A PERFORMANCE EVALUATION OF THE DASH MESSAGE-PASSING SYSTEM [1]

Shin-Yuan Tzou
David P. Anderson

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California 94720

tzou@ucbarpa.Berkeley.EDU
anderson@ernie.Berkeley.EDU

ABSTRACT

DASH is a distributed operating system kernel. Message-passing (MP) is used for local communication, and the MP system uses virtual memory remapping instead of software copying for moving large amounts of data between virtual address spaces. This design eliminates a bottleneck in high-performance communication, and increases the feasibility of moving services such as file services into user spaces.

Other systems that have used VM remapping for message transfer have suffered from high per-operation overhead, limiting the use of the technique. The DASH design is intended to reduce this overhead. To evaluate our design, we measured the performance of the DASH kernel implementation on Sun 3/50 workstations. Our throughput measurements show that large messages can be moved between user spaces at a rate of more than 30 MB/sec, an order of magnitude higher than with software copying. Furthermore, the per-operation overhead is low, so performance for small messages is not sacrificed.

To further understand the performance of the DASH MP system, we then 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.

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

1. INTRODUCTION

The DASH project has defined the communication architecture for a large, high-performance distributed system [1]. We are now designing a portable operating system kernel to run on the nodes of this system. The DASH kernel supports the distributed system architecture by providing high-performance communication, support for user-level services, and transparent remote service access.

DASH supports protected virtual address spaces (VAS's). Communication between VAS's is done by message-passing (MP). The MP system is integrated with the virtual memory (VM) system: VM remapping is used to move large messages from one VAS to another. This approach is an attractive alternative to software data copying because updating a page table entry is much faster than copying a page on most machines. By reducing software copying we hoped to increase IPC performance, and thereby increase the feasibility of user-level data services.

However, the experience of other systems has been that combining the VM system with MP can produce high per-operation overhead. For example, Accent takes 1.15 milliseconds to send a short message without using VM remapping, and 10 milliseconds for a message with 1KB of data remapped [2]. The sources of overhead include 1) buffer allocation, 2) manipulating VM system data structures representing remapped objects, 3) adjusting the representation of a message to reflect address changes due to remapping, and 4) remapping objects twice (from a user VAS to the kernel, and from the kernel to another user VAS).

One goal of the DASH kernel design is to reap the benefits of VM remapping without incurring a high per-operation cost. This paper studies the performance of the DASH MP system, with an emphasis on its use of VM remapping. In addition to the overall performance, we study the system behavior 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 paper is organized as follows. Section 2 argues against software data copying in high-performance communication. Section 3 describes the design of the DASH MP system. Section 4 describes the throughput measurements. Section 5 presents the microscopic cost breakdown of an MP operation, and the measurement technique that was used on the Sun 3/50, which has a low-resolution clock. Section 6 discusses the results and evaluates the DASH design. Finally, section 7 summarizes and concludes the paper.

2. THE DATA COPYING BOTTLENECK

It has long been known that IPC systems should avoid unnecessary software copying of data. Copying may be done in communication protocols for retransmission, in data transfer between user and kernel VAS's, and in data transfer between two user VAS's on a single host [3, 4].

With current technological trends, copying is becoming a more severe bottleneck. Communication technology, particularly fiber optics, is advancing rapidly [5]. Gigabit bandwidths exist at the link level, but a variety of bottlenecks prevent user processes from fully exploiting this bandwidth. The system bus of the host computer is often such a bottleneck, since it limits the rate at which data can be moved between the network interface and main memory [6].

Copying is especially undesirable because it is bus-intensive. For high-bandwidth data (e.g., real-time video), copying always produces heavy traffic on the system bus, even when the system has cache memory or does I/O directly to or from cache. This traffic slows down DMA devices and the computations of other CPU's.

The copying problem is exacerbated by the trend in operating system design towards moving data servers (file servers, transaction managers, etc.) from the kernel to user VAS's. Some examples are V [7], Ridge [8], and QuickSilver [9]. A data access in such an organization usually involves

several data movements between VAS's. A file request from a user client process to a user-level file server might be routed through a transaction manager and a network communication manager at each end. If copying is used to move data between VAS's, this organization amplifies the negative performance impact of copying.

