SOFTWARE–PRACTICE AND EXPERIENCE, VOL. 21(3), 251–267 (MARCH 1991)

The Performance of Message-passing using Restricted Virtual Memory Remapping

SHIN-YUAN TZOU* AND DAVID P. ANDERSON
Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, U.S.A.

SUMMARY

DASH is a distributed operating system kernel. Message-passing (MP) is used for local communication, and the MP system uses virtual memory (VM) remapping instead of software memory copying for moving large amounts of data between virtual address spaces. Remapping eliminates a potential communication bottleneck and may increase the feasibility of moving services such as file services to the user level. Previous systems that have used VM remapping for message transfer, however, have suffered from high per-operation delay, limiting the use of the technique. The DASH design reduces this delay by restricting the generality of remapping: a fixed part of every space is reserved for remapping, and a page's virtual address does not change when it is moved between spaces.

We measured the performance of the DASH kernel for Sun 3/50 workstations, on which memory can be copied at 3.9 MB/s. Using remapping, DASH can move large messages between user spaces at a rate of 39 MB/s if they are not referenced and 24.8 MB/s if each page is referenced. Furthermore, the per-operation delay is low, so VM remapping is beneficial even for messages containing only one page. To further understand the performance of the DASH MP system, we broke an MP operation into short code segments and timed them with microsecond precision. The results show the relative costs of data movement and the other components of MP operations, and allow us to evaluate several specific design decisions.

KEY WORDS Performance Operating systems Virtual memory Remapping Message-passing Interprocess communication

INTRODUCTION

It has long been known that interprocess communication (IPC) systems should minimize software memory copying. Copying may be done in communication protocols for retransmission, in data transfer between user and kernel VASs, and in data transfer between two user VASS on a single host. [1,2] With current technological trends, copying is becoming a more severe bottleneck. Communication technology, particularly fibre optics, is advancing rapidly. [3,4] Gigabit bandwidths exist at the link level, but copying can prevent user processes from exploiting this bandwidth. When data is copied, every word passes through the CPU, the main memory and the memory bus. Each of these components-therefore is a potential bottleneck. The

*Now with the IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120 -6099, U.S.A.

0038–0644/91/030251–17$08.50
© 1991 by John Wiley & Sons, Ltd.
Received 3 November 1989
Revised 17 September 1990

load imposed by copying can have second-order effects as well. For example, the added bus traffic slows down DMA devices and the computations of other CPUs. [5]

The copying problem is exacerbated by the trend in operating system design towards moving functions such as file systems, network protocols and transaction management out of the kernel and into user VASs. A data access in such an organization usually involves several data movements between VASs. A file request from a user client process to a user-level file server might also be routed through a user-level transaction manager and a network communication manager at each end. If copying is used to move data between VASs, this organization amplifies the negative performance impact of copying.

Virtual memory remapping is a class of techniques for logically moving or copying a page of data from one VAS to another. Remapping is an attractive alternative to software memory copying because, on most machines, updating a page table entry is much faster than copying a page. Remapping has been successfully used by many operating systems. Tenex used *copy-on-write* in creating new VASs. [6] Accent used remapping for interprocess communication. [7,8] Mach refined and extended the Accent VM system. [9–11] In these systems, remapping is used primarily for functions such as copying an entire address space and opening a memory-mapped file. These operations are invoked relatively infrequently. They move or copy large amounts of data, and this data is often not resident in physical memory. High per-operation delay is therefore not a major problem.

Remapping involves more than just updating page tables. The operating system may have to manage buffers in the source and destination VASs, manipulate VM maps at various levels, adjust data to reflect address changes due to remapping, and perhaps perform remapping twice (from a user VAS to the kernel, and from the kernel to another user VAS). In fact, the experience of previous systems has been that remapping can have high per-operation delay. For example, Accent takes 1·15 ms to send a short message without using VM remapping, and 10 ms for a message with lKB of data remapped. [12] Mach takes 0·9 ms to move a null message between two VASs, and 7 ms for a message with 8KB of data remapped. †

As stated above, per-operation delay has not been important in previous systems. In high-bandwidth communication, however, remapping will be performed at a high rate on data streams to and from the network. Hence the per-operation delay of remapping is critical. The goal of our work, undertaken as part of the DASH project, is to build an operating system kernel that reaps the benefits of VM remapping without incurring high per-operation delay.

Like Accent and Mach, DASH supports protected virtual address spaces. Communication between VASs is done by message-passing (MP). The MP system is integrated with the VM system: remapping is used for data movement. The DASH design differs from others mainly in the emphasis on performance; we optimized the remapping mechanism for the needs of the MP. A significant performance gain was achieved by restricting the generality of remapping: the virtual address of a page

†To the authors' knowledge, there is no publication that systematically evaluates Mach's IPC performance. These numbers were measured by the authors on a Sun 3/160 workstation running Mach/4,3 #5.1 Version X66, which was built in January 1989. We measured the elapse time of a program that pings a message between two processes a large number of times. The quoted numbers are one half of the mean loop time. In our program. the large message is sent in copy-on-write mode, and is not accessed by the receiver at all; the time is longer if the received message is accessed.

does not change when it is remapped, and only pages in a limited address range, called the 'IPC region', can be remapped.

