14th ACM Symposium on Operating System Principles (SOSP)
5th-8th December, 1993, Asheville, North Carolina

Improving IPC by Kernel Design

Jochen Liedtke
German National Research Center for Computer Science (GMD) *
jochen.liedtke@gmd.de

Abstract

Inter-process communication (ipc) has to be fast and effective, otherwise programmers will not use remote procedure calls (RPC), multithreading and multitasking adequately. Thus ipc performance is vital for modern operating systems, especially μ-kernel based ones. Surprisingly, most μ-kernels exhibit poor ipc performance, typically requiring 100 μs for a short message transfer on a modern processor, running with 50 MHz clock rate.

In contrast, we achieve 5 μs; a twentyfold improvement.

This paper describes the methods and principles used, starting from the architectural design and going down to the coding level. There is no single trick to obtaining this high performance; rather, a synergetic approach in design and implementation on all levels is needed. The methods and their synergy are illustrated by applying them to a concrete example, the L3 μ-kernel (an industrial-quality operating system in daily use at several hundred sites). The main ideas are to guide the complete kernel design by the ipc requirements, and to make heavy use of the concept of virtual address space inside the μ-kernel itself.

As the L3 experiment shows, significant performance gains are possible: compared with Mach, they range from a factor of 22 (8-byte messages) to 3 (4-Kbyte messages). Although hardware specific details influence both the design and implementation, these techniques are applicable to the whole class of conventional general purpose von Neumann processors supporting virtual addresses. Furthermore, the effort required is reasonably small, for example the dedicated parts of the μ-kernel can be concentrated in a single medium sized module.

1 The IPC Dilemma

Inter-process communication (ipc) by message passing is one of the central paradigms of most μ-kernel based and other client/server architectures. It helps to increase modularity, flexibility, security and scalability, and it is the key for distributed systems and applications.

To gain acceptance by programmers and users, ipc has to become a very efficient basic mechanism. Surprisingly, most ipc implementations perform poorly. Dependent on the processor speed, transferring a short message by ipc typically takes between 50 and 500 μs. A lot of effort has been invested in improving ipc performance [Che 84, Sch 89, Ber 89, Dra 91], but this has not led to a real breakthrough. So ipc (and the μ-kernel approach) is widely regarded as a nice concept, but its use is hotly debated because of the perceived lack of efficiency. One consequence is that programmers try to circumvent ipc [Ber 92].

We have overcome the ipc dilemma by carefully constructing a μ-kernel that more than achieves our aim of a tenfold improvement in ipc performance over comparable systems.

2 Related Work

Comparable message based kernels are Amoeba [Mul 84], BirliX [Hr 92], Chorus [Gui 82], and Mach [Acc 86]. Ipc improvement was addressed earlier by Cheriton [Che 84a] using registers for short messages.

A strictly RPC [Bir 84] oriented ipc system was implemented in DEC's Firefly workstation [Sch 89], the SRC RPC. It paid special attention to the performance of same-machine RPC, e.g. by using a special path through the scheduler for RPC context switching and

*GMD I5.RS, Schlo Birlinghoven, 53757 Sankt Augustin, Germany
0

message buffers shared across all domains (thus trading safety for performance).

Bershad constructed an even faster method, called LRPC [Ber 89], also implemented on the Firefly. It achieves its good performance (3 times faster than SRC RPC) mainly by using simple stubs, direct context switching and shared message buffers.

But while LRPC is restricted to synchronous RPC-like communication (blocking, without timeouts) and has some constraints concerning security, message structure, message size and number of clients (see 5.2.3), our technique supports Mach-like long and structured messages, has true ipc semantics and performs significantly faster than LRPC (see table 2).

3 L3 – The Workbench

L3 is a μ-kernel based operating system built by GMD [Lie 91, Bey 88, Lie 92a] which has been used for 4 years as a production system in business and education. To date, about 500 systems have been shipped to end users. L3 is now in daily use in a variety of industrial and commercial contexts.

The L3-kernel is an abstract machine implementing the data type *task*. A task consists of *threads*, memory objects called *dataspaces* and an *address space* into which dataspaces can be mapped. As in Mach, paging is done by default or external pager tasks. All interactions between tasks, and with the outside world, are based on inter-process communication (ipc).

The L3 ipc model is quite straightforward. Active components, i.e. threads, communicate via messages which consist of strings and/or memory objects. Each message is sent directly from the sending to the receiving thread. There are neither communication channels nor links, only global thread and task identifiers (uids) which are unique in time. A server usually concludes from the uid of the message sender whether the requested action is permitted for this client or not. The integrity of messages, in conjunction with the autonomy of tasks, is the basis for higher level protection.

The ipc mechanism is heavily used inside L3, since all logical and physical device drivers are implemented as user level tasks communicating exclusively via ipc. Hardware interrupts are integrated into this concept by transforming them into interrupt messages which are delivered by the μ-kernel to the appropriate thread.

Further important aspects of L3 are the *persistence of data and threads* [Lie 93] and its *Clans & Chiefs* model [Lie 92] permitting message redirection. Due to limited space the additional implications of these concepts for efficient ipc implementation are not discussed here.

4 Principles And Methods

Since all efforts to improve ipc by later optimizations seemed to give only poor results, we decided to approach the problem from the opposite direction. We seized the opportunity of the L3 μ-kernel's redesign to aim for really high performance by reconstructing the process control and communication related sections from scratch.

Paradigms and the basic architecture (tasks, threads, ipc, address spaces, pager) were predefined by older L3 versions, but we were free to change implementation methods and binary interfaces. The principles we adopted for this internal redesign were:

- *Ipc performance is the Master.*
  Anything which may lead to higher ipc performance has to be discussed. In case of doubt, decisions in favour of ipc have to be taken. But the performance and security qualities of other components must not be seriously impacted.

- *All design decisions require a performance discussion.*
  This will initially be based on more or less realistic models, but must be validated later.

- *If something performs poorly, look for new techniques.*

- *Synergetic effects have to be taken into consideration.*
  Combining methods may lead to reinforcement as well as to diminution. Especially, a new method may require further new methods for proper efficiency.

- *The design has to cover all levels from architecture down to coding.*

- *The design has to be made on a concrete basis.*
  It is important to discuss the different levels of dependency during the design process. Some results will totally depend on specific hardware, others will be of more general nature, adaptable for the next concrete design.

- *The design has to aim at a concrete performance goal.*
  This is essential for determining weaknesses and poor techniques.

5 A Concrete Design

This section describes the concrete construction and discusses the internal design decisions taken when implementing version 3 of L3's μ-kernel on 386/486 based hardware. It demonstrates the practical application of the principles given above.

For illustration, an Intel 486-DX50 processor is used. Running with a 50 MHz clock rate it contains an on-chip cache of 8 Kbyte, and typical instructions take 1 or 2 cycles, assuming cache hits [i486]. The memory management unit (MMU) translates 32-bit virtual addresses using 4 Kbyte pages. Page access rights are read or read/write and kernel or user/kernel. Address translation is supported by a 4-way set-associative translation lookaside buffer (TLB) holding 32 entries. This TLB is flushed on switching address space.

