On Migrating a Distributed Application to a Multi-Threaded Environment

Thuan Q. Pham, Pankaj K. Garg – Hewlett-Packard Laboratories

ABSTRACT

Light-weight computation threads in a multi-threaded operating system promise to provide low-overhead computation and fully sharable addressing space not available in conventional process-oriented operating systems. Traditional distributed applications based on processes can be re-architectured to use concurrent threads in a multi-threaded platform to take advantage of faster context switches and shared-memory communication.

We investigated this expectation by porting an existing distributed application to a multi-threaded environment. As a result, we virtually eliminated the cost of message-based IPC, replacing it with shared-memory communication between threads.

In this paper we address the benefits, the difficulties, and the trade-offs of such a re-architecture. We also comment on some feasible architectures for migrating currently distributed applications to multi-threaded environments.

Introduction

The usual process abstraction [Sal66] has too many things anchored to it to meet the needs of aggressively concurrent applications. 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]. Furthermore, since the processes do not share resources, distributed applications with significant data sharing must communicate via expensive message-based interprocess communication (IPC) mechanisms.

Earlier efforts have tried to circumvent these problems by utilizing coroutine and user thread packages to simulate and manage multiple contexts, with shared memory, within a single process [Mar90, JRG+87]. However, since the kernel has no knowledge of such coroutines or sub-processes, these coroutine packages cannot take advantage of the operating system's scheduling services, and an application using them cannot utilize more than one processor in a multi-processor environment. Thus the questions of alleviating expensive context switching and expensive interprocess communication are not completely addressed by user-created coroutines and sub-processes.

The above problems have been addressed by operating systems supporting light-weight threads and shared resources (e.g., OSF1). The creation and maintenance of threads require lower operating system overhead than processes. A thread, when created, has access to all the process information in the task[1]. Since the computation threads share all resources within a task, including the memory address space, "inter-process" communication can be done cheaply and efficiently with shared memory.

A systematic reduction of heavy-weight processes to light-weight threads with shared memory, whenever possible, provides a substantial improvement in performance [ABB+86, FR86, JRG+87, TR87]. However, the threads facility restricts the architecture of distributed systems. Performance improvements come at the expense of the lost generality to the process model: processes are machine independent; threads may not be. Further cost is incurred by the effort spent in porting existing software systems to a new, multi-threaded platform. These issues are investigated by our re-architecture of an existing distributed application from a single-threaded environment to a multi-threaded one. We report on the problems and benefits of such a re-architecture.

The Experiment

The distributed application used in this experiment is *Matisse*, a knowledge-based team programming environment derived from the Workshop System [Cle88]. *Matisse* offers automated support for communication and coordination in team programming. Its architecture is illustrated in Figure 1. The core of each unit of *Matisse* includes an expert system, an object editor, and a graphical object browser, all residing in the programmer's workstation. In the current implementation, all components of *Matisse* are separate processes that run concurrently and share information via sockets. By merging some of these UNIX processes into threads within the same task, we can obtain significant performance improvement with light-weight threads and shared-memory IPC.

[1]A task with one thread is equivalent to a process. We use *task* to denote a multi-threaded process.

For this experiment, we have ported and merged only the *Workshop* and *Editor* processes (Figure 2). These processes are prime candidates to receive the benefits of the merge because they naturally reside on the same machine and share a large collection of data. The locality of these processes enable them to be merged without any loss of generality or usefulness, and their interprocess communication can benefit from direct memory sharing.

In addition to providing performance measurements, the experiment gives us some understanding of the difficulty of this porting process, 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. Our experience suggests that performance improvements do not come without costs and it is up to the designer to judge whether this approach is feasible for specific applications.