For the DASH kernel running on a Sun 3/50 workstation, it takes 0·96 ms to move a null message from one VAS to another, 1·186 ms for a message with 8 KB of data remapped, and 2·645 ms for a message with 64 KB of data remapped. These numbers show that we can move data between VASs at a bandwidth of 24·8 MB/s (64 KB/2·645ms). This figure rises to 39 MB/s if the data is not referenced by the receiver, as is often the case. Moreover, the per-operation delay of using remapping is low, so remapping is beneficial even for messages containing only one page.

This paper describes the design and evaluates the performance of the DASH MP system, with an emphasis on its use of remapping. In addition, we study the system performance at a 'microscopic' level by breaking down a user-level MP operation into short segments and measuring their individual performance. The results are used to evaluate and refine our design decisions.

THE DASH MESSAGE-PASSING SYSTEM

This section sketches the DASH kernel's MP and VM systems, which are described in more detail elsewhere. [13,14] A DASH kernel can support multiple VASs. There is a single *kernel VAS,* and multiple protected *user VASs.* A VAS may contain multiple concurrent *processes.* A process is an execution sequence; a VAS is the VM environment in which a process runs. The MP system is *local;* it handles communication between entities on a single host. The MP system does not handle network communication directly, but provides the interface between the various entities (network drivers, protocols, etc.) that together support network communication. [15]

Message-passing abstractions

MP operations are directed to *ports.* Each port is *bound* to a particular VAS; only processes in that VAS can receive messages from the port. Processes in multiple VASs may send messages to a single port. A VAS may have multiple ports bound to it.

The MP system supports two types of ports: *synchronous* and *asynchronous.* The operations on a synchronous port are request_reply(), get_request() and send_reply(). The client process (i.e. the sender of the request) blocks until the server process has replied. Pending requests are enqueued on the port. The operations on an asynchronous port are send() and receive(). Send() may return before the message is delivered to the receiver process. Outstanding messages are enqueued on the port.

MP operations that return a message (received(), request_reply() and get_request()) include a Boolean *immediate access* hint indicating whether the receiver plans to access the message's data pages immediately. This flag should be set to false by programs such as file servers, transaction managers, and protocols, that forward data without accessing it.

Restricted remapping

A message is represented by a *header,* which may contain pointers to separate *data pages.* The data pages need not be contiguous in virtual address. A small

message, up to 4KB, has only a header and no data pages. The header is moved between VASs by software copying, and the sender and receiver processes manage the buffers for headers. For MP operations that return a message, only the header address is passed as an argument; the MP system fills in data page pointers within the header. Data pages are unmapped from the sender's VAS and mapped into the receiver's VAS. Once a message has been received, the system ensures that it cannot be accessed by processes in other VASs.

A special part of every VAS, called the *ZPC region,* is used for messages.* Both the header and data pages must be within this region. Virtual pages in the IPC region (*IPC pages*) are allocated by the kernel. A given IPC page is 'owned' by at most one VAS at a time. When a message is sent, ownership of its data pages is transferred from the sender to the receiver. (A process may also explicitly allocate or free IPC pages using system calls). If an IPC page is owned by any VAS, it is also mapped into the kernel VAS.

When a message is transferred from one VAS to another, its data pages are assigned the same virtual addresses in the receiver's VAS as they had in the sender's (see Figure 1). The restriction has the following benefits:

1. Buffer management is minimized. A given IPC page is used by only one VAS at a time; therefore if an IPC page is allocated to the sending VAS, the corresponding IPC page in the receiver VAS is guaranteed to be free. We eliminate the need for allocating buffers when remapping a message into the receiving VAS, even when the receiver does not know the message size in advance.
2. Message headers can be copied without modification. If data page addresses changed, it would be necessary to scan and modify the message header to fix the data page pointers.
3. Headers can usually be directly copied. If a receive() precedes the corresponding send(), the receive buffer is known when the send() is done, and the header is

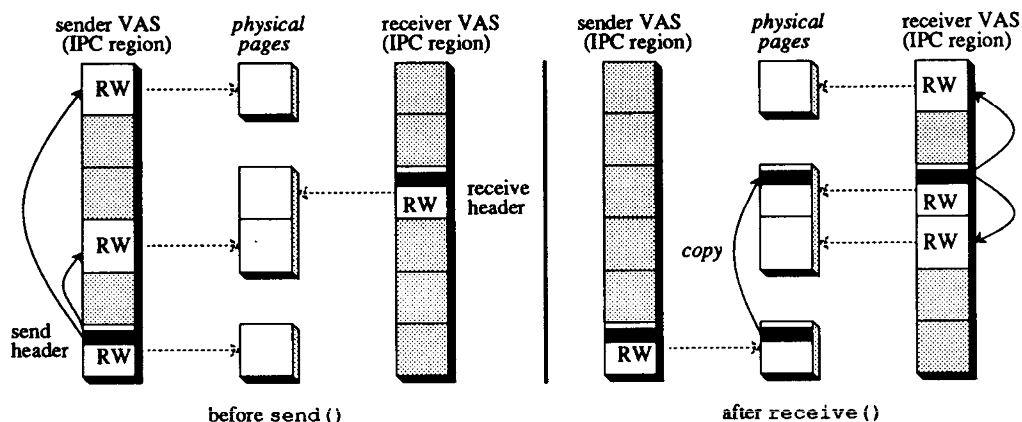

Figure 1. A message consists of a header and several data pages, all in the IPC region. When it is moved by remapping, the virtual addresses of the data pages do not change

