data, especially our timing measurements of the threads to account for both IPC overhead and CPU costs. We further described the testing scenarios and analyzed their operations and resource composition to identify the sources of costs and benefits in the performance analysis.

And finally, this chapter attempts to summarize the whole process, to assess our results, and to take a brief look ahead into the possible future extensions of the current effort.

5.2 Discussion and future considerations

5.2.1 Shared-memory architecture

In general, the result of the experiment matched our initial expectation. As reported in chapter 4, we virtually eliminated the cost of message-based IPC by replacing it with shared-memory IPC. Of course, the replacement of message-based IPC by shared-memory IPC is not possible for existing applications under the traditional sin-

gletheaded environment since singlethreaded processes cannot share memory. Even with operating systems providing interprocess shared memory mechanisms such as UNIX System V, shared memory can only be done by explicitly allocating and managing specific regions of memory. With a system like *Matisse* which has two separate LISP environments, sharing LISP objects via an explicitly managed region of memory would require implementing a garbage collector while not making use of the ones already available in each LISP environment. This redundant work and potentially complicated process defeats the purpose of a quick and inexpensive migration of existing application to the new platform.

Furthermore, we have formulated a rough guideline for the migration process. By laying out the steps and identifying the potential problems associating with each step, we hope future migrations can be done quickly and painlessly.

In the migration process, however, not all components of a multi-process application need to be merged as concurrent threads, nor would we want to do so. Although migrating singlethreaded processes to concurrent threads within a multithreaded task enables the threads to communicate cheaply via shared-memory, the trade off is that we lose the generality of the original process model: losely-coupled processes can be run simultaneously across different machines. On the other hand, tightly-coupled threads sharing memory are restricted to the same machine. In effect, by porting and merging processes as concurrent threads, the threads must now be executed on the same machine, whereas they could be run on different machines before. Therefore, candidates of this re-architecture process should be those that have a large IPC overhead, and always reside on the same machine. Many existing applications fit this requirement, and are good candidates for migration.

5.2.2 System threads vs user threads

The concept of merging singlethreaded processes to gain shared-memory communication capability can be applied to coroutine packages as well. Such an approach does not require a migration to an operating system with multithreaded support, but rather the merging of processes to *user threads* of a coroutine package. This merging

process is similar to that described in chapter 3. If the user already has a coroutine package that manages the scheduling of user threads, the merge can yield performance improvement by the resulting user threads having shared-memory IPC.

In order to speculate about the relative performance of an application using user threads versus the performance of the same application using kernel threads, some background on threads scheduling is needed. First of all, as a scheduler picks a thread to run on a processor in a time-sharing fashion, each kernel thread gets a time slice. User threads, however, multiplex within the process' single time slice, and are not seen by the system scheduler.

Thus, with the scheduling issues illustrated, we would expect that on a lightly loaded system, the performance of an application using coroutine package (with user threads) would be comparable or faster than the performance of the same application running in a multithreaded environment (with system threads). The reason for this is that on a lightly loaded system, ideally with only one process and its user threads, the process can have much CPU time. And within this one time slice, the user threads can slightly outperform light-weight kernel threads because they are lighter in weight and have the same ability to directly share memory.

However, in a heavily loaded system, we would expect the coroutine implementation to run slower since each system thread is a candidate to be scheduled on a processor, while the user threads are only recognized as one schedulable. Therefore, when there is competition among threads to run, a multi-system threaded application is given more opportunity to run. Another important difference between user and kernel threads is that in a multithreaded multiprocessor environment, the system threads can potentially be scheduled on several processors, exploiting the real concurrency, while the user threads in a coroutine package can be scheduled and run on only one processor at a time.

5.2.3 Multiprocessor implications

A logical next step would be to rehost *Matisse* onto a multithreaded, multiprocessor environment running MACH. The migration to this type of environment would require

no additional work beyond that described. In a multiprocessor MACH environment, each thread is a candidate for scheduling on any processor. If two or more threads sharing memory are run concurrently on different processors, the system transparently manages the shared memory addressed by the different CPUs. Given that the operating system for the multiprocessor machine is designed and implemented properly, then we would expect to see a multithreaded application run faster on a multiprocessor MACH machine than on a uniprocessor MACH machine as a result of the parallelism of the multiprocessor architecture.

To better take advantage of multiprocessor machines, however, this work can be extended to break up the code into many threads, hence “parallelize” the application whenever possible. The numerous threads can then be run in parallel across the processors, thereby effectively utilizing the hardware resources. For example, in *Matisc*, the Workshop’s garbage collector can be implemented as a separate thread. Also, object updates or queries can be done in parallel by forking a thread for each job, rather than simply doing them in serial.