5.1 Performance Objective

When considering different models and algorithms it is essential to have an idea about the achievable performance. Assume a simple scenario: thread A sends a null message to thread B which is ready to receive it. Both threads run at user level and reside in different address spaces. Since primary interest is in the lower bound for the time needed, we begin by looking at the minimal sequence of essential basic actions:

thread A (user mode):	load id of B
	set msg length to 0
	call kernel
kernel:	access thread B
	switch stack pointer
	switch address space
	load id of A
	return to user
thread B (user mode):	inspect received msg

All operations concerning parameter passing and testing, all scheduling actions and everything necessary to transfer non null messages is omitted. The scenario is, therefore, not very realistic, but allows a rough best case estimation. We assembled the minimal necessary 20 (!) instructions for the basic actions mentioned above (see table 3). Assuming the complete absence of write or code prefetch delays and cache misses, the execution cycles sum up to 127. (107 of these cycles are consumed by the two instructions for entering and leaving kernel mode which are extremely expensive.) Since the 486 MMU is flushed when changing the address space, at least 5 TLB misses will also occur consuming 9 cycles each. Together this results in a minimum of 172 cycles (3.5 μs) as a lower bound for ipc.

Thus, when starting the experiment, we decided to aim at a performance of about 350 cycles (7 μs) per short message transfer. In fact, we achieved 250 cycles (5 μs). Later, this value of 5 μs, denoted as T, will be used as a scale when discussing optimizations.

5.2 Architectural Level

5.2.1 System Calls

System calls are expensive: simply entering and leaving kernel mode costs 2 μs (40% T). Therefore ipc has to require as few system calls as possible. Since most client/server systems operate synchronously, introducing the system calls call and reply & receive next (besides to the non-blocking send and receive call) permits an implementation using only 2 system calls instead of 4 per RPC, i.e. one per ipc. Other systems contain similar optimizations, e.g. SRC RPC [Sch 89].

Since both call and reply & receive combine sending the outgoing message with waiting for an incoming message into a single primitive, client/server protocols become simpler (the server can be sure that the client is ready to receive the reply) and there is no need for scheduling to handle replies differently from requests.

5.2.2 Messages

The costs of system calls and address space switches (together about 3 μs, 60% T) suggest the need to support complex messages in such a way that a sequence of send operations can be combined into a single one, if no intermediate reply is required. Besides higher efficiency, this also results in simpler communication protocols.

In L3, one message may contain a direct string (mandatory), indirect strings (optional), and memory objects (optional) (see figure 1).

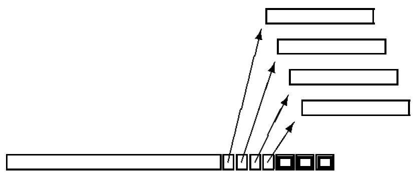

Figure 1: *A Complex Message*

Direct and indirect strings are copied strictly, memory objects lazily (similar to a Mach outline message). Since the transfer of memory objects includes mapping, unmapping and usually also pager activity, its efficiency is not discussed in this paper; but it is comparable with the equivalent actions in Mach.

Indirect strings help to avoid copy operations at user level. When, for example, text has to be sent to a screen driver, the message obviously has to contain the operation code, perhaps coordinates, and the text string itself, which may be placed anywhere else by the compiler. Then the message will contain operation code and coordinates in the direct string and the text as an indirect value specified by address and length.

To simplify user level programming receive buffers are structured in the same way: one part receives the direct string, further ones (specified by address and length parameters in the receive operation) the indirect strings, and the last ones the memory object identifiers. In this way complex messages may be used to transfer values directly from the sender's program variables to the receiver's program variables (figure 2).

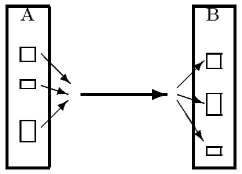

Figure 2: *Sending a Complex Message*

This is similar to the multi-part messages of QNX [Hil 92] but differs from Mach's outline message transfer: in Mach, a receiver cannot specify the buffers for outline messages. Furthermore, Mach's outline transfer is primarily designed for larger messages and is very expensive when used for smal or unaligned data.

5.2.3 Direct Transfer by Temporary Mapping

One basic problem of inter-process communication is the transfer of messages between address spaces. Most μ-kernels solve this problem by a twofold copy: user A space $\rightarrow$ kernel space $\rightarrow$ user B space. The message is copied twice, because the address space model consists of a user accessible part and a fixed kernel accessible part which is shared by all address spaces (see figure 3). Since the user parts are distinct, messages must be transferred via the kernel part.

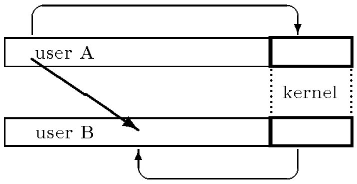

Figure 3: *Twofold Message Copy*

This is of no extra cost if the message has to be buffered by the kernel. But in a modern multithreaded client/server system most RPC's operate synchronously: the client blocks until it gets the reply or a timeout; so message buffering is superfluous.

Copying an n-byte message costs about $20 + 0.75n$ cycles plus additional TLB and cache misses. Even transferring an 8-byte message by the above algorithm would be about 0.5 μs (10% T) more expensive than a single copy method. With larger messages, increasing cache misses and flooding will lead to even higher costs.

Therefore, at least non-short messages should be transferred directly from source to destination address space. LRPC and SRC RPC share user level memory of client and server to transfer messages. Then only one copy (sender $\rightarrow$ shared buffer) is required. But this seriously affects security and has other disadvantages:

- Multi-level security [DoD 83] can be penetrated by using shared communication buffers as hidden channels without ipc system control.
- The receiver cannot check a message's legality, since it may be changed by the sender during or after checking. (A second copy into the receiver's private memory area would solve the problem but eat up the one-copy benefit.)
- When servers communicate with many clients and messages are sometimes very long, shared regions of the virtual address space may become a critical resource.
- Shared regions require explicit opening of communication channels. (L3 permits communication without prior opening.)
- Shared buffers are not application friendly, since they do not allow direct transfer from variable to variable as described in 5.2.2.

L3 therefore uses a new method based on temporary mapping: the kernel determines the target region of the destination address space, maps it temporarily into a *communication window* in the source address space, and then copies the message directly from the sender's user space into its communication window. Due to aliasing the message appears at the right place in the receiver's address space.

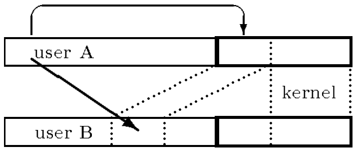

Figure 4: *Direct Message Copy*

The communication window is only kernel accessible, but unlike other kernel areas it exists per address space, i.e. is not shared between all spaces.

Direct transfer avoids the disadvantages of user level shared communication buffers, since temporary mapping is done on demand and buffers are shared only at kernel level. (When A sends a message to B, kernel A and user B can access the receiver buffer, but user A cannot.)

Problems implementing communication windows are:

- Temporary mapping must be fast.

- Different threads must coexist in the same address space.