*In the current Sun 3/50 implementation, this region is 1 MB long and is resident in physical memory.

copied directly. This is possible because the kernel has all IPC pages mapped and can therefore access both the send header and the receive header at the same time. (If a send() precedes the corresponding receive(), the header is first copied to a temporary kernel buffer, and then to the receive buffer.)

Other performance enhancements

Two other features of the DASH MP design contribute to increased performance. First, because a port is always bound to a single VAS, the destination VAS of a message can be determined at send() time. Therefore, remapping can start even if a receive() operation has not yet been issued.

Secondly, we use *lazy evaluation* (the principle of delaying an operation until it is needed) where possible. The representation of a VAS's memory map consists of two parts. The *high-level* memory map is independent of hardware architecture, and is easy to update; we always update it on remapping. The *low-level* memory map, on the other hand, depends on hardware architecture, and may be expensive to update. A page is added to the low-level memory map of a VAS on demand by the page fault handler. Lazy evaluation saves a pair of low-level map and unmap operations if a page is mapped into and out of a VAS without being accessed, but incurs the extra cost of a page fault if the page is accessed. The *immediate access* argument to the MP operations turns off lazy evaluation; it is intended to save the extra page fault.

MEASUREMENTS OF MESSAGE-PASSING THROUGHPUT

This section presents the throughput measurements made with the Sun 3/50 DASH kernel. The measurements show that VM remapping can be significantly faster than software copying in moving large data between VASs, and that the per-operation delay is low.

Experiment set-up and basic hardware performance

We designed an experiment to measure maximum data throughput between user VASs on a single host. On an idle DASH system, two processes in separate user VASs send a message back and forth using either a synchronous or asynchronous port (see Figure 2). For an asynchronous port, each pass through the loop includes two context switches, two send() operations, and two receive() operations. For a synchronous port, each pass through the loop includes two context switches, one request_ reply() operation, one get_request() operation, and one send_ reply() operation. The loop was executed 10,000 times, and the total time measured using a clock with 10 ms resolution. Variation between successive runs was negligible.

The Sun 3/50 has an MC68020 CPU running at 15 MHz. A null procedure call takes 4.3 μs; a procedure call with three arguments takes 8·1 μs. The memory management unit uses 8KB pages. Page table entries are stored in a two-level hardware map rather than in main memory. Updating a page table entry takes about 30 μs, whereas copying an 8KB page takes 2·105 ms.

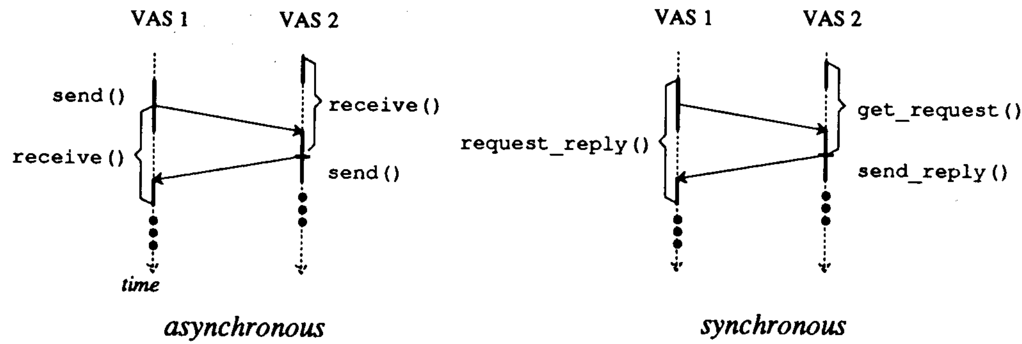

Figure 2. Time-line diagrams of the experiments used to measure round-trip message-passing delay. Arrows indicate message transfer. The cycle was repeated 10,000 times

Measurement results

Table I shows the delay of sending asynchronous message between user VASs, measured as 1/2 of the average loop time. Each table entry includes the delay of a user-level send() operation, a context switch, and a user-level receive() operation. In the receive() operation, the immediate-access flag is not set. Hence an IPC page is mapped into the low-level memory map of the receiver's VAS by the page fault handler when the receiver first accesses that page.