In DASH, we use VM remapping to avoid memory copying. This reduces the bus bandwidth used in high-performance communication, and reduces the performance impact of moving operating system services to the user level.

3. THE DESIGN OF THE DASH MESSAGE-PASSING SYSTEM

This section describes the design of the DASH kernel's MP system, with an emphasis on how a large message is moved between VAS's.

A DASH kernel can support multiple VAS's. There is a single *kernel VAS*, and multiple *user VAS's*. A VAS may contain any number of concurrent *processes*. A process is an execution sequence; a VAS is the VM environment in which a process runs.

DASH MP 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.

3.1. Message-Passing Operations

The DASH kernel is written in C++ [10] and has an object-oriented internal structure. An MP operation (such as *send* or *receive*) is invoked on a *message-passing object* (*MPO*). Each MPO is *bound* to a particular VAS; only processes in that VAS can receive message from the MPO. Processes in multiple VAS's may send messages to a single MPO. A VAS may have multiple MPO's bound to it.

The MP system supports two modes of operation. In *stream* mode, the primitives are `send()` and `receive()`. The sender process may return before the message is delivered to the receiver process. Outstanding messages are enqueued on the MPO. In *request-reply* mode, the primitives are `request_reply()`, `get_request()`, and `send_reply()`. The client process blocks until the server process has replied. An MPO may have multiple requests pending; these requests are enqueued on the MPO.

MP operations that return a message (`receive`, `request_reply`, and `get_request`) include a Boolean *immediate access* argument indicating whether the receiver plans on referring to the message's data pages immediately. This flag is false for programs (e.g., file services and user-level protocols) that forward data to their clients rather than accessing it themselves.

3.2. Data Transfer and Virtual Memory Support

A message is represented by a *header*, and separate *data pages* are linked to the header by pointers. A small message has only a header and no separate pages. The header is moved between VAS's by copying, and the sender and receiver processes handle the buffer management by themselves. The data pages are moved by VM remapping.

Each VAS includes an *IPC region*, occupying the same address range in all VAS's. A virtual page in the IPC region is called an *IPC page*. All data to be moved between VAS's without copying must be placed in IPC pages. There is a single "meta-level" mapping from IPC pages to physical pages. The memory mapping of the IPC region in a VAS is a subset of the meta-level mapping; some pages may be read-only or not accessible. If multiple VAS's have a particular IPC page in their memory mapping, they share the same physical page.

An IPC page is moved between VAS's by changing the access rights to the page in the memory mappings of the VAS's. An IPC page appears at the same virtual address in both the source and

destination VAS. Hence "remapping" involves changing protection, not address mappings. No allocation is needed because the virtual page in the destination VAS is predetermined, and is always free. No pointer adjustment in the message header is needed because remapping does not change the virtual address of a page. Moreover, since an MPO is bound to a VAS, the destination VAS is determined when the `send` operation is invoked. Therefore, pages in a message can be directly remapped into the destination VAS, even before a `receive` operation is done.

If the *immediate access* argument to the MP operation is not set, the VM system does *lazy remapping*. The representation of a VAS's memory map is divided into two parts. The *high-level memory map* is independent of hardware architecture, and is always updated when a page is transferred. The *low-level memory map* depends on hardware architecture, and may be expensive to update. A page is mapped into the low-level memory map of a VAS on demand by the page fault handler. Lazy remapping saves a pair of map and unmap operations if a page is mapped into and out of a VAS without being accessed.

Message headers are stored in IPC pages, and are copied as follows. If a `send` precedes the corresponding `receive`, the header is first copied to a temporary kernel buffer, and then to the receive buffer. If the `receive` precedes the `send`, the receive buffer is known when the `send` is done, and the header is copied directly. This is possible because the kernel has all IPC pages mapped and can access both the send header and the receive header at the same time.

4. 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 VAS's, and that the per-operation overhead is low.