In conclusion, the primary purpose of the work presented in this thesis is to establish a useful framework for the migration of existing singlethreaded applications to a multithreaded environment. Such applications can then undergo a rapid and simple re-architecture to replace the costly message-based interprocess communication mechanism with shared-memory IPC. It is also an intention of this work to explore, identify, and resolve the many common problems of the migration process. In doing so, it is hoped that future implementors will find this a useful guide in targeting and porting similar systems to multithreaded environments in order to achieve better performance with minimal cost and effort.

Bibliography

- [ABB⁺86] M.J. Accetta, R.V. Baron, W. Bolosky, D.B. Golub, R.F. Rashid, A. Tevanian, and M.W. Young. Mach: A new kernel foundation for unix development. *Proceedings of Summer Usenix*, page 5, July 1986.
- [AGS89] J. Ambras, P.K. Garg, and R. Splitter. The workshop: A team programming environment. *Proceedings of the 2nd European Software Engineering Productivity Conference*, pages 55–57, May 1989.
- [Art88] Artificial Intelligent Section. Lyndon B. Jonhson Space Center. *Clips reference manual*, 1988.
- [Bet] David M. Betz. *Xlisp: An object-oriented lisp*. Unpublished manual, pages 1-3. Available from author at P. O. Box 144, Peterborough, NH 03458, April 1989.
- [BS90] B.W. Beach and R. Splitter. Eclipse: A c-based inference engine embedded in a lisp interpreter. Technical Report HPL-90-213, Hewlett-Packard Laboratories, Software Systems Lab, pages 1-2, December 1990.
- [CD88] E.C. Cooper and R.P. Draves. C threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, School of Computer Science, pages 1-10, June 1988.
- [CM89a] D.L. Caswell and S. Marovich. Stl mach project retrospective. Technical Report STL-89-20, Hewlett-Packard Laboratories, Software Systems Lab, pages 2-4, August 1989.

- [CM89b] D.L. Caswell and S. Marovich. Stl mach project retrospective. Technical Report STL-89-20, Hewlett-Packard Laboratories, Software Systems Lab, pages 13-14, August 1989.
- [ea85] M. Satyanarayanan et. al. The itc distributed file system: Principles and design. *ACM 10th Symposium on Operating Systems Principles*, pages 35-50, December 1985.
- [FR86] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and inpterprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2):147-149, May 1986.
- [GBF+91] P. K. Garg, B. Beach, W. Fong, A. Ishizaki, and T. Pham. Matisse: A knowledge-based team programming environment. Technical report, Hewlett-Packard Laboratories, Software Systems Lab, in preparation, April 1991.
- [GYA90a] P.K. Garg, D. Young, and J. Ambras. An emacs-based software object editor. Technical Report HPL-90-72, Hewlett-Packard Laboratories, Software Systems Lab, pages 2-4, June 1990.
- [GYA90b] P.K. Garg, D. Young, and J. Ambras. An emacs-based software object editor. Technical Report HPL-90-72, Hewlett-Packard Laboratories, Software Systems Lab, pages 4-9, June 1990.
- [JRG+87a] A. Tevanian Jr., R.F. Rashid, D.B. Golub, D.L. Black, E. Cooper, and M.W. Young. Mach threads and the unix kernel: The battle for control. Technical Report CMU-CS-87-149, Carnegie-Mellon University, School of Computer Science, pages 1-2, August 1987.
- [JRG+87b] A. Tevanian Jr., R.F. Rashid, D.B. Golub, D.L. Black, E. Cooper, and M.W. Young. Mach threads and the unix kernel: The battle for control. Technical Report CMU-CS-87-149, Carnegie-Mellon University, School of Computer Science, pages 0-1, August 1987.

- [Pha90] Thuan Q. Pham. Creating a hierarchical network of objects that represents a file of programs. Bachelor Thesis in Computer Science and Engineering, Massachusetts Institute of Technology, pages 1-5, June 1990.
- [Ras86] Richard F. Rashid. Threads of a new system. *UNIX Review*, page 39, August 1986.
- [Sal66] Jerome H. Saltzer. *Traffic Control in a Multiplexed Computer System*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, page 2, June 1966.
- [TR87] A. Tevanian and R.F. Rashid. Mach: A basis for future unix development. Technical Report CMU-CS-87-139, Carnegie-Mellon University, School of Computer Science, pages 1-2, June 1987.