Efficient Local Data Movement in Shared-Memory Multiprocessor Systems[1]

Shin-Yuan Tzou
David P. Anderson
G. Scott Graham[2]

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

ABSTRACT

The DASH research project is addressing the general problem of achieving high-performance network communication in large-scale distributed systems. The efficiency of moving a large amount of data between virtual address spaces (both user and kernel) on a single machine is a major component of this problem. Virtual memory (VM) remapping, as opposed to memory copying, is an attractive approach to moving data. However, remapping in shared-memory multiprocessors can be costly due to the problem of translation lookaside buffer (TLB) inconsistency.

This paper describes the design of the DASH mechanism for moving data between virtual address spaces. This design integrates interprocess communication (IPC), virtual memory, and process scheduling mechanisms. By adopting a particular choice of IPC semantics based on a protected shared memory model, we are able to eliminate many of the overheads that would otherwise arise from VM remapping in shared-memory multiprocessors. Put simply, we reduce the need for synchronous unmapping and, when it is necessary, we do it efficiently.

1. INTRODUCTION

Future distributed systems may offer a wide variety of services, many of them (such as those based on digital audio and video) requiring high-bandwidth interprocess communication (IPC). The performance of IPC is likely to become the dominant measure of these systems. This performance involves several components:

---

[1]Sponsored by MICRO, IBM, Olivetti, MICOM-Interlan, NSERC of Canada, Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871. Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

[2]On research leave from the Department of Computer Science and the Computer Systems Research Institute, University of Toronto, Toronto, Canada

- The movement of data across networks.
- The movement of data between device interfaces (including network interfaces) and main memory.
- Delays due to process scheduling and synchronization.
- The movement of data between virtual address spaces on a single host. In a client/server structure, services are moved out of the kernel into user-level address spaces. Access to a local service requires data movement between two user spaces, and access to a remote service requires data movement between user and kernel spaces at both ends.

This paper is concerned with the last area: how to move large amounts of data efficiently between address spaces. We describe a portion of the prototype distributed system being developed by the DASH research project at UC Berkeley [AFR87]. Our approach is based on the following observations of likely technological trends over the next few years:

- Processor and network speeds will increase much faster than the access time for main memories.
- Shared-memory multiprocessors will become increasingly common.

Software memory copying is the straightforward way to move data between spaces. However, memory bandwidth is improving at a slower rate than processor and network speeds. Thus, memory copying is likely to be an IPC bandwidth bottleneck and a major source of IPC delay. On shared-memory multiprocessors, memory access is usually provided by a single bus, shared by all processors. The bus traffic generated by memory copying will degrade overall system performance.

DASH has therefore adopted virtual memory (VM) remapping as the basis for moving large messages between spaces. These messages are assumed to include entire VM pages in their data part. In its simplest form, memory remapping for IPC involves unmapping message data pages from the address space of the sending process and mapping them into the address space of the receiving process. There are at least two performance problems to be solved for this approach.

Basic performance

In experiments on the Accent system [FiR86], copying a resident 512-byte page took 0.357 milliseconds (ms), whereas mapping a resident page took 1.1 ms. Thus, VM overhead can make memory remapping more expensive than copying.

Consistency of multiple TLB's

Consider a shared-memory multiprocessor in which the page table stored in main memory is partially replicated in the translation lookaside buffer (TLB) on each processor. Whenever the page mapping is updated on one processor, the TLB's may be inconsistent. Software mechanisms for TLB consistency that assume the page table is updated only on page-in or page-out operations will not work effectively when pages are remapped dynamically.

This paper describes the portion of the DASH IPC system involving data movement between VM spaces. It is one of the features that supports high-performance communications, a crucial feature to the success of very large distributed systems. The issues of

VM remapping and TLB inconsistency are given more attention in Section 2. Section 3 shows how an integration of IPC semantics, a protected shared memory model, process scheduling, and VM mechanisms can solve the problem effectively. Related work is surveyed in Section 4.

2. THE IPC LOCAL DATA MOVEMENT PROBLEM

The DASH IPC system moves data between address spaces by VM remapping. This creates two problems: 1) remapping may create TLB inconsistency on shared-memory multiprocessors, and 2) remapping itself is potentially expensive. These problems are examined in more detail in this section.

In shared-memory multiprocessors, VM mapping information may be partially replicated in several places (see Figure 1): system-wide machine-independent structures (e.g., IPC page ownership), system-wide machine-dependent structures, and per-processor TLB's. Partial replication introduces two types of potential inconsistency: *vertical* (among different levels, such as the machine-independent part and the machine-dependent part) and *horizontal* (among different processors on the same level, such as the TLB's).