The 486 MMU uses a two level page table for translating virtual addresses. The first level table, also called the page directory, holds 1024 entries, each of them pointing to a 1024-entry second level table. Thus each directory entry corresponds to 4 MB of the virtual address space. As shown in figure 5, temporary mapping of such a region requires only one word to be copied from page directory B to page directory A.

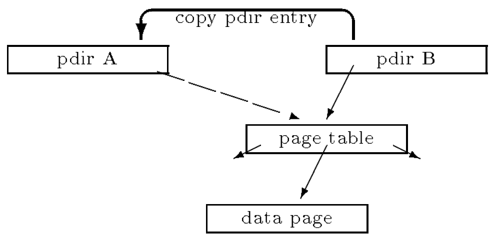

Figure 5: *Fast Temporary Mapping*

To avoid region crossing problems we restrict message string size to 4 MB (there may be up to 255 strings in a message) and always map an 8 MB region with the receive buffer beginning in its lower half.

For proper operation the TLB must not hold any page table entries relating to use of the communication window by earlier or concurrent operations. If this is true, we call the TLB *window clean*; a flushed TLB is obviously window clean.

A window clean TLB could be ensured by flushing the complete TLB before each temporary mapping, what is quite an expensive operation. (The 486 processor allows the complete TLB to be flushed or just single pages, but there is no efficient mechanism to flush intermediate-sized regions of the address space.) To find a more efficient method we first look at a one thread per address space scenario:

1. At the very beginning the TLB is window clean.

2. Immediately after switching back to this thread, the TLB is window clean. (It is the only thread in the address space, and address space switch flushes the TLB.)

3. Therefore, the first send operation starts with a window clean TLB. It remains window clean during transfer.

4. When an address space switch is part of the ipc operation and takes place after copying the message, the TLB is window clean after ipc. Thus the TLB is always window clean.

Handling multiple threads per address space is more complicated, since concurrent ipc would violate the window clean constraint. Allocating as many communication windows as threads in an address space would solve the problem, but would result in an intolerable restriction on the number of threads per address space.

Instead, only one window is used and the problem of multiple threads accessing it concurrently is solved by:

1. Enforcing an additional TLB flush when thread switching does *not* change the address space *and* page directory entries related to the communication window are actually in use. (This happens only when an inter-address space transfer is interrupted by timeslice exhaustion, a page fault or a hardware interrupt. Ipcs that happen to be intra-address space are handled without using the communication window.)

2. Invalidating the communication window entries in the page directory upon thread switch. (When control is returned to this thread, accessing the communication window leads to a page fault so that the page fault handler can reestablish the temporary mapping.)

Multiprocessor: One window per processor in each address space should be used. Additional lock/unlock operations on page directories are not needed.

Different Processor: The method shown is nice, because in most cases it does not require additional actions and the necessary TLB flush is free of charge. If, however, we assume the TLB to be capable of holding page table entries of different address spaces in parallel, TLB flush is not normally required upon changing address space and the method would become expensive. To avoid this any TLB which supports multiple spaces should offer efficient flush operations on larger address space regions.

5.2.4 Strict Process Orientation

For simplicity and efficiency, threads that are temporarily running in kernel mode should be handled in the same way as when running in user mode. Thus it is natural to allocate one kernel stack per thread. This appears to be the most efficient way, since interrupts (including clock interrupts), page faults, ipc and other system calls already save state information (instruction counter, flags, user stack pointer) on the actual kernel stack.

Continuations [Dra 91] or similar techniques could reduce the number of kernel stacks, but require either additional copy operations between kernel stack and continuation, or stack switching. Both methods induce additional TLB misses and therefore are more expensive (stack switching: 0.5 μs, 10% T; copying: 1.7 μs, 33% T). Furthermore, continuations cannot be used in all cases and require special programming support. In practice, continuations interfere with other optimizations at lower levels, e.g. thread control block addressing. Thus the real costs will be even higher.

One kernel stack per thread leads to a large number of stacks; but this is only a minor problem if these stacks are objects in virtual memory (see figure 6) and combined with their corresponding control blocks.

5.2.5 Control Blocks as Virtual Objects

Thread control blocks (tcb's) are used to hold kernel- and hardware-relevant thread-specific data. This includes registers, state information and a kernel stack. An efficient way to manage tcbs is to hold them all in a large virtual array (see figure 6) located in the shared part of all address spaces. Of course, this structure must only be accessed by the kernel. Benefits of this method are:

- It permits fast tcb access. Only the address array base + tcb no $\times$ tcb size has to be calculated. Furthermore, explicitly checking whether the addressed tcb is allocated and swapped in becomes superfluous. This job can be shifted to the page fault handler which deals with it only when an unmapped tcb is accessed.
- It saves 3 TLB misses per ipc (0.5 μs, 10% T): one by directly accessing the destination tcb (without using a table) and a further two, since the kernel stacks of the sender and receiver are located in the corresponding tcb pages.
- Locking a thread can be done simply by unmapping its tcb.
- It helps to make threads persistent.
- Ipc can be implemented orthogonally to / independently from memory management. Page faults or tcb swap outs during message transfers are invisible to the ipc system.

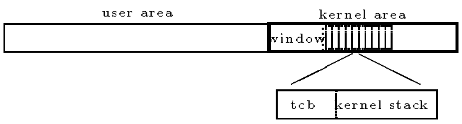

Figure 6: *Address Space With Thread Control Blocks*

Allocating a thread's kernel stack within its tcb permits an even faster access method to the matching tcb, when tcbs are 2^n-aligned: simply and the stack pointer with a bit mask.

Similar mechanisms can be used for task control blocks and page directories. Recall that these first level page tables have to be accessed for temporary mapping (see figure 5). It is useful to have a virtual array of all page directories inside each address space as well as a fixed location for this address space's own directory. Temporary mapping then reduces to

my pdir [window] := pdir [dest] [buffer $\gg$ 22];
my pdir [window+1] := pdir [dest] [(buffer $\gg$ 22)+1].

Multiprocessor: The destination tcb must be locked for ipc in a multiprocessor system, but this is not required for the sending tcb, unless the sender has to insert itself into a queue. Due to lazy scheduling (see 5.3.4) this generally only happens when the destination is not ready to receive. In a multiprocessor system writing the tcb's state field will be more expensive, since it requires a locked write (through cache into external memory).

5.3 Algorithmic Level

5.3.1 Thread Identifier

As mentioned in the previous section, a tcb address can easily be calculated from the thread number. In user mode, however, a thread is always addressed by its unique identifier (uid). L3 uses 64-bit wide thread uids containing thread number, generation (for time uniqueness), station number and chief id. (Clans & Chiefs is a concept unique to L3 described elsewhere [Lie 92].)

To support the calculation of the tcb address from a given uid, the uid contains the thread number in its lower 32 bits in such a way that only anding it with a bit mask and adding the tcb array's base address is necessary. Index checking can be omitted, if the valid tcb numbers range from 0 to 2^m, and only 3 cycles (0.06 μs, 1.2% T) are required.

If the given uid specifies a thread on a different station, or has become invalid in the meantime, the above algorithm accesses the wrong tcb. Each tcb therefore contains its thread uid which is checked against the requested uid.[1] In the usual case (no page fault, uid matching) this costs another 4 cycles, i.e. 0.08 μs, 1.6% T.