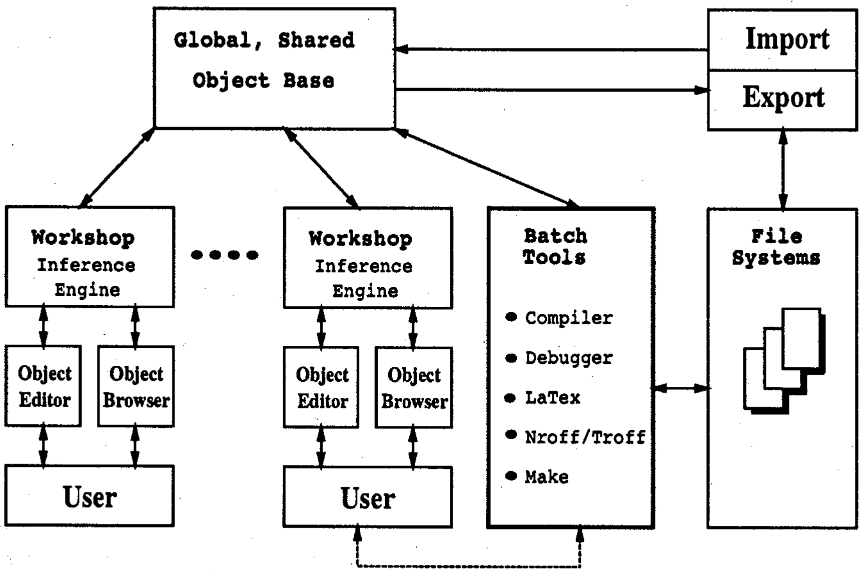

Figure 1: *Matisse* Architecture: A central shared information base is shared by all team members. Each person has an individual (active) information base which talks to various interactive tools.

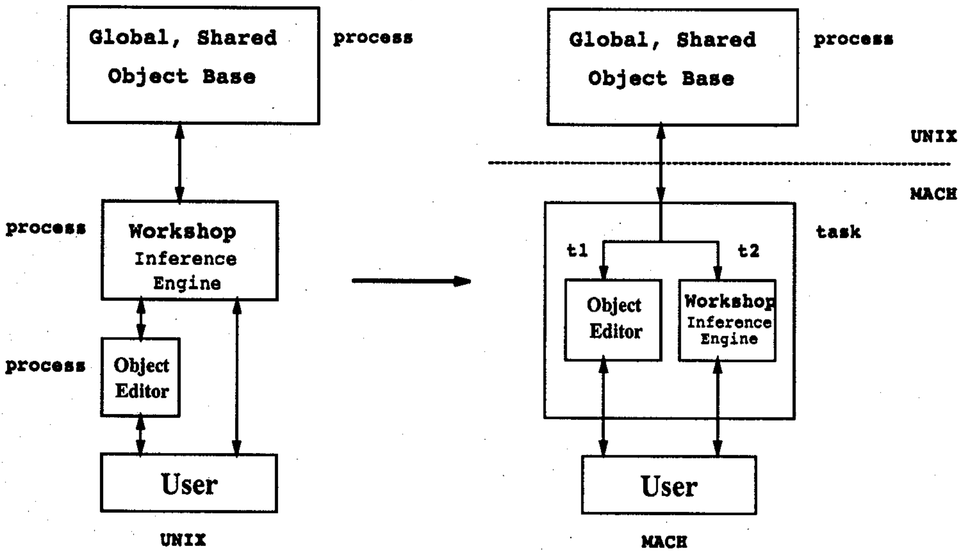
Global, Shared Object Base
process
process
Workshop Inference Engine
process
Object Editor
User
UNIX
Global, Shared Object Base
process
UNIX
MACH
task
t1
t2
Object Editor
Workshop Inference Engine
User
MACH

Figure 2: *Matisse:* experimental subset

System Re-Architecture

The experiment is divided into three stages. The first stage was a "straight port" of the Editor and the Workshop to the multi-threaded operating system, making each process a single-threaded task in the new environment. The second stage involved merging these two single-threaded 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 to shared data in memory, and a new communication protocol using these primitives. Figure 3 depicts the stages of the 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 experiment, discuss the difficulties that were encountered, and present our solutions.