Table II shows the corresponding delays when the immediate-access flag is set in the receive() operation. The low-level memory map of IPC pages is updated by the

Table I. The average delay (in ms) in moving a message between two user VASs, The immediate-access flag is not set, so pages are mapped in on reference

Number of pages accessed by the receiver	Message size (number of 8KB pages) 0	1	2	4	8
0	0·957	1·077	1·163	1·334	1·679
1		1·248	1·334	1·506	1·851
2			1·500	1·673	2·020
4				2·008	2·354
8					3·026

Table II. The average delay (in milliseconds) in moving a message between two user VASs. The immediate-access flag is set, so pages are mapped in by the receive() operation

Number of pages accessed by the receiver	Message size (number of 8KB pages) 0	1	2	4	8
0	0·957	1·186	1·387	1·790	2·596
1		1·194	1·392	1·798	2·603
2			1·400	1·804	2·609
4				1·815	2·621
8					2·645

receive() operation instead of the page fault handler. A page fault is saved for each page accessed, but unnecessary map and unmap operations are done for each page not accessed. This method performs better only when the receiver accesses all message pages. The performance gain is about 12·6 per cent for 64KB messages (from 3·026 to 2·645 ms).

Figure 3 plots some entries from Tables I and II. It shows that the time needed to move a message between VASs increases linearly with the number of pages in the message. The times needed for data copying are included for comparison. In operating systems such as UNIX, moving a page between two VASs requires copying it twice (from sender to kernel, and from kernel to receiver). The dotted lines represent only the cost of copying, i.e. the cost of moving a null message is not included.

Table III lists the average incremental delay to move an 8KB page between user VASs. The incremental delay of the first page is slightly higher because of the initial overhead of loops. VM remapping uses 87 to 291 μs per page, depending on whether, and how, the page is accessed by the receiver.

Figure 4 shows the maximum data transfer throughput between DASH user VASs using MP. The numbers for VM remapping are derived from Tables I and II. The two horizontal lines represent only the throughput of copying, i.e. the cost of moving

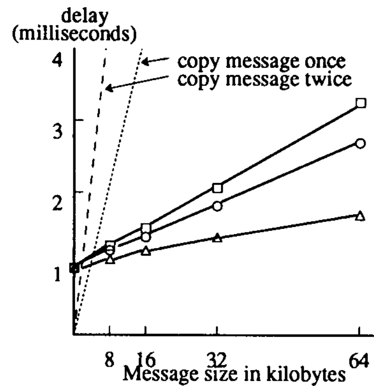

Figure 3. Measured latency of moving data between two user VASs: Δ pages are mapped on demand, but no pages are accessed; O pages are mapped by the receive operation, and all pages are accessed; ❑ pages are mapped on demand, and all pages are accessed

Table III. The incremental cost of moving 8KB pages between two user VASs

How the page is moved	Delay in ms first page	Delay in ms subsequent pages
mapped on demand but is not accessed	0·120	0·087
mapped by the receive operation	0·237	0·208
mapped on demand and is accessed	0·291	0·254
copy once	2·105	2·105
copy twice	4·210	4·210

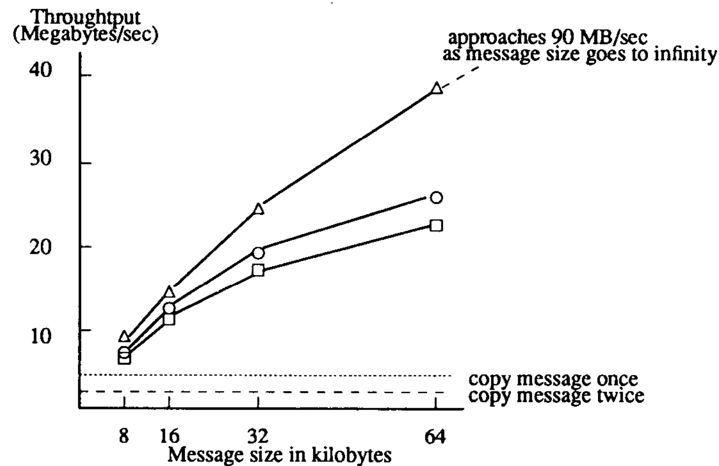

Figure 4. Measured throughput of data movement between two user VASs: Δ pages are mapped on demand, but no pages are accessed; ○ pages are mapped by the receive operation, and all pages are accessed; □ pages are mapped on demand, and all pages are accessed

a null message is not included. This Figure shows that (at least on the Sun 3/50 architecture) VM remapping is significantly faster than software copying for moving large amounts of data between VASs.

Table IV compares the performances of the two MP modes. In both cases, an 8KB message is passed back and forth between two processes. The message is mapped on demand, and is accessed by the receiver process. The reported numbers are the total elapsed time of a loop in which a message is moved between VASs twice. Synchronous mode is faster than asynchronous mode because send() and receive () are replaced by request_reply(), so a trap into the kernel is eliminated. A kernel process can perform an MP operation faster than a user process because (1) it invokes the operation by a procedure call instead of by a trap, (2) many checks are skipped and (3) IPC pages are always mapped in the kernel VAS, so no unmapping operation is needed and no page fault is generated.