[1] Inter-node ipc requires a further discussion. In L3 this is automatically covered by the Clan concept. Since a thread residing in a different node must belong to a different clan, ipc is automatically redirected to the sender's chief which resides on its own node. This is done before tcb access; so inter node communication will not lead to 'wrong' tcb accesses. In other systems the μ-kernel could either check for inter-node communication before tcb access or manage it after discovering the uid difference. The latter case may be faster, but leads to superfluous tcb swap ins.

If the μ-kernel supports thread migration (L3 does not), duplicate numbers for threads residing in the same node can be circumvented by means of proxies remaining on the threads original node.

5.3.2 Handling Virtual Queues

The kernel handles a variety of thread queues, e.g. busy queue, present queue and a *polling-me* queue per thread, which contains all threads actually trying to send *me* a message.

The most efficient implementation uses doubly linked lists, where the links are held in the tcbs. However, parsing the ready queue, or getting the next sender out of the actual thread's polling-me queue, must not lead to page faults. Unmapping a tcb therefore includes removal from all these queues. So tcbs are chained in virtual address space, but parsing the chains and inserting or deleting tcbs will never lead to page faults.

5.3.3 Timeouts And Wakeups

Timeouts can be specified in each ipc operation. A timeout value t means the operation fails (and the thread is awakened) if message transfer has not started t ms after invoking the operation. The frequently used values $t = \infty$ and $t = 0$ can be implemented very easily, but real timeouts require something like a wakeup queue.

Since far more ipc operations succeed than fail due to timeout, insertion into and deletion from the wakeup queue must be very fast. The fastest method is a large array, indexed by thread number, holding the wakeup time for each thread; but sequentially parsing the array on each clock interrupt is far too expensive: For 16K entries, 2 ms would be needed; the on-chip cache would also be flooded. We finally decided to use a set of n unordered wakeup lists implemented by doubly linked lists. If a thread is entered with wakeup time τ, its tcb is linked into the list $\tau \bmod n$. If a total of k threads are contained in the wakeup lists, the scheduler will have to inspect k/n entries per clock interrupt, on average. Furthermore, the scheduler removes a thread if its wakeup point is far in the future and it has already been waiting for some seconds. Such threads are held in a long time wakeup list until their wakeup time approaches; they are then reinserted into the normal wakeup lists.

Using a set of unordered wakeup lists combines fast insert/delete with low bookkeeping costs for a moderate number of active threads. Since we use $n = 8$ lists and a wakeup granularity of 4 ms, 400 active threads will lead to at most $400/8 \times 250 = 12500$ inspected wakeup entries per second, the processor then spends less than 1% of its cycles on scheduling. Note that this scenario requires at least 12,500 ipc's per second, i.e. at least 6% of the cpu time is used for pure ipc. Since in practice three quarters or more of the ipc operations use timeouts 0 or ∞ not requiring queue operations, the scenario would probably require even 50,000 ipc operations per second, 25% of the processor time. So wakeup handling usually costs less than 4% of the total ipc time.

Another problem related to wakeups is the representation of time. Using 1 ms as the time unit, a 32 bit value can denote intervals up to 48 days. Since a system may run for far longer, this is sufficient for timeout intervals but not for wakeup times. (Simply powering down a system does not clear the wakeup queue. Since in L3 everything is persistent, wakeups survive power off intervals.) 64 bit values are, however, too expensive on a 32 bit processor, particularly because they occupy more registers.

We use (base+offset) to represent a point in time, e.g. a wakeup. The base is controlled by the kernel in such a way that the actual time is always represented by an offset less than 2^{24}. Furthermore, timeouts are restricted to a maximum of 2^{31} ms ($\approx$ 24 days). Then wakeups can always be calculated and managed by 32 bit arithmetic using only one register or memory word. Whenever the actual time offset reaches 2^{24}ms ($\approx$ 4.5 hours), the kernel increases the base and updates all offsets in the wakeup lists.

5.3.4 Lazy Scheduling

Conventionally, an ipc operation call or reply & receive next requires some scheduling actions:

1. deleting the sending thread from the ready queue,
2. inserting it into the waiting queue,
3. deleting the receiving thread from the waiting queue,
4. inserting it into the ready queue.

Insertion needs at least 7, and deletion 4, load/store operations. Together with 4 expected TLB misses this will take at least 58 cycles (1.2 μs, 23% T).

Multiprocessor: Due to the need for locking, these operations would be even more expensive in a multiprocessor system.

Instead, a method we call *lazy scheduling* is used. Ipc tries to avoid queue manipulation and changes only the thread state variable in the tcb from ready to waiting or vice versa. The invariants for ready and wakeup queues are:

> The ready queue contains *at least* all ready threads, except possibly the current one.
>
> Each wakeup queue contains *at least* all threads waiting in this class.

There may be threads in the ready queue which are now waiting or polling, and the thread actually controlling a processor may not be in the queue even though it is ready. Furthermore, threads may be in many queues. Whether a thread really belongs in a queue must be deduced from the tcb's state field. Whenever a queue is parsed the scheduler removes all threads that no longer belong in it. Hence the delete operations mentioned above can always be omitted, and insert operations become unnecessary when the thread was enqueued earlier and has not yet been removed. In theory, this interferes with the idea of using n wakeup lists. Since a thread may now be contained in *all* lists, in the worst case k entries, rather than k/n, have to be inspected. In practice this seldom happens.

Furthermore, insertion into the ready queue can always be omitted at call and reply & receive next operations. All ipc operations except send block the invoker. So there are only three situations in which a thread looses processor control and still remains ready: end of timeslice, hardware interrupt and send. To guarantee the ready queue invariant in these cases the current thread is inserted into the ready queue if necessary.

In the worst case (few ipcs per interval) lazy scheduling is comparable with strict scheduling; happily it performs better and better with increasing ipc rate. In an L3 system with one active user, between 500 and 2,000 ipcs typically occur per second, with peak values (e.g. when browsing through a file with the editor) exceeding 10,000. The ratio 'ipcs : lazy queue update operations' typically ranges between 2:1 and 5:1, though very high ipc rates can lead to a 50:1 ratio.

5.3.5 Direct Process Switch

For a remote procedure call it is natural to switch the flow of control directly to the called thread, donating the current timeslice to it (as also LRPC does).This is also the most efficient method, since it only involves changing stack pointer and address space. The same method is used for replies and non-blocking send operations. However, in L3, when B sends a reply to A and another thread C is waiting to send a message to B (polling B), C's ipc to B is immediately initiated before continuing A.

Fairness will not be discussed in detail here, but some basic properties are stated:

- Very loosely, ipc overhead is the only difference in timing between a normal procedure call and a remote procedure call. Two threads playing ping-pong will not be punished by the scheduler. This results mainly from timeslice donation and the ready queue's stability due to lazy scheduling.
- When multiple threads try to send messages to one receiver, it will get the messages in the sequence in which the ipc operations were invoked, i.e. no sender may dominate a receiver. Recall that messages are not buffered by the kernel, only the polling threads are queued.