4.1. Experiment Setup and Basic Hardware Performance

We designed an experiment to measure maximum data throughput between user VAS's on a single host. On an idle DASH system, two processes in separate user VAS's send a message back and forth in either stream or request-reply mode. In stream mode, each pass through the loop includes two context switches, two `send` operations, and two `receive` operations. In request-reply mode, 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 millisecond resolution. Variation between successive runs was negligible.

The Sun 3/50 has a MC68020 CPU running at 15 MHz. A null procedure call takes 4.3 microseconds; a procedure call with three arguments takes 8.1 microseconds. 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 microseconds, while copying an 8KB page takes 2.105 milliseconds.

4.2. Measurement Results

Table 1 shows the time needed to send a stream-mode message between user VAS's, measured as 1/2 of the average loop time. Each table entry includes the time 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 hardware memory map of the receiver's VAS by the page fault handler when the receiver first accesses that page.

Table 2 shows the corresponding times when the immediate-access flag is set in the `receive` operation. The hardware mapping of IPC pages is updated by the `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

Table 1: The average elapsed time (in milliseconds) of moving a message between two user VAS's. 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

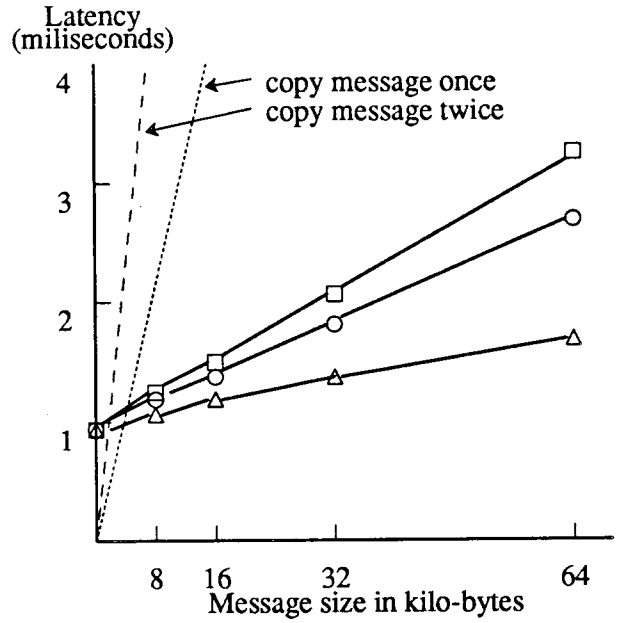
Table 2: The average elapsed time (in milliseconds) in moving a message between two user VAS's. The immediate-access flag is set, so pages are mapped in by the `receive` operation.

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

the receiver accesses all message pages. The performance gain is about 11% for 64KB messages. Figure 1 plots some entries in Table 1 and 2. It shows that the time needed to move a message between VAS's 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 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 overhead of moving a null message is not included.

Table 3 lists the average incremental time needed to move an 8KB page between user VAS's. The incremental time of the first page is slightly higher because of the initial overhead of loops. VM remapping uses 87 to 291 microseconds per page, depending on whether, and how, the page is accessed by the receiver.

Figure 2 shows the maximum data transfer throughput between DASH user VAS's using MP. The numbers for VM remapping are derived from Table 1 and 2. The two horizontal lines represent only the throughput of copying, i.e., the overhead of moving a null message is not included. This figure shows that (at least on the Sun 3/50 architecture) VM remapping is significantly faster than data copying for moving large amounts of data between VAS's.



△ pages are mapped on demand, but no pages are accessed
○ pages are mapped by the receive operation
□ pages are mapped on demand, and all pages are accessed

Figure 1: Measured latency of moving data between two user VAS's.

Table 3: The incremental cost of moving 8K-Byte pages between two user VAS's.

how the page is moved	time in milliseconds	
	first page	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