MICROSCOPIC ANALYSIS OF MESSAGE-PASSING COST

This section studies the performance of the DASH MP system at a 'microscopic' level. Instead of measuring only the overall delay, we now break up an MP operation

Table IV. Comparison of different MP modes. In each case, an 8KB message is passed back and forth between two processes

Mode and process types	round trip time in ms
two user processes, asynchronous	2·496
two user processes, synchronous	2·162
a user and a kernel process, asynchronous	2·085
a user and a kernel process, synchronous	1·751

sequence into *code segments* (some as small as a few instructions) and measure the average execution time of each segment.

The following operation sequence is analysed. A process in one user VAS sends an 8KB message asynchronously to a process in another user VAS. The message is mapped into the receiver VAS on demand, and is accessed by the receiver process. This sequence exercises most of the components of the MP system. It is executed in a loop by two processes, which alternate sending each other messages.

Measurement method and its accuracy

The average execution time of some code segments is only a few microseconds, but the Sun 3/50 has a low-resolution clock (100 Hz). Our measurements therefore use a statistical approach. We added *probe calls* to both kernel and user code to divide the loop into 69 segments. Each probe call calculates the difference between the current clock reading and its previous reading, and accumulates the difference into an array location. We ran the loop 1,000,000 times. At the end, we calculated the execution time of each code segment by dividing its accumulated clock reading by 1,000,000, multiplying the quotient by 10 ms, and subtracting the overhead of the probe call (measured separately) from the product.

The value we obtained for each time interval is the mean of 1,000,000 random variables. It is itself a random variable, and has approximately a normal distribution according to the central limit theorem. To determine the accuracy of the results, we repeated the same experiment 10 times, and calculated the mean (μ) and standard deviation (σ) for each of the 69 time intervals. Figure 5 shows the σ/μ ratio for these 69 intervals. The ratio is less than 5 per cent for 59 intervals, which together account for 97 per cent of the total loop time. Hence for most cases, the 95 per cent confidence interval of a time interval is within 10 per cent of its mean (the 95 per cent confidence interval for a normal distribution is $[\mu - 1.96\,\sigma, \mu + 1.96\,\sigma]$).

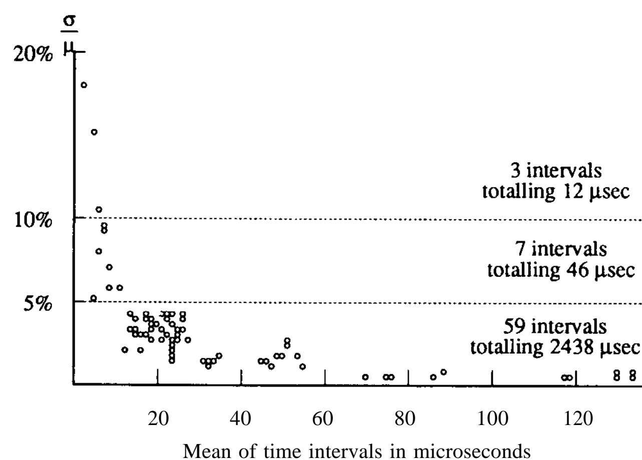

Figure 5. The coefficient of variation for the 69 time intervals. The σ/μ ratio is less than 5 per cent for 59 intervals that together account for 97 per cent of the total loop time

Results and discussion

Figure 6 summarizes the results of the microscopic measurements. It represents one half the loop, and includes the detailed delays of a user-level send() operation, a user-level receive() operation, a context switch, and the handling of a page fault. Most of the individual operations listed in the Figure are explained below.

Trapping into and returning from the kernel mode

A user process makes a kernel request by executing a trap instruction with parameters passed in registers.

Switching to and from the kernel VAS

In the Sun 3 architecture, a VAS switch is not done by the mode switch. The DASH kernel switches to the kernel VAS at the beginning of the trap, and switches back to the caller's VAS before returning from the trap.

Converting and checking user port references

A port is stored in the kernel VAS. A user process refers to a port by an index into the *user object reference* (UOR) table of its VAS. The trap handler checks the UOR table to see whether the requested MP operation is valid, and converts the index to the table into a pointer to the port.

Checking the message header

The trap handler checks whether the message header is valid. It locks the page containing the header so that other processes in the same VAS will not send out the page while it is being used by the MP system. The page is unlocked when the MP operation is completed.

Kernel-level send operation

In this scenario, the receive() operation always precedes the corresponding send() operation. The send() operation checks the MP object's reader queue, removes the receiver process from the queue, and gets parameters from its context block. Then the message is transferred; the header is copied to the receiver's header and the IPC pages are remapped into the receiver VAS. The miscellaneous delays includes the overhead of a procedure call, a loop and its initial assignments, and the check for whether a temporary header buffer is necessary. Finally, the receiver process is awakened. The MP system has several features (e.g. flow control and scheduling deadline assignment) that are not used in this scenario. These contribute to the MP operation delays, however, because of the checks that are done to see if these features are being used.