5.3.6 Short Messages Via Registers

Usually, a high proportion of messages are very short, since frequently used RPCs have few input or output parameters. For example ack/error replies from drivers are very short, as are hardware initiated interrupt messages. In an L3 system, on average, between 50% and 80% of messages contain 8 bytes (plus 8 bytes sender id) or less. The direct transfer of short messages via cpu registers (similar to Cheriton's experiment [Che 84a] or as proposed by Karger [Kar 89]) may therefore be worthwhile. The 486 processor has 7 general registers, but three are needed for the sender id and result code. So only four are available. Whether two of them can efficiently be used for transferring 8-byte-messages required a coding experiment.

To decide whether this uncertain gain is worth additional coding efforts – recall that direct message transfer has to be implemented anyway – we made an optimistic best case estimation of the costs of temporary mapping (see table 4) which suggested an overhead of at least 62 cycles (1.2 μs, 25% T).

We succeeded in coding the transfer of 8-byte messages via registers, but this seems to be the upper limit on this processor. In fact, we achieved a performance gain of 2.4 μs or 48% T, since special treatment of such short messages permitted additional coding optimizations.

Different Processor: Register transfer pays for the 486 processor, but perhaps not for other processors with fewer registers or an MMU which gives fewer TLB faults. Even in such a case, special treatment of short messages may pay, since it permits some optimizations on the coding level.

5.4 Interface Level

Ipc performance is not only determined by the kernel algorithms, but also by the user/kernel interface. It is important to support typical usage and permit compilers to optimize code. RPC stubs should especially be as simple as possible. Ideally, they should only load some registers, issue a system call, and check its success. This is short enough to permit the compiler to generate stubcode in-line.

5.4.1 Avoiding Unnecessary Copies

As described in section 5.2.2, messages may be compound values composed of direct strings, indirect strings and memory objects to reduce the number of ipc calls and avoid unnecessary copying.

However, objects cannot be arbitrarily mixed in a message, but must be grouped by their types. This permits more efficient kernel algorithms, and simplifies message parsing at both kernel and user level.

Message manipulation, tracing and forwarding become easier and faster when data in send and receive buffers are structured in the same way. In this case unnecessary copies can be avoided by using the same variable for receiving and sending. But, of course, using different variables in call or reply & receive next is also possible. This avoids copying when using "message constants" for orders or replies. Messages are described by dope vectors containing the actual length as well as the maximum size of the message objects.

5.4.2 Parameter Passing

The kernel's ipc interface should use registers for parameter passing whenever possible, since

1. registers can be accessed far more efficiently than user stack,
2. input/output parameters in registers give compilers better opportunities for code optimization.

Furthermore, registers which are not used for parameter passing should always be defined as scratch. The kernel would, otherwise, always have to save and restore them, whereas a code generator knows whether and how to reconstruct their old values most efficiently. The 486 register usage in L3 is shown in table 6.

5.5 Coding Level

5.5.1 Reducing Cache Misses

The most frequently used kernel code should be as short as possible. So it should use short jumps, registers instead of memory and short address displacements. (486 instructions address memory by means of base/index register and one or four byte displacements.) Consequently, frequently accessed tcb data should be reached by one byte displacements.

To reduce data cache misses, the tcb and other tables have to be organized so that frequently used data are concentrated in few cache lines, and so that data which are often used together are placed in the same cache line. To reduce delays on cache misses, the cache line fill sequence should match the usual data access sequence.

5.5.2 Minimizing TLB Misses

Since TLB misses are expensive and the TLB is flushed on address space switch, the ipc related kernel code should be placed in one page. Equally, all processor internal tables[2] that are accessed when switching thread and address space should be placed in one page, if possible together with the frequently used kernel data, e.g. the system clock. Larger tables should be placed so that they end or start in this page and the most heavily used entries are located in this page. We saved 4 TLB misses by this technique, i.e. 0.7 μs or 14% T.

Note that together with handling control blocks and kernel stacks as virtual objects (see 5.2.5) and lazy scheduling (see 5.3.4) at least 11 TLB misses are saved, 2.0 μs or 40% T.

5.5.3 Segment Registers

Segment register loading is expensive: 9 cycles. So the preferred memory model uses only one flat segment covering the complete address space. Once the segment registers are initialized with the descriptor of this flat segment, new loads should be superfluous. Unfortunately, there is user level software which relies on different models. If the kernel guaranteed to maintain the user segment registers and load the necessary *flat* descriptor, this would cost 66 cycles, 1.3 μs or 26% T per ipc.

Instead, the kernel checks on entry whether the segment registers contain the flat descriptor, and guarantees that they contain it when returning to user level. So segment register loads are avoided at both kernel and user level when the flat memory model is used. In this case only 10 cycles are required for checking the segment registers (0.2 μs, 4 % T).

5.5.4 General Registers

Besides applying standard register usage optimization, one has to pay attention to the user/kernel interface. Its register conventions influence coding possibilities in the kernel.

A special 486 feature is the aliasing of four 32-bit registers with pairs of 8-bit registers. We used this by restricting direct strings to 255×4 bytes, and allowing only 255 indirect strings (of up to 4 MB each) and 255 memory objects. As a consequence a complete message dope fits into one 32-bit register and needs only one memory access, whereas each counter can be directly accessed through the corresponding 8-bit register.

[2] Due to the 486's segment support (inherited from 432, 286 and 386) there are three descriptor tables (GDT, IDT and TSS) which are of no use in this context but are accessed by hardware.

5.5.5 Avoiding Jumps and Checks

Conditional jumps taken are more expensive than those which are not taken (3:1 cycles), induce pipeline delays, and potentially lead to worse cache utilization. Therefore basic code blocks should be arranged so that the "main stream" executes as few jump instructions as possible.

Since illegal ipcs can be executed more slowly, essential parameter checks should be shifted to seldom executed alternatives. For example sending a message to the sender itself is illegal, but it is sufficient to check this only if the destination is not ready to receive.

5.5.6 Process Switch

In most cases a process switch only requires the stack pointer, and perhaps the address space, to change. Additional actions become necessary when the source thread has used the processor's debug registers[3] or the numeric coprocessor. Since use of both resources can be monitored by the kernel, the corresponding save/restore actions are only invoked when really necessary. In the case of the numeric coprocessor, save/restore is handled lazily, i.e. is delayed until a different thread tries to access the coprocessor registers.

Sparc-like Processor: On Sparc[4] processors with large number of registers, we propose to experiment with lazy schemes for saving and restoring register windows [Lie 93a], similar to the coprocessor handling mentioned above.

5.6 Summary of Techniques

add new system calls	(5.2.1)
rich message structure,	
symmetry of send & receive buffers	(5.2.2)
single copy through temporary mapping	(5.2.3)
kernel stack per thread	(5.2.4)
control blocks held in virtual memory	(5.2.5)
thread uid structure	(5.3.1)
unlink tcbs from queues when unmapping	(5.3.2)
optimized timeout bookkeeping	(5.3.3)
lazy scheduling	(5.3.4)
direct process switch	(5.3.5)
pass short messages in register	(5.3.6)
reduce cache misses	(5.5.1)
reduce TLB misses (careful placement)	(5.5.2)
optimize use of segment registers	(5.5.3)
make best use of general registers	(5.5.4)
avoid jumps and checks	(5.5.5)
minimize process switch activities	(5.5.6)

[3] 486 debug registers allow the use of data and code breakpoints.