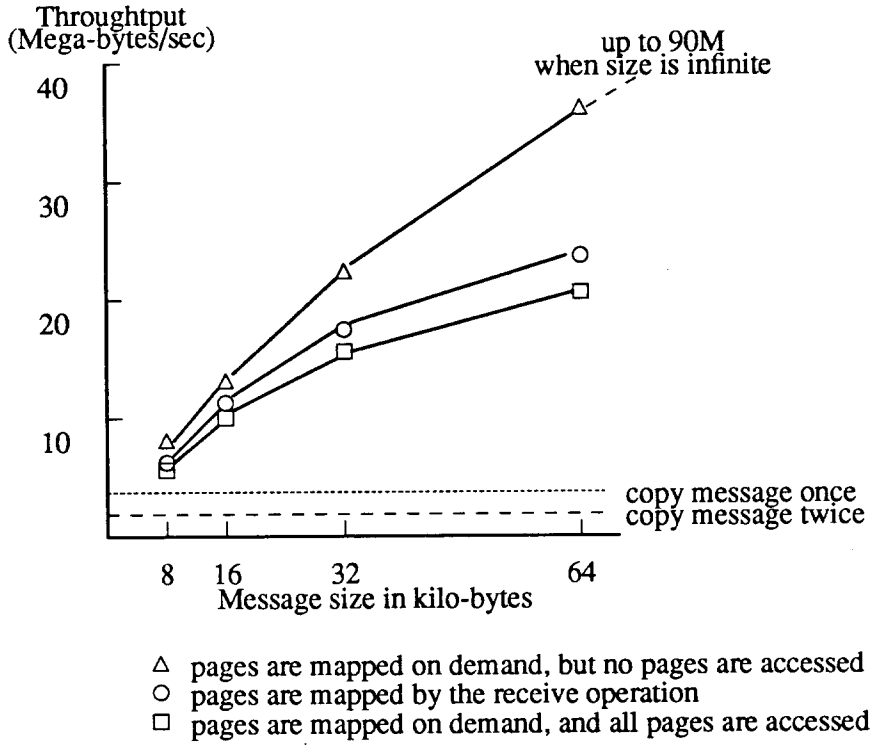


Figure 2: Measured throughput of moving data between two user VAS's.

Table 4 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 VAS's twice. Request-reply mode is faster than stream mode because two operations (`send` and `receive`) are combined into one (`request_reply`), and 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.

5. 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 elapsed time, we now break up an MP operation sequence into *code segments* (some as small as a few instructions) and measure the average elapsed time of each segment.

The following operation sequence is analyzed. A process in one user VAS sends an 8KB message in stream mode 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.

Table 4: 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 milliseconds
two user processes, stream mode	2.496
two user processes, request-reply mode	2.162
a user and a kernel process, stream mode	2.085
a user and a kernel process, request-reply mode	1.751

5.1. Measurement Method and Its Accuracy

The average time of some code segments is only a few microseconds, and 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 length of each code segment by dividing its accumulated clock reading by 1,000,000, multiplying the quotient by 10 milliseconds, and subtracting the overhead of the probe call (measured separately) from the product.

To determine the accuracy of this method, we repeated the same experiment 10 times, and calculated the mean (μ) and standard derivation (σ) for each of the 69 time intervals. Figure 3 shows the $\frac{\sigma}{\mu}$ ratio for these 69 intervals. The ratio is less than 5% for 59 intervals, which together account for 97% of the total loop time. Hence for most cases, the 95% confidence interval of a time interval is within 10% of its mean.

An alternative measurement approach for low-resolution clocks uses a run-time profiler that builds histograms from periodic samplings of the program counter (e.g., the UNIX *gprof* facility [11]). We did not use this technique because it reports statistics only at the granularity of a procedure call, while we are interested in code segments smaller than a procedure call. More than two thirds of the code segments shown in Figure 4 are sub-procedure segments.

5.2. Results and Discussion

Figure 4 summarizes the results of the microscopic measurements. It represents one half the loop, and includes the detailed costs 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.