- 1231 Total elapsed time
  - 641 User-level send operation
    - 34 trap into and return from kernel mode
    - 49 switch to and from the kernel VAS
    - 15 convert and check user object references
    - 46 check the message header
    - 7 dispatch to the send operation
    - 490 kernel-level send operation
      - 138 processing before message transfer
      - 221 transfer the message
        - 74 copy the message header
        - **19 check a page descriptor in the message (A)**
        - **81 transfer a page between VASs**
          - **32 lookup and check memory maps (B)**
          - **26 invalidate a page table entry (C)**
            (this is not needed if the page hasn't been mapped)
          - **23 miscellaneous (D)**
        - 47 miscellaneous
      - 131 processing after message transfer
  - 296 User-level receive operation
    - 33 trap into and return from kernel mode
    - 46 switch to and from the kernel VAS
    - 16 convert and check user object references
    - 46 check the header of the receive message
    - 6 dispatch to the receive operation
    - 98 kernel-level receive operation
      - 91 processing before the process is blocked
      - 7 processing after the process is awakened
    - 51 complete the message transfer
      - 24 copy header from temporary buffer to receive buffer, if needed
      - **16 ensure that the page transfer is completed (E)**
      - 11 miscellaneous
  - 120 Context switch between user processes
  - 173 Access the received message
    - 37 find the address of data in a structured message
    - **134 handle a page fault**
      - **24 save states and get parameters of the fault (F)**
      - **55 lookup and check memory maps (G)**
      - **32 update a page table entry (H)**
      - **23 return from fault (I)**
    - 2 miscellaneous

Figure 6. Breakdown (in microseconds) of the elapsed time for moving an 8KB message between VASs. Numbers are accurate to within ± 10 per cent. Operations in boldface are performed for every page in the message, whereas the others are independent of the size of the message

Transferring an IPC page between VAS

The page descriptors in the message header are checked. The high-level memory map for the IPC region is checked and updated. The page is unmapped from the low-level memory map (i.e. the hardware page table) of the sender VAS if it has been mapped. The page is not immediately mapped into the low-level map of the receiver VAS.

Kernel-level receive operation

The message queue of the MP object is checked. It is always empty in this case, so the receiver process blocks. The parameters of the operation are stored in the context block of the receiver. Later a send() operation will awaken the process as described above.

Completing the transfer of the message

If the message header is in a temporary kernel buffer, it is copied to the buffer supplied by the receiver (this does not occur in the present case). An IPC page is unmapped from the low-level memory map of the sender VAS asynchronously. The receive() operation ensures that the unmapping is completed before returning. *

Context switch

This includes saving the state of the current process, making a scheduling decision, and restoring the state of the new process.

Finding data in a structured message

This scans the descriptors in the message header, and finds the data address for a given offset.

Handling a page fault

Page fault handling consists of four parts, as listed in Figure 6. The hardware saves the state of the processor and jumps to an assembler page fault handler, which in turn calls a high-level language handler. This handler determines that the faulting address is in the IPC region, checks whether the VAS of the resulting process is eligible to access the page, and finds the physical address of the IPC page. The page table entry corresponding to the faulting address is updated. Finally, the state of the CPU is restored, and the faulting instruction is resumed.

Cross-checking the results

The results of the microscopic analysis are compatible with the throughput measurements described in Tables I–IV. As an example, we can compare the incremental delay per page under the various access options.

*This mechanism is designed for shared-memory multiprocessors in which synchronous unmapping is expensive because of the inconsistency problem of translation lookaside buffers. [16] On the Sun 3/50, this is pure overhead.

1. If a page is mapped on demand but not accessed, the incremental delay includes operations A, B, D and E in Figure 6.
2. If a page is mapped by the receive operation, the incremental delay includes operations A–E, G and H.
3. All of the operations from A to I are needed if a page is mapped on demand and accessed.

Adding up the delays from the microscopic analysis (Figure 6) we obtain incremental perpage delays of 90, 205 and 250 μs for the above three cases. The corresponding delays obtained from the throughput measurements (Table III) are 87,208 and 254 μs. The differences are within the error tolerances of the measurements.

DISCUSSION AND EVALUATION

This section discusses the results in the previous sections, and uses them to evaluate the design of the DASH MP system.

Data movement dominates the cost for large messages

Figure 7 groups the numbers listed in Figure 6 by function. It shows that data movement takes 43·5 per cent of the time spent in passing an 8KB message between VASs, whereas control transfer (process sleep/wakeup and context switching) takes only 39·6 per cent. Furthermore, the delay of data movement increases as the size of the message increases, but other delays do not. The incremental delay of moving an 8KB page ranges from 87 to 254 μs (Table III). If pages are mapped on demand and accessed, data movement takes 65·5 per cent of the total time for a 32KB message, and 77·1 per cent for a 64KB message. Therefore, data movement dominates the delay of moving large messages, even though we avoided software copying. This justifies our concern with the efficiency of data movement.