[4] SPARC is a trademark of Sun Microsystems.

The effects of some optimizations can be easily quantified. Table 1 shows these techniques and the additional time which would be needed without them. For better comparison the time is given relative to the ipc times obtained. 100% means that removing this (and only this) optimization would double the ipc time. For short messages, the most valuable technique is register transfer, whereas direct message transfer dominates all other optimizations when messages become longer.

removed optimization	time increase (n-byte ipc) 8	12	128	512	4096
short msg via reg	49%	–	–	–	–
direct transfer	–	9%	23%	58%	157%
lazy scheduling	23%	16%	12%	7%	1%
no segm reg	21%	14%	11%	6%	1%
reply & wait[a]	18%	13%	10%	5%	1%
condensed tables[b]	13%	9%	7%	4%	1%
virtual tcb[c]	10%	7%	5%	3%	1%

[a] when used to implement RPC

[b] due to less TLB misses

[c] only reduced TLB miss effect

Table 1: *Easily quantifiable Effects*

Note that this table completely ignores synergetic effects. Removal of all optimizations shown in the table will increase the 8-byte ipc time by far more than 134% ($= 49 + 23 + 21 + 18 + 13 + 10$).

Unfortunately, the effects of the other techniques described here cannot be quantified as easily. Some techniques have no unique alternative; in other cases the effects cannot be isolated. For example an alternative address space structure, where control blocks are no longer virtual objects, could lead to strong dependencies between the ipc system and memory management. If an ipc system had to avoid paging and to lock each memory page before accessing it, it would differ radically from ours.

The relevance of such decisions influencing internal system structure and interface can be demonstrated by a simple example. For various hardware platforms and operating systems, Ousterhout [Ous 90] measured the costs of entering and leaving the OS kernel by executing the trivial `getpid` call. All the times were at least 10 times (sometimes even 20 or 30 times) larger than the bare machine time for calling the kernel from user level and returning. For Mach on a 486 (50 MHz) we measured 18 μs for the `mach_thread_self` call, whereas the bare machine time for user→kernel→user is 2 μs. Using the Mach system call implementation would increase our 8-byte ipc time by 300%.

6 Results

For measurement a 'noname' PC was used, containing an Intel 486 DX-50 (running at 50 MHz), 256 KB external cache and 16 MB memory. Two user level threads running in different address spaces repeatedly played pingpong with a message of n bytes net size (+ sender/receiver id). The ping client uses call, whereas the pong server operates by reply & receive next. Pingpong (which in fact is a true synchronous RPC) is executed 10,000 times, and the average ipc time is calculated by dividing the totally elapsed time by 20,000:

> A short cross address space ipc (user $\rightarrow$ user) takes 5.2 μs.

To measure the cache usage the cache was first flushed, then one pingpong was executed and afterwards the valid cache lines were counted.

> All code and data together use 592 bytes (7%) of the on-chip-cache.

So the cache penalty should be bearable in real use, *and* user programs which communicate through short messages will not be punished by the kernel flooding the cache. The worst case cache penalty for short ipc is 6.4 μs. It was measured by flushing the cache before each ipc.

> Longer ipcs, transferring n bytes, take about $7 + 0.02n$ μs (see table 7).

To illustrate the relation to conventional ipc timing we made the same measurements on top of Mach (NORMA MK13 kernel), using the same hardware. The minimum of a series of measurements was always taken to ensure that the performance of Mach was not misrepresented. We chose Mach because it is a typical μ-kernel based on the ipc paradigm, is highly optimized, and both L3 and Mach are in use as commercial systems. Furthermore, L3's ipc functionality is roughly comparable to that of Mach.

Figure 8 compares ipc times for larger messages in Mach, L3, L3 with enforced cache flush per ipc, and the bare processor data move time, 15 ns/byte (optimal, without cache miss and write delay).

> For $8 < n < 2\text{K}$, and given sufficient cache hits, L3 ipc takes
>
> $$7 + 0.02\, n\ \mu s.$$
>
> For larger n and increasing cache misses it takes
>
> $$10 + 0.04\, n\ \mu s$$
>
> The 40 ns/byte rate is given by the external cache and memory system. In contrast, Mach takes about
>
> $$120 + 0.08\, n\ \mu s.$$

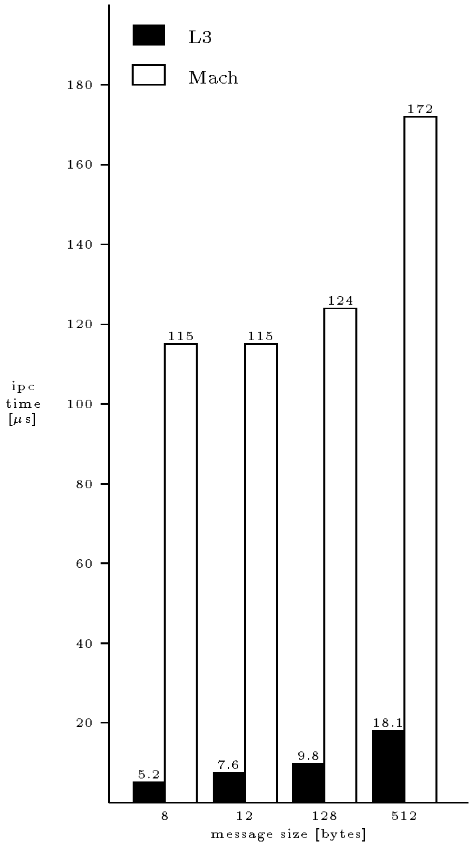

Figure 7: *486-DX50, L3 versus Mach Ipc Times*

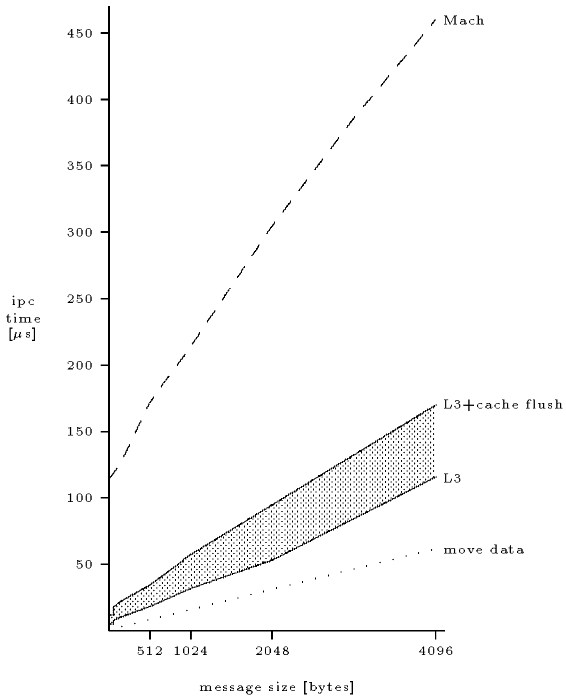

Figure 8: *486-DX50, L3 versus Mach Ipc Times*

The 80 ns/byte rate shows the effect of copying a message twice. Apparently Mach's large basic overhead floods the on-chip cache so that both kernel and user lose its benefits when using ipc.