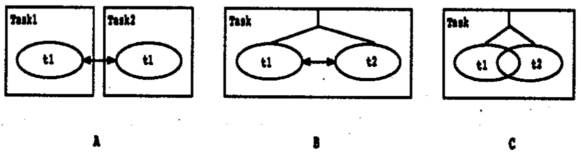

Figure 3: Re-architecture process: from disjoint single-thread tasks to multi-threads task with memory sharing capability; a) single-thread tasks, b) multi-thread task, c) multi-thread task with memory sharing.

From Processes To Single-Thread Tasks

In stage 1, we had the problem of having different names for [equivalent] system calls and a different file directory hierarchy. This can be rectified by systematically checking, identifying, and tracking down the right functions and files in the new environment. Although this was the simplest and most straight-forward step of the experiment, it was, however, not necessarily easy, depending on the portability of the existing system. This step could 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 [Sta84], was particularly hard to port since its complex code exploited many system-specific features, and exposed numerous system differences.

An interesting problem we encountered was the difference of implementation of certain operating system services. One example was the incompatible side-effects of the functions *regex/regcmp* across the two operating systems. One library routine 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 and contained no static internal state. It was not possible for us to re-implement such services to suit only our needs since other existing programs depended on them. Our solution was to define the functions and data structures necessary to use instead of the existing resources.

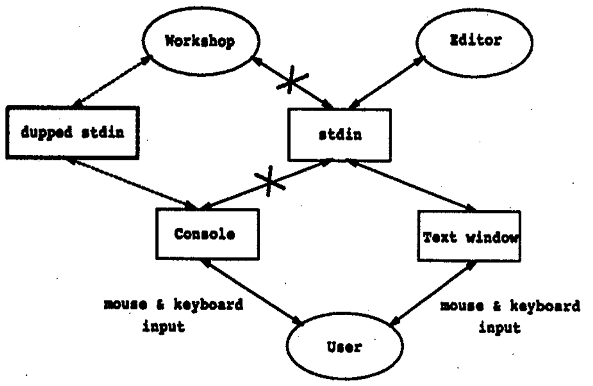

Figure 4: resource contention: Although the *Editor* has it own editing window and does not use the shell window that the other thread uses for input (which is really *stdin*), its code is written in such a way that the core of the *Editor* expects user inputs from *stdin* by mapping the channel associated with the editing window to *stdin*. The two threads' dependence on the input stream created a confusion that led to system failure. The communication channel *stdin* was "dup"ed and the copy was given to the Workshop process to avoid contention.

Merging Single-Thread Tasks To A Multi-Thread Task

In stage 2, we created a new *main* function to set up global, shared resources such as thread IDs, mutex and condition variables, and then fork the two execution threads. Since the interface allows only one argument to be passed to a thread at creation time, the *main* function must parse the command-line arguments, package them in data structures, and hand them to the appropriate thread creation routine.

I/O Contention

A thread-unsafe situation resulted when the merged threads both wanted to access the same file descriptor (most commonly *stdin*). To give one thread exclusive control of the file descriptor would require a major rewrite of the other thread. Our solution was to *dup* the overlapping file descriptors and give each thread a different handle. As a result, the two threads no longer contend for the same file descriptor. This problem is depicted in Figure 4.

Problems With UNEXEC

Applications often use a mechanism called *unexec* to freeze the image of the process and dump it out to an *a.out* format file capable of being restarted. 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, 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 because the individual thread states are 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, only once. In our *C threads* [CD88] package, for example, a static flag 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 forced the C start up code to call the initialization routine at the next start up.

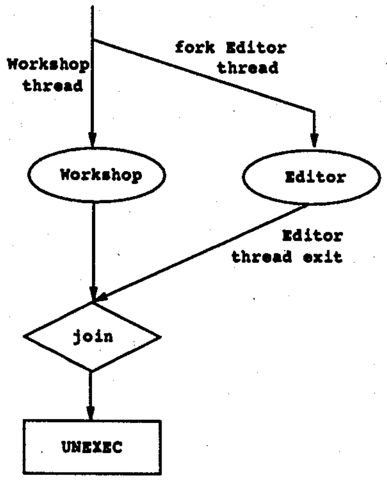

Figure 5: Synchronization on thread exit. To ensure that the Editor thread exits cleanly, the Workshop thread waits for the join operation to complete before proceeding to call *unexec*