39.6% Control Transfer
- 9.7% context switch
- 21.9% fixed delay of the `send()` operation
- 8.0% fixed delay of the `receive()` operation

43.5% Data movement
- 29.6% data transfer
  - 7.5% check the message header
  - 8.0% copy the message header
  - **1.5 % check a page descriptor in the message**
  - **8.1 % transfer a page between VASs**
  - 4.6%. miscellaneous
- 13.9% data access
  - 3.0% find the address of data in a structured message
  - **10.9% page fault handler**

15.6% User/kernel interface
- 5.4% trap into and returning from the kernel mode
- 7.7% switch to and from the kernel VAS
- 2.5% convert and check user object references
- 1.1% miscellaneous

Figure 7. Cost breakdown by function, showing that data movement dominates the delay of the MP operation. Operations in boldface are performed for every page in the message

The effectiveness of the DASH design

The numbers in Figure 6 show that we have achieved the design goals of restricted VM remapping. Pages are remapped only once, directly from the sender VAS to the receiver VAS. Also, the message header is copied only once when a receive() precedes the corresponding send(). No buffer allocation is needed. Finally, the delay of checking and updating the memory map for the IPC region is relatively low. This is because the data structure for the IPC region is separated from the rest of the VM system, and is simple.

The delay due to non-contiguous message organization is 148 μs. This includes 92 μs for checking the header, 19 μs for checking a page descriptor, and 37 μs for finding the location of data within a message at access time. Placing a message header in the IPC region allows it to be copied directly from a source VAS to a destination VAS. Otherwise a temporary kernel buffer and extra copying time are needed. The savings are about 70 μs for avoiding additional copying, and about 40 μs for not having to allocate a fixed-size temporary buffer. The message structure also eliminates the need for allocating buffers for the whole message; the savings are about 200 μs.

The benefit of lazy evaluation (i.e. mapping data pages on demand) depends on how a message is accessed, as well as the relative delay of manipulating the low-level memory map and handling a page fault. Operations G and H in Figure 6 are needed to map a page into the low-level memory map. If they are invoked by the receive operation instead of by the page fault handler, 47 μs (operations F and I) can be saved. In addition to G and H, operation C is needed to manipulate the low-Ievel memory map. Therefore, lazy evaluation saves 113 μs (C, G and H) if a page is not accessed, and wastes 47 μs if a page is accessed. We were concerned about the delay of handling a page fault because of the experience of other systems. We therefore added the immediate-access flag to save page faults, and optimized the page fault handler. It turned out that the delay of handling a page fault is much lower than we expected, and overriding lazy evaluation is worth while only when the number of pages accessed is at least three times the number of pages not accessed.

We believe that our experiences are applicable across a range of hardware architectures and operating system designs .This is supported by the following claims: (1) the delay of hardware-level VM remapping is about the same on the Sun 3 as on other current architectures;* (2) message-passing delays (including both the control transfer and data transfer parts) are roughly comparable in DASH and in other current message-based operating systems.

In support of the second claim, Table V shows performance figures for message-passing in DASH and other current systems: Accent, [12] Amoeba, [17] Mach,† Quicksilver [18] and the V system. [19,20]

CONCLUSIONS

In the DASH kernel, the VM system is integrated with the MP system to avoid software copying when moving large amounts of data between VASs. The purposes

*This argument may not hold on some shared-memory multiprocessors. [16]

†See footnote † on page 252.

Table V. The performance of several message-passing systems for local operations. Numbers should not be compared directly because these systems run on different hardware and reported numbers in different ways

Operating system	Hardware	Delay, ms small messages	Delay, ms large messages	Remap large messages
DASH	Sun 3/50 15 MHz 68020	0·96 send, rev. context switch	1·08 send 8KB, rcv 8KB, csw	Y
Accent	PERQ	1·15 send	10·0 send 1KB	Y
Mach	Sun 3/160	0·9 send, rev, csw	7 send 8KB, rcv 8KB, csw	Y
Amoeba	16·7 MHz 68020	0·8 RPC	2·5 8KB request, null reply	N
Quicksilver	IBM RT/PC	0·66 RPC	1·16 1KB request, null reply	N
V system	10 MHz 68000	0·94 RPC	2·12 1KB request, null reply	N

of this integration are (1) to eliminate a bottleneck in high-performance network communication, and (2) to reduce the performance penalty for moving data services into separate VASs. For these purposes, the per-operation delay must be low, and the throughput of data movement between VASs must be high. Other systems (e.g. Accent and Mach) also integrate the VM system with the MP system. However, they use this facility for tasks such as whole-file transfer and address space duplication, for which per-operation delay is not a major factor.

The primary goal of this work is to push the performance of message-passing using VM remapping to the limit. We achieved good performance by restricting the generality of remapping, and by reserving a fixed part of every VAS for remapping. The major findings of our throughput measurements are:

1. On the Sun 3/50, DASH can move data between VASs at a rate of more than 24·8 MB/s if the receiver accesses the data, and 39 MB/s otherwise. In contrast, software copying can achieve only 3·9 MB/s and consumes far more bus bandwidth.
2. Although we emphasize large messages, we have not sacrificed performance for small messages. A null message can be moved between VASs in less than one ms. This speed is comparable to that of other systems that do not used VM remapping.
3. The initial cost of using VM remapping is low. The incremental cost of adding the first page to a message is only about 30 μs higher than that of adding a subsequent page. Remapping is beneficial even when a message contains only one page.