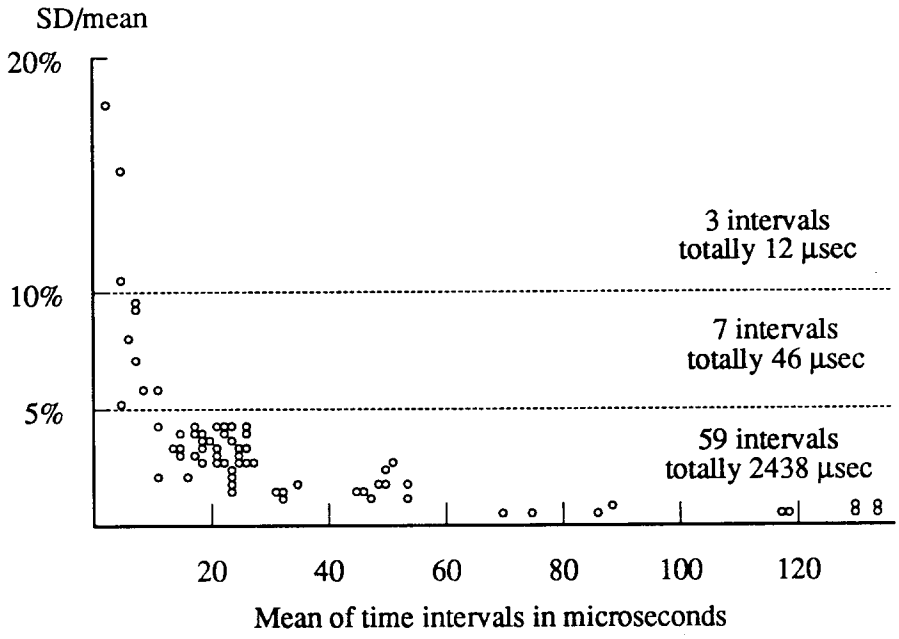


Figure: 3 The coefficient of variation for the 69 time intervals. The $\frac{\sigma}{\mu}$ ratio is less than 5% for 59 intervals that together account for 97% of the total loop time.

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 VAS's**
      - **32 look up 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 look up and check memory maps (G)**
  - **32 update a page table entry (H)**
  - **23 return from fault (I)**
- 2 miscellaneous

Figure 4: Breakdown (in microseconds) of the elapsed time for moving an 8KB message between VAS's. Numbers are accurate to within ± 10%. Operations in boldface are performed for every page in the message, while the others are independent of the size of the message.

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 object references:

An MP object is stored in the kernel VAS. A user process refers to this object 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 object.

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 then transferred; the header is copied to the receiver's header, and the IPC pages are remapped into the receiver VAS. The miscellaneous costs 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 overhead, however, because of the checks that are done to see if these features are being used.

Transferring an IPC page between VAS's:

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 transferring 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.[2]

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:

---

[2] This mechanism is designed for shared-memory multiprocessors in which synchronous unmapping is expensive because of the inconsistency problem of translation lookaside buffers [12]. On the Sun 3/50, this is pure overhead.

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 4. The hardware saves the state of the processor and jumps to an assembler page fault handler, which in turn calls a C++ handler. The C++ handler determines that the faulting address is in the IPC region, checks whether the VAS of the faulting 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.

5.3. Cross-Checking the Results

The results of the microscopic analysis are compatible with the throughput measurements described in Section 4. As an example, we can compare the incremental cost per page under the various access options.

- If a page is mapped on demand but not accessed, the incremental cost includes operations A, B, D, and E in Figure 4.
- If a page is mapped by the receive operation, the incremental cost includes operations A - E, G, and H.
- All of the operations from A to I are needed if a page is mapped on demand and accessed.

Adding up the costs from the microscopic analysis (Figure 4) we obtain incremental per-page costs of 90, 205, and 250 microseconds for the above three cases. The corresponding costs obtained from the throughput measurements (Table 3) are 87, 208, and 254 microseconds. The differences are within the error tolerances of the measurements.

6. DISCUSSION AND EVALUATION

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

6.1. Data Movement Dominates the Cost for Large Messages

Figure 5 groups the numbers listed in Figure 4 by function. It shows that data movement takes 43.5% of the time spent in passing an 8KB message between VAS's, while control transfer (process sleep/wakeup and context switching) takes only 39.6%. Furthermore, the cost of data movement increases as the size of the message increases, but other costs do not. The incremental cost of moving an 8KB page ranges from 87 to 254 microseconds (Table 3). If pages are mapped on demand and accessed, data movement takes 65.5% of the total time for a 32KB message, and 77.1% for a 64KB message. Therefore, data movement dominates the cost of moving large messages, even though we avoided software copying. This justifies our concern with the efficiency of data movement.