Even if the thread initialization routine were being called every time, we still could not 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 tried to use the internal data structures for the threads already present in the dumped image from the previous run, but failed to re-initialize these structures corresponding to the states of the newly started threads. The stale data in these structures corrupted the threads and led to system crashes. To remedy this, we extended the thread-exit code to free all such data structures, forcing the thread initialization code to create new ones each time the program was restarted from an *unexec.* To ensure that the thread calling the unexec code was the last one to exit, we synchronized all thread exits via the *join* operation (Figure 5).

Sharing Memory

Having merged the threads, the Workshop and Editor now shared the same address space and could access each others' data directly. At this time, we no longer needed to keep separate copies of the data in both the Workshop and the Editor. Since the Workshop process does most of the computation with the data, we let the Workshop thread manage all the data. A transparent layer of memory-read/write primitives was implemented to perform object read and conversion from the Workshop format to that of the Editor (Figure 6).

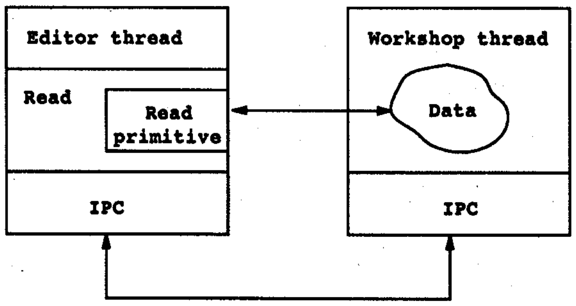

Figure 6: Data sharing: The new read primitive transparently reads the data directly from the Workshop heap and converting it to the format usable by the Editor, thus eliminating the need for IPC.

It is worth mentioning here that other implementations for this type of data sharing are possible, such as providing a separate thread to manage the entire shared object base, including synchronization primitives and memory management techniques (Figure 7). However, due to the architecture of *Matisse,* 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 a more general scheme.

Garbage Collection Considerations

For an application with an embedded LISP environment, we must carefully consider the memory management issue. In *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 objects and reading them. This contention is handled 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[2].

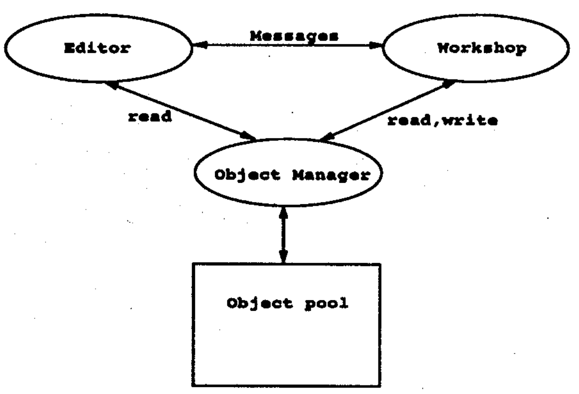

Figure 7: Shared-memory alternative: the object management mechanism can be implemented by an independent thread. Synchronization of object access between the Workshop and Editor threads can be handled by the Object Manager.

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 is resolved by requiring the Editor to yield the mutex lock to the garbage collector and redo its read operation later[3].

Performance Evaluation

As described by the previous sections, *Matisse* evolved through three stages. Version 1 is a "straight port", featuring a one-to-one mapping of one UNIX process to one single-threaded task. Version 2 is the merge of these single-threaded tasks into one multi-threaded task with the IPC mechanism unaltered. Finally, version 3 is the multi-threaded version similar to version 2, but most of the IPC messages are 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: V_{3a} reduced the number of IPC bytes being transferred; and V_{3b} reduced both the number of IPC messages and bytes. Version 1 of *Matisse* serves as the baseline with which all performance measurements are compared[4].