We then did a 'microscopic' analysis of the cost of an MP operation, using a statistical approach. As expected, the results show that data movement dominates the cost of passing large messages. The analysis also explains how we reduced the cost of using VM remapping. These performance improvements may suffice to reduce the data-movement bottleneck in future operating systems.

ACKNOWLEDGEMENTS

Domenico Ferrari was involved in the experiment design, and made valuable comments on the presentation of the paper. Raj Vaswani implemented the message-passing system. Heinz Beilner assisted in developing the techniques for microscopic analysis. Comments of referees greatly improved the quality of the paper.

The work was sponsored by the California MICRO program, Cray Research, IBM Corporation, Hitachi, Ltd., Olivetti S.p.A, and the U.S. Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

REFERENCES

1. L. F. Cabrera, E. Hunter, M. Karels and D. Mosher, 'User-process communication performance in networks of computers', *IEEE Trans. Software Eng.,* **SE-14,** 38–53 (1988).
2. R. W. Watson and S. A. Mamrak, 'Gaining efficiency in transport services by appropriate design and implementation choices', *Trans. Computer Systems,* **5,** 97–120 (1987).
3. B. Leiner (ed), 'Critical issues in high bandwidth networks', *DARPA Internet RFC 1077,* November 1988.
4. W. R. Byrne, T. A. Kilm, B. L. Nelson and M. D. Soneru, 'Broadband ISDN technology and architecture', *IEEE Network,* January 1989, pp. 23–28.
5. R. Wilson, 'Designers rescue superminicomputers from 1/0 bottleneck', *Computer Design,* October 1987, pp. 61–71.
6. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, 'TENEX, a paged time sharing system for the PDP-10', *Comm. ACM,* **15,** (March) 135–143 (1972).
7. R. Rashid and G. Robertson, 'Accent: a communication-oriented network operating system kernel', *Proc. 8th ACM Symposium on Operating System Principles,* Pacific Grove, California, December 1981, pp. 64–75.
8. R. Fitzgerald and R. Rashid, 'The integration of virtual memory management and interprocess communication in Accent', *Trans. Computer Systems,* **4.** 147–177 (1986).
9. R. F. Rashid, 'From RIG to Accent to Mach: the evolution of a network operating system', *Technical Report,* Computer Science Department, Carnegie-Mellon University, May 1986.
10. R. Rashid, A Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew, 'Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures', *IEEE Trans. Computers,* **37,** (8), 896–908 (1988).
11. M. Young, A Tevanian, R. Rashid, D. Golub, J. Eppinger. J. Chew, W. Bolosky, D. Black and R. Baron, 'The duality of memory and communication in the implementation of a multiprocessor operating system', *Proc. 11th ACM Symposium on Operating System Principles,* Austin, Texas, 8–11 November 1987, pp. 63–76.
12. R. P. Fitzgerald, 'A performance evaluation of the integration of virtual memory management and inter-process communication in Accent', *Ph. D. Dissertation,* Carnegie-Mellon University, October 1986.
13. D. P. Anderson and S. Tzou, 'The DASH local kernel structure', *Technical Report No. UCB/CSD 88/463,* Computer Science Division, EECS Department, University of California at Berkeley, November 1988.
14. D. P. Anderson, S. Tzou and G. S. Graham, 'The DASH virtual memory system', *Technical Report No. UCB/CSD 88/461,* Computer Science Division, EECS Department, University of California at Berkeley, November 1988.
15. D. P. Anderson and R. Wahbe, 'The DASH network communication architecture', *Technical Report No. UCB/CSD 881462,* Computer Science Division, EECS Department, University of California at Berkeley, November 1988.
16. S. Tzou, D. P. Anderson and G. S. Graham, 'Efficient local data movement in shared-memory multiprocessor systems', *Technical Report No. UCB/CSD 87/385,* Computer Science Division, EECS Department, University of California at Berkeley, December 1987.
17. S. J. Mullender, G. Rossum, A. S. Tanenbaum, R. Renesse and H. Staveren, 'Amoeba: a distributed operating system for the 1990s', *Computer,* **23,** 44–53 (1990).

18. R. Haskin, Y. Malachi, W. Sawdon and G. Chan, 'Recovery management in QuickSilver', *Trans. Computer Systems,* **6,** 82–108 (1988).
19. W. Zwaenepoel, 'Message passing on a local network', *Ph.D. Dissertation,* Computer Science Department, Stanford University, October 1985.
20. D. R. Cheriton and W. Zwaenepoel, 'The distributed V kernel and its performance for diskless workstations', *Proc. 9th ACM Symposium on Operating System Principles,* Bretton Woods, New Hampshire, 10–13 October 1983, pp. 128–140.