6.2. The Effectiveness of the DASH Design

Because of the experience of Accent, we were concerned about the overhead introduced by combining the VM system and the MP system. In order to reduce this overhead, we designed a special IPC region, and special remapping semantics. The numbers in Figure 4 show that we have avoided all the operations we wanted to avoid. The message header is copied only once when a `receive` precedes the corresponding `send`. Also, pages are remapped only once, from the sender VAS directly to the receiver VAS. No buffer allocation is needed. Finally, the cost of checking and updating the memory map for the IPC region is relative low. This is because the

39.6% Control Transfer
- 9.7% context switch
- 21.9% fixed overhead of the `send` operation
- 8.0% fixed overhead 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 VAS's**
  - 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 5: Cost breakdown by function, showing that data movement dominates the cost of the MP operation. Operations in boldface are performed for every page in the message.

data structure for the IPC region is separated from the rest of the VM system, and is simple.

The overhead due to non-contiguous message organization is 148 microseconds. This includes 92 microseconds for checking the header, 19 microseconds for checking a page descriptor, and 37 microseconds for finding the location of data within a message. Placing a message header in the IPC regions 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 microseconds for avoiding additional copying, and about 40 microseconds 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 microseconds.

The benefit of lazy mapping (i.e., mapping data pages on demand) depends on how a message is accessed, and the relative cost between manipulating the low-level memory map and handling a page fault. Operations G and H in Figure 4 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 microseconds (operations F and I) can be saved. In addition to G and H, operation C is needed to manipulated the low-level memory map. Therefore, lazy mapping saves 113 microseconds (C, G and H) if a page is not accessed, and wastes 47 microseconds if a page is accessed. We were concerned about the overhead of handling a page fault because of the experience of other systems. We therefore added the immediate-access flag to save page faults, and we optimized the page fault handler. It turned out that the cost of handling a page fault is much lower than we expected,

and overriding lazy remapping is a win only when the number of pages accessed is at least three times of the number of pages not accessed.

We believe that our main conclusions are applicable across a range of hardware architectures and operating system designs. This is supported by the following claims: 1) that the cost of hardware-level VM remapping is about the same on the Sun 3 as on other current architectures[3]; 2) that message-passing costs (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 5 shows performance figures for message-passing in DASH and other current systems: Accent [2], LYNX [13], Mach [14], Quicksilver [9], Topaz [15], and the V system [16, 17].

7. SUMMARY

In DASH, the VM system is integrated with the MP system to avoid software copying when moving large amounts of data between VAS's. The purposes 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 VAS's. For these purposes, the overhead of using VM remapping must be low, and the bandwidth of moving data between VAS's must be high. Some early systems (e.g., Accent) also integrate the VM system with the MP system. However, they use this facility for such tasks as whole-file transfer and address space duplication, for which per-operation overhead of remapping is not a major factor.

The major findings of our throughput measurements are:

- On the Sun 3/50, DASH can move data between VAS's at a rate of more than 30 MB/sec, an order of magnitude higher than the bandwidth of software copying.
- Although we emphasize large messages, we have not sacrificed performance for small messages. A null message can be moved between VAS's in less than one millisecond. This

Table 5: Performance comparison of several message-passing systems. DASH is comparable to other systems.

operating system	hardware	request/reply time (milliseconds)	
		two small messages	one small, one large message
Accent	PERQ	1.15	10.0 (1KB) [4]
DASH	Sun 3 (15 MHz MC68020)	1.59	2.16 (8KB each direction)
LYNX	Butterfly (8MHz MC68000)	2.82	4.42 (1KB each direction)
Mach	IBM RT/PC	4.0	5.1 (1KB)
Quicksilver	IBM RT/PC	0.66	1.16 (1KB)
Topaz	Firefly (DEC MicroVAX)	0.94	1.482 (1440 bytes)
V system	10 MHz MC68000	0.94	2.13 (1KB)

[3] This argument may not hold on some shared-memory multiprocessors [12].

[4] These numbers are for a one-way send operation only, as reported in [2]. The times for a receive operation are similar. A request/reply operation would involve two send and two receive operations.

speed is comparable to that of other systems that do not use VM remapping.