Timing Measurements

To effectively illustrate the performance improvement from the re-architecture described in the previous section, two scenarios with significant IPC overhead were chosen as benchmark tests for each version of the system as it evolved from two single-threaded tasks to one multi-threaded task with shared-memory IPC. The IPC cost of these scenarios, mostly involving passing data back and forth between the Workshop and the Editor, is estimated at 40% of the system overhead. We use our *estimate* here because the actual execution time, comprised of *user time* [5] and *system time* [6] , 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.

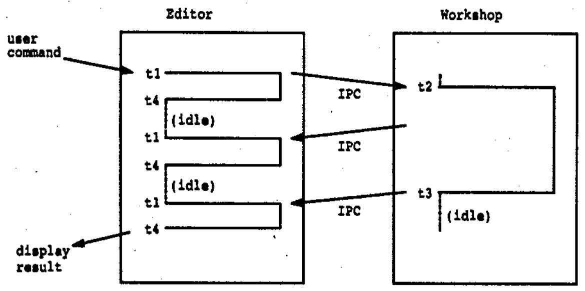

Figure 8: Timing measurement. This operation scenario contains 3 *Editor* transactions and 1 *Workshop* transaction. The total execution time is the sum of three (t4-t1) and one (t3-t2), in both user time and system time. Note that the idle time is not charged to the execution time of either thread.

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 execution time of the two entities due to some other system threads (such as the scheduler, the window manager) utilizing the CPU in between. In our test machine,

[2]A thorough coverage of concurrent programming, threads, and mutexes can be found in [Bir89].

[3]The *redo* is done simply by having the read primitive release the lock, call the garbage collector, and then call itself with the original arguments.

[4]We made no attempt to compare the performance of the ported application to the original one running on the UNIX platform since there are simply too many system differences between the two operating systems to have a meaningful comparison.

[5]*user time* is the total amount of time the system spends executing in user mode

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

the experimental threads are the only user threads running other than the essential system threads. 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 was 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 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 8 illustrate the timing measurement for a typical transaction.

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 versions 1 and 2, and via shared memory 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 make 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.

Light-Weight Threads And Shared Memory

The first set of experiments compares the performance of versions V_1, V_2, and V_{3a}. Table 1 illustrates the percentage reduction in IPC bytes and CPU time between every two versions compared.

As seen in Table 1 and Table 2, the improvement between V_1 and V_2 is indicative of the light-weight thread issues. Unfortunately, only a slight improvement is observed here because the scheduler of our operating system[7] 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, we would expect to do much better. 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 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 [CM89].

	Δ *IPC*	Δ *CPU*
$V_1 \rightarrow V_2$	-3.80%	-5.30%
$V_2 \rightarrow V_{3a}$	-31.25%	-11.11%
$V_1 \rightarrow V_{3a}$	-33.87%	-15.82%

Table 1: Performance Improvement, test Scenario 1: small IPC messages

	Δ *IPC*	Δ *CPU*
$V_1 \rightarrow V_2$	-1.87%	-5.05%
$V_2 \rightarrow V_{3a}$	-40.86%	-15.38%
$V_1 \rightarrow V_{3a}$	-41.97%	-17.11%

Table 2: Performance Improvement, test Scenario 2: large IPC messages

The tables also show that the performance improvement between V_2 and V_{3a}, however, is much more significant due to shared memory. 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 over head is IPC related, we have effectively slashed the IPC overhead by approximately 25%.

In addition, second order effects exist which indirectly improve system performance. These beneficial factors occurred naturally as part of our re-architecture. For example, with the data sharing in version V_{3a}, the Editor no longer has to keep its local copy of the data, saving 400K of memory at run time. 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.

[7]We used a University of Utah's port of MACH 2.0 on HP platform because it is readily available. Although MACH 2.5 exists for the HP9000 machines, it was not stable enough. OSF1 was not available at the time of this experiment.

Bytes vs. Messages

After examining V_{3a}, we discovered that we can do much better by not only reducing the number of bytes being sent between threads, but also the number of IPC messages. We examined the result of two slightly different variations of memory sharing techniques: version V_{3a} which reduces the IPC bytes, and version V_{3b} which reduces the number of IPC messages as well.