Table 2 summarizes L3's and other systems' ipc performance. Since some of them are strictly RPC oriented, we chosed the synchronous Null-RPC for comparison. In message based systems it is implemented by two message transfers.

System	CPU, MHz	~MIPs	μs	~MIPs $\times$ μs
L3	486, 50	10	10	100
L3	486, 33	6.6	15	99
L3	386, 20	2	60	120
LRPC	FF-CVAX	2	157	314
QNX	486, 33	6.6	76	502
SRC RPC	FF-CVAX	2	464	928
SRC RPC	FF-μVAX II	1	937	937
Amoeba	68020, 15	1.5	800	1200
Mach	386, 20	2	535	1070
Mach	486, 33	6.6	346	2284
Mach	486, 50	10	230	2300
Dash	68020, 15	1.5	1920	2880

Table 2: *Null-RPC performance*

The L3 and the Mach-486 data are measured by ourselves, whereas the remaining performance data is taken from [Ber 89, Hil 92, Sch 89, Ren 88, Dra 91, Tzo 91].

7 Remarks

7.1 Introducing Ports

Since L3 ipc operates directly from thread to thread, a question arises as to how expensive the introduction of ports would be. Here the buffering feature of Mach ports will not be taken into consideration, but we extend L3 ipc to support indirection and port rights.

We use one *port link table* per address space, holding links to the system-global *port table*. Both tables are only accessible by the kernel. At user level, ports are represented by indices identifying the accessed port by: port table [port link table [port index].access].

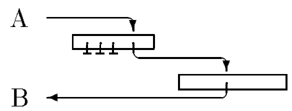

Figure 9: *Port Link and Port Table*

access is either 'read' or 'write' and determines which port link is chosen. Illegal accesses are marked in the port link table by a special value pointing to a non-mapped page. This shifts the port right checking code to the page fault handler, and legal port access requires no checking overhead at all.

The global port table points to the related thread, if any. Therefore the uid and state inspection necessary for L3 ipc can be replaced by investigating the port entry. From the performance point of view the relevant additional ipc overhead is accessing the port table.

The best case estimation (see table 5) gives 29 cycles, i.e. 0.6 μs or 12% T. A test implementation in the L3 μ-kernel led to the same result. We conclude that, in principle, port-based ipc can also be implemented efficiently.

7.2 Dash-like Message Passing

The Dash kernel uses 'restricted virtual memory remapping' [Tzo 91] for passing longer messages (several pages) to achieve low latency and high bandwidth. A Dash message consists of one or more pages which must all be located within a special part of the address space, called the ipc region. On sending, these pages are removed from the sender's address space and mapped *at the same virtual address* in the receiver's address space. In the implementation described by Tzou and Anderson the ipc region is 1 MB long and resident in physical memory.

To evaluate how our techniques work when applied to this specific type of message passing, we implemented (for experimental purpose only) Dash-like messages in the L3 kernel.

> L3 (Dash-like) ipc takes $8+n$ μs transferring n pages (including access to each page in the receiver's address space).

Running on a 16 MHz 386, the corresponding time is $61+6.3n$ μs. Dash (running on a Sun 3/50) takes $986+208n$ μs [Tzo 91].

7.3 Cache

One problem with direct mapped caches is thrashing arising from multiple working set parts being mapped to the same cache line. Such collisions can be minimized by adequate mapping of virtual pages to real memory frames [Bra 90] and by finding link orders with low collision rates [Gs 93].

On processors with direct mapped caches, similar techniques should be applied to reduce cache collisions between the ipc system and user programs. On the 486, collision avoidance by software is of minor relevance, since its on-chip cache is 4-way associative. Nevertheless, the small cache working sets achieved are very important. If the ipc system floods the cache, not only would the ipc itself be slowed down but also ipc users.

7.4 Processor Dependencies

Most of the techniques described in section 5 can be applied to any general purpose von Neumann processor, provided it supports virtual address spaces of sufficient size, permits hierarchical mapping and aliasing, and distinguishes kernel and user modes. Things become difficult if the MMU does not support hierarchical mapping or the cache does not permit synonyms (different virtual addresses mapped to the same physical address). Such hardware should only be used for one thread per processor applications.

In most cases all the methods at the architectural and algorithmic levels should be usable, except perhaps for transfering short messages via registers. Different methods may be required for parameter passing (interface level) and at the coding level, although the techniques for reducing TLB and cache misses are widely applicable.

Note that entering and leaving kernel mode is far more expensive on 486 and compatible processors than on most others. This results from the built-in segment system which automatically loads and checks segment descriptors when switching between user and kernel mode. Other processors may profit from not being burdened with segments. A hypothetical 486-compatible processor without the segment system (saving 80 cycles) and with a multi-address-space TLB (saving 45 cycles) could reduce the time needed for short ipc from 5 μs to 2.5 μs.

Although the methods are fairly general, a processor specific implementation is required to get really high performance. Since there are no compilers (as far as we know) which permit interfaces to be specified at register level and basic block sequences to be optimized by programmer supplied usage information, we had to use hand coding for the critical ipc related parts. These are combined into one module of 5 Kbytes (2082 lines of commented code).

8 Conclusions

It has been shown that fast, cross address space ipc can be achieved by applying principles like *performance based reasoning*, *hunting for new techniques* if necessary, *consideration of synergetic effects* and *concreteness*. This needs a variety of methods on all levels, from architecture down to coding, which must be combined in a design aimed at a specific performance goal right from the beginning.

The methods presented are applicable to other μ-kernels and different hardware.

The achievable quantitative gain is so high ("22 times faster") that it may perhaps count as a qualitative improvement.

Acknowledgements

I would like to thank Hermann Hrtig for various helpful discussions and Martin Gergeleit who measured the Mach ipc by means of his Jewel [Ger 92] performance measurement software. I would also like to thank Peter Dickman for proofreading this paper and helpful comments. This paper was written using LaTeX on top of L3.

References

[Acc 86] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, M. W. Young. *Mach: A New Kernel Foundation for UNIX Development.* Proceedings Usenix Summer'86 Conference. Atlanta, Georgia, June 1986, pp. 93-113.

[Ber 89] B. N. Bershad, T. E. Anderson, E. D. Lazowska, H. M. Levy. *Lightweight Remote Procedure Call.* Proceedings 12th ACM Symposium on Operating Principles, Litchfield Park, Arizona, December 1989, pp. 102-113.

[Ber 92] B. N. Bershad. *The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems.* Proceeedings Micro-kernel and Other Kernel Architectures Usenix Workshop, Seattle, April 1992, pp. 205-211.

[Bir 84] A. D. Birrel, B. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems. February 1984, pp. 39-59.

[Bey 88] U. Beyer, D. Heinrichs, J. Liedtke. *Dataspaces in L3.* Proceedings ISMM International Symposium on Mini and Microcomputers and Their Applications (MIMI '88), Barcelona, June 1988, pp. 408-414.