- The initial overhead of using VM remapping is low. The incremental cost of adding the first page to a message is only about 30 microseconds higher than that of adding a subsequent page. Therefore, 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 cost breakdown also explains how we reduced the overhead of using VM remapping. Several important design decisions are:

- To avoid dynamic heap allocation while supporting variable-size messages.
- To remap pages from the source VAS to the destination VAS directly.
- To avoid adjusting pointers when pages in a structured message are remapped.

8. ACKNOWLEDGEMENTS

We would like to thank Domenico Ferrari for his involvement with the experiment design, and suggestions on improving the presentation of the paper. We are also thankful to Heinz Beilner for his assistance in developing the techniques for microscopic analysis, and Raj Vaswani for implementing the message-passing system.

References

1. D. P. Anderson and D. Ferrari, "The DASH Project: An Overview", Technical Report No. UCB/Computer Science Dpt. 88/405, Computer Science Division, EECS, UCB, Berkeley, CA, Feb. 1988.
2. R. P. Fitzgerald, *A Performance Evaluation of the Integration of Virtual Memory Management and Inter-Process Communication in Accent*, Ph.D. Dissertation, CMU, Pittsburgh, PA, Oct. 1986.
3. L. F. Cabrera, E. Hunter, M. Karels and D. Mosher, "A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD", Technical Report No. UCB/Computer Science Dpt. 84/217, Computer Science Division, EECS, UCB, Berkeley, CA, Dec. 1984.
4. D. D. Clark, "Modularity and Efficiency in Protocol Implementation", *DARPA Internet RFC 817*, July 1982.
5. H. Rudin, "Trends in Computer Communications", *IEEE Computer*, Nov. 1986.
6. R. Wilson, "Designers Rescue Superminicomputers From I/O Bottleneck", *Computer Design*, Oct. 1987, 61-71.
7. D. R. Cheriton, "The V Kernel: a Software Base for Distributed Systems", *IEEE Software 1*, 2 (Apr. 1984), 19-43.
8. E. Basart, "The Ridge Operating System: High Performance through Message-Passing and Virtual Memory", *Proc. of the IEEE 1st International Conf. on Computer Workstations*, San Jose, California, Nov. 11-14, 1985, 134-143.
9. R. Haskin, Y. Malachi, W. Sawdon and G. Chan, "Recovery Management in QuickSilver", *Trans. Computer Systems 6*, 1 (Feb. 1988), 82-108.
10. B. Stroustrup, "The C++ Programming Language", *Addison-Wesley*, 1986.
11. S. L. Graham, P. B. Kessler and M. K. McKusick, "gprof: A Call Graph Execution Profiler", *Proc. of the SIGPLAN Notices '82 Symposium on Compiler Construction, SIGPLAN Notices Notices 17*, 6 (JUNE 1982), 120-126.
12. S. Tzou, D. P. Anderson and G. S. Graham, "Efficient Local Data Movement in Shared-Memory Multiprocessor Systems", *Technical Report No. UCB/Computer Science Dpt. 87/385*, Berkeley, CA, Dec. 1987.
13. M. L. Scott and A. L. Cox, "An Empirical Study of Message-Passing Overhead", *7th International Conference on Distributed Computing Systems*, Berlin, Sep. 1987, 635-643.
14. A. Z. Spector, J. L. Eppinger, D. S. Daniels, R. Draves, J. J. Bloch, D. Duchamp, R. F. Pausch and D. Thompson, "High Performance Distributed Transaction Processing in a General Purpose Computing Environment", CMU, Pittsburgh, PA Computer Science Dpt. Technical Report, Sep. 9, 1987.
15. M. Burrows and M. Schroeder, "Performance of Firefly RPC", Internal Report, Dec. Systems Research Center, Nov. 1987.
16. W. Zwaenepoel, *Message Passing on a Local Network*, Ph.D. Dissertation, Computer Science Dpt., Stanford Univ., Oct. 1985.
17. D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", *Proc. of the 9th ACM Symp. on Operating System Prin.*, Bretton Woods, New Hampshire, Oct. 10-13, 1983, 128-140.