The original *Matisse* processes sent and received the [OID,Slot,Value] messages via sockets. Version V_{3a} eliminated most of the byte transfer by allowing the Workshop to send only the small [OID,Slot] messages, while making the Editor perform the lookup and copy of large *Value* data fields directly from the Workshop's memory.

Version V_{3b}, reduced the number of messages to only one per object. The Editor thus had a greater responsibility to look up the Slot and Value attributes of the desired object. The additional computation needed to determine what slots of the given OID needed updating was still much cheaper than sending the list of [OID,Slot] via IPC messages. 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 gained 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 was not very beneficial to just reduce the size of data packets since the payoff would stop after the 8K-bytes packet size.

	Δ *IPC bytes*	Δ *IPC msgs*	Δ *CPU*
$V_1 \to V_2$	-3.80%	0%	-5.30%
$V_2 \to V_{3a}$	-31.25%	0%	-11.11%
$V_1 \to V_{3a}$	-33.87%	0%	-15.82%
$V_{3a} \to V_{3b}$	-80.88%	-80.50%	-28.50%
$V_1 \to V_{3b}$	-87.35%	-80.50%	-39.81%

Table 3: Performance Profile, test Scenario 1: small IPC messages

	Δ *IPC bytes*	Δ *IPC msgs*	Δ *CPU*
$V_1 \to V_2$	-1.87%	0%	-5.05%
$V_2 \to V_{3a}$	-40.86%	0%	-15.38%
$V_1 \to V_{3a}$	-41.97%	0%	-17.11%
$V_{3a} \to V_{3b}$	-85.23%	-82.95%	-20.44%
$V_1 \to V_{3b}$	-91.42%	-82.95%	-34.05%

Table 4: Performance Profile, test Scenario 2: large IPC messages

Additional performance is gained by reducing also the number of messages. This explains the observation that, 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%. And 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 V_{3b}, we have reduced the traffic volume by as much as 85% and CPU time by 28% over version V_{3a}. Comparing this performance with the original version V_1, 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, which was 40%. Obviously, there is still a trickle of IPC messages present in the system since we have not completely eliminated it yet[8] , but the performance improvement resulting from any secondary effects have already covered this cost. In addition, the performance gained from these secondary effects also covered the cost incurred from the shared-memory read/write protocol.

System Threads vs. User Threads

The concept of merging single-threaded 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 multi-threaded support, but rather the reduction of processes to *user threads* of a coroutine package. 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. An important difference between user and kernel threads is that in a multi-threaded 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 one thread at a time, on only one processor at a time.

Multiprocessor Implications

A logical next step would be to rehost *Matisse* onto a multi-threaded, multiprocessor environment. The migration to this type of environment would require no additional work beyond that described. In a multiprocessor 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

[8]For this experiment in process migration, we decided to only implement the shared memory IPC to replace socket-based data transfer between the two threads. Some socket-based IPC messages are still used for control and communication between the threads. These messages can also be replaced by signals and condition variables between threads, with control data transfer facilitated by common data buffers with locks.

different CPUs. Given that the operating system for the multiprocessor machine is designed and implemented properly, we would expect to see a multithreaded application run faster on a multiprocessor machine than on a uniprocessor 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 *Matisse*, 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 then simply doing them serially.

Porting Effort

The port and re-architecture of this experiment took about two man-months complete, since we had no previous experience in porting a system this large and complex. The effort was eased considerably by the help and insight of people in the labs who had ported the MACH operating system onto our machines. Along the way, we carefully documented our path, as reported above, so that future porting efforts can be done much faster. Knowing what we know now, this experiment can be repeated in a one or two-weeks time frame. The approximate time we spent on each part of this experiment is listed in Table 5.