[Bra 90] B. K. Bray, W. L. Lynch, M. J. Flynn. *Page Allocation To Reduce Access Time of Physical Caches.* Stanford University, Technical Report CSL-TR-90-454. November 1990.

[Che 84] D. R. Cheriton. *The V Kernel: A Software Base for Distributed Systems.* IEEE Software, April 1984, pp. 19-42.

[Che 84a] D. R. Cheriton. *An Experiment Using Registers For Message Based Interprocess Communication.* Operating Systems Review, October, 1984, pp. 12-20.

[DoD 83] DoD. *Trusted Computer Evaluation Criteria.* DoD Computer Security Center, CSC-STD-001-83. August 1983.

[Dra 91] R. P. Draves, B. N. Bershad, R. F. Rashid, R. W. Dean. *Using Continuations to Implement Thread Management and Communication in Operating Systems.* Proceedings 13th ACM Symposium on Operating Principles, Pacific Grove, California, October 1991, pp. 122-136.

[Gs 93] K. Gsmann, C. Hafer, H. Lindmeier, J. Plankl, K. Westerholz. *Code Reorganization for Instruction Caches.* Proceedings 26th Annual Hawaii International Conference on System Sciences. Hawaii 1990, Vol. I pp. 214-223.

[Gui 82] M. Guillemont. *The Chorus Distributed Operating System: Design and Implementation.* Proceedings ACM International Symposium on Local Computer Networks, Firenze, April 1982, pp. 207-223.

[Hr 92] H. Hrtig, W.E. Khnhauser, W. Reck. *Operating Systems on Top of Persistent Object Systems –The BirliX Approach –.* Proceedings 25th Hawaii International Conference on Systems Sciences, IEEE Press 1992, Vol 1, pp. 790–799.

[Hil 92] D. Hildebrand. *An Architectural Overview of QNX.* Proceeedings Micro-kernel and Other Kernel Architectures Usenix Workshop, Seattle, April 1992, pp. 113-126.

[i486] Intel Corporation. *i486 Processor Programmer's Reference Manual.* Santa Clara, 1986

[Kar 89] P. A. Karger. *Using Registers to Optimize Cross-Domain Call Performance.* Proceedings 3rd Conference on Architectural Support for Programming Languages and Operating Systems. April 1989, pp. 194-204.

[Ger 92] F. Lange, R. Krger, M. Gergeleit. *JEWEL: Design and Implementation of a Distributed Measurement System.* IEEE Transactions on Parallel and Distributed Systems, November 1992.

[Lie 91] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, G. Szalay. *Two Years of Experience with a μ-Kernel Based OS.* Operating Systems Review, April 1991, pp. 51-62.

[Lie 92] J. Liedtke. *Clans & Chiefs.* Proceedings 12. GI/ITG-Fachtagung Architektur von Rechensystemen, Kiel 1992, A. Jammel (Ed.), Springer-Verlag, pp. 294-305.

[Lie 92a] J.Liedtke. *Fast Thread Management and Communication Without Continuations.* Proceeedings Micro-kernel and Other Kernel Architectures Usenix Workshop, Seattle, April 1992, 213-221.

[Lie 93] J.Liedtke. *A Persistent System in Real Use – Experiences of the First 13 Years –.* submitted to International Workshop on Object-Orientation in Operating Systems. Asheville, North Carolina, December 1993.

[Lie 93a] J.Liedtke. *Lazy Context Switching Algorithms for Sparc-like Processors.* Arbeitspapiere der GMD No. 776. St. Augustin, 1993.

[Mul 84] S.J. Mullender et al. *The Amoeba Distributed Operating System: Selected Papers 1984–1987.* CWI Tract. No. 41, Amsterdam 1987.

[Ous 90] J. K. Ousterhout. *Why Aren't Operating Systems Getting Faster As Fast as Hardware?* Proceedings Usenix Summer Conference 1990. Anaheim, California, 1990, pp. 247-256.

[Ren 88] R. van Renesse, H. van Staveren, A. S. Tanenbaum. *Performance of the World's Fastest Distributed Operating System.* Operating Systems Review, October 1988, pp. 25-34.

[Sch 89] M. D. Schroeder, M. Burroughs. *Performance of Firefly RPC.* Proceedings 12th ACM Symposium on Operating Principles, Litchfield Park, Arizona, December 1989, pp. 83-90.

[Tzo 91] S.-Y. Tzou, D. P. Anderson. *The Performance of Message-passing using Restricted Virtual Memory Remapping.* Software–Practice and Experience, Vol. 21(3), pp 251-267. March 1991.

Appendix

action	instruction	cycles	
		execution	TLB miss
load id of B	*2×ld*	2	
set msg len to 0	*ld*	1	
call kernel	*int*	71	
access B	*2×cmp+jmp*	4	1×9 [a]
load id of A	*2×ld*	2	
switch stack	*st sp*	1	
	ld sp	1	
	add+st [b]	2	
switch address	*ld*	1	
space	*flush tlb*	4	1×9 [c]
return to user	*iret*	36	2×9 [d]
inspect msg	*cmp+jmp*	2	1×9 [e]
		127	45

[a]TLB miss: thread control block of B.
[b]Sets the new kernel stack bottom address.
[c]TLB miss: kernel code.
[d]TLB miss: new kernel stack + GDT (486 built in table).
[e]TLB miss: user code.

Table 3: *Minimal Instructions for Null IPC*

action	instruction	cycles	
		execution	TLB miss
restrict port index	*and*	1	
load port link	*ld*	2	
load port entry	*ld*	2	1×9 [a]
check empty	*cmp+jmp*	2	
enter A for reply	*st*	1	
get B reply index	*ld*	1	
set B reply link	*st*	2	1×9 [b]
		11	18

[a]TLB miss: port table.
[b]TLB miss: port link table.

Table 5: *Minimal Instructions for Port Access*

action	instruction	cycles	
		execution	TLB miss
load rcv addr	*ld*	1	
calc 8 MB region	*ld+shr*	3	
load pdir B addr	*ld*	1	
copy pdir entries	*ld+st*	4	2×9 [a]
	ld+st	4	
calc dest addr	*and+add*	2	
move data startup		20	1×9 [b]
		35	27

[a]TLB miss: page directory of A + page directory of B.
[b]TLB miss: receive buffer of B.

Table 4: *Minimal Instructions for Temporary Mapping*

input	register	output
receive buffer addr	EAX	result code
send timeout	EBX	bytes 0...3 rcved msg
send message addr	ECX	bytes 4...7 rcved msg
destination thread id	EDX+ESI	source thread id
receive timeout	EDI	— scratch —
	EBP	— scratch —
stack pointer	ESP	stack pointer

Table 6: *Register Usage for IPC Parameters (486)*

message size [bytes]	L3 ipc [μs]	worst case cache penalty [μs]		Mach ipc (NORMA MK13) [μs]
8	5.2	+6.4		115
12	7.6	+9.7		124
128	9.8	+11.6		180
512	18.1	+16.3		214
1024	31.7	+25.5		305
2048	53.2	+41.6		386
4096	115.6	+54.3		460

Table 7: *Ipc timing, 486 50 Mhz*