Porting Task	*Time Taken*
UNIX→MACH port	2 weeks
Single-threaded → Multi-threaded	1 week
I/O contention problems	1 week
Unexec dump problems	2 weeks
Shared-memory IPC implementation	2 weeks

Table 5: Experiment time table

The Importance Of Threads

Earlier sections of this paper on the design and porting effort of Matisse describe at length the threads mechanism. The performance charts, however, show that most of the performance gained is from shared-memory, not threads. The importance of threads cannot be under-estimated because it is the mechanism which delivers shared-memory. Without threads, there is no way to obtain the performance improvement the way we had with shared-memory and still keep the existing program interface on IPC. Changing this interface to use system V shared memory would require months of rewrite for a software system this complex. Although the system threads themselves do not contribute much toward the performance gain in this particular system, they are getting faster and lighter, as the importance of threads increase now and in the future, with the prevalent needs for multiprocessing and fine-grained parallelism.

Conclusion

The result of the experiment matched our initial expectations and we have formulated a rough guideline for the migration process. By laying out the steps and identifying the potential problems associated with each step, we hope future migrations can be done quickly and painlessly. However, not all components of a multi-process application should be merged as concurrent threads. Although migrating single-threaded processes to concurrent threads within a multi-threaded task enables the threads to communicate cheaply via shared-memory, the tradeoff is that we lose the machine-independent property of the original processes. By porting and merging processes as concurrent threads, they must now be executed on the same machine, whereas they previously could be run on different machines. Thus, 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.

Acknowledgments

The authors would like to thank professor Jerome H. Saltzer of MIT for the insightful advice, questions, and comments about the project.

Our colleagues Scott Marovich and Debbie Caswell provided enormous help and technical expertise with the UNIX and MACH/OSF1 operating systems. Brian Beach assisted with the memory management issues of the C and LISP environments. Many thanks to Debbie, Kevin Wentzel, and Professor Saltzer for their suggestions on improving the paper.

We thank Dan Fishman, Martin Griss, and Paula Hawthorn for their continuous support of the Matisse project under which the experiment was conducted.

References

[ABB+86] M. J. Acetta, 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.

[Bir89] Andrew D. Birrell. An Introduction to Programming with Threads. Technical report, Digital SRC, January 1989.

[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.

[Cle88] Geoff Clemm. The Workshop System: A practical Knowledge-Based Software

Environment. Proceedings of the 3rd ACM Software Engineering Environments Conference, Pages 55-64, December 1988.

[CM89] D.L. Caswell and S. Marovich. STL MACH Project Retrospective. Technical Report STL-89-20, Hewlett-Packard Laboratories, Software Systems Lab, August 1989.

[FR86] R. Fitzgerald and R. F. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. ACM Transactions on Computer Systems, 4(2):147-149, May 1986.

[JRG+87] 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, August 1987.

[Mar90] Scott B. Marovich. Intraprocess Concurrency under UNIX. Technical Report HPL-91-02, Hewlett-Packard Laboratories, March 1990.

[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.

[Sta84] R. Stallman. EMACS: The Extensible, Customizable, Self-Documenting Display Editor. In D. R. Barstow, H. E. Shrobe, and E. Sandelwall, editor, Interactive Programming Environments, pages 300-325. McGraw-Hill Book Company, 1984.

[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.

Author Information

Thuan Pham is a Member of Technical Staff at Hewlett-Packard Laboratories, Software Technology Lab. He received his M.S. in Electrical Engineering and Computer Science and the B.S. in Computer Science and Engineering from the Massachusetts Institute of Technology. His research interests include software engineering, operating systems, and user interface. Contact him at pham@hplabs.hp.com.

Pankaj Garg is a Project Leader at Hewlett-Packard Laboratories, Software Technology Lab. He got his Ph.D in Computer Science at the University of Southern California, in February 1989. Pankaj was an All-University-Pre-Doctoral-Merit fellow of the graduate school, USC, for the years 1984 through 1987. He was a research associate in the Computer Science Department, USC, from 1987 through February 1989. He got his Bachelor of Technology degree in Computer Science, from Indian Institute of Technology, Kanpur (INDIA). His research interests are in artificial intelligence, hypertext systems, and software engineering. Contact him at garg@hplabs.hp.com.