2.1. Remapping and IPC Semantics

With the VM remapping[3] approach to IPC, message pages will be mapped into the address space of the receiving process at some level of mapping. There are several

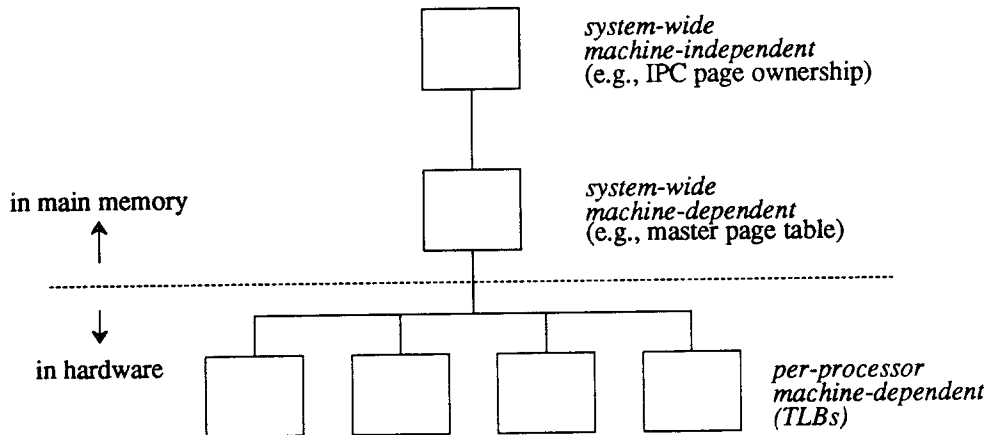


Figure 1: Memory Mapping Information is Replicated in Various Forms.

[3] We use the term *remap* to denote the act of changing the mapping (of a virtual page). In certain cases, this may consist of changing only the VM access protection for a page.

options on how to change the mappings of the address spaces of the sending and receiving processes. They differ in the particular choice of IPC semantics.

- One option is to simply add the mapping to the receiver's space, leaving the sender's mapping unchanged. The message page is then read/write shared between the address spaces. While efficient, it is not a general solution, as it requires a high degree of coordination (and trust) between address spaces for correct operation.
- A second option is to use *by-value* semantics for message passing. This can be implemented using a *copy-on-write* mechanism. The message page is mapped into the receiver's space, but is set to *read-only* protection in both the receiver's and the sender's space. Both the sending and receiving processes then have their own logical copy of the data. On a write attempt by either process, the page fault handler will create a new physical copy of the message page.
- A third option is for the sending process to ''give away'' the IPC page. The page is removed from the sender's space and added to the receiver's space. The receiving process will then have the only copy of the page.

In all three options, it is not always necessary to map the message page into the receiver's address space at the hardware level. If there is not already a hardware mapping of the page, it is sufficient to leave it unmapped at the hardware level and simply record the change in a software structure. If the page is later referenced, the page fault handler can decide how the page should be mapped at the hardware level.

Both the second and third options require a change in the mapping of the message page in the address space of the sending process. We now address the question of how this can be done.

2.2. TLB Inconsistency

A TLB is a cache of memory mappings used by the memory management unit (MMU) to speed up address translation in VM systems. We assume that there is one TLB per processor, and that TLB's are flushed or invalidated only by software (other possibilities are discussed in Section 4).

Maintaining complete TLB consistency is expensive. Our approach is to tolerate *limited TLB inconsistency*. This involves the following steps:

- We categorize different types of TLB inconsistencies, and determine the potential consequences of each type.
- We define IPC semantics that allow limited TLB inconsistency. For example, the receiver of a message can specify that the message need not be immediately unmapped from the sender's space.
- We develop mechanisms for implementing these IPC semantics with the minimal amount of work: TLB consistency is achieved only where it is needed. In doing so, we can reduce VM overhead significantly.

The first step is to categorize the types of inconsistency:

Missing mappings

When a message is received, the new mapping of its data pages is recorded in high-level structures, but TLB's either have no entry for the page or have an invalidated entry. When a page is referenced from a processor, the TLB refresh hardware (after a page fault, perhaps) will bring the mapping into the TLB of that processor. However, the page may still be marked as invalid in other TLB's. This inconsistency causes no problems; a reference to the page from another processor will succeed, perhaps after a page fault.

Extra mappings

An extra mapping can occur after a process sends a message if that page is unmapped from the address space of the sending process, but not all TLB's are updated. (This is the particular semantics for DASH; other semantics discussed in Section 2.1 also lead to an unmap operation.) The mapping for the IPC page may still exists in the TLB's of other processors; these are the extra mappings. This inconsistency may constitute a security or protection violation.

Incorrect mappings

An incorrect mapping occurs when a physical page is mapped into an address space on one processor while an old mapping for the same virtual page exists in the TLB on another processor. This problem can be avoided by not allocating new pages until old mappings have been invalidated.

Extra mappings must be removed to maintain TLB consistency. The TLB's of all processors (or at least, of those processors having the extra mapping) must be invalidated. This is potentially expensive if it is done synchronously with the unmap operation, as interprocessor interrupts must be issued from the processor doing the unmapping, and it must wait until all of these requests are handled successfully.

The unmapping operation could be made more efficient by batching the TLB invalidations to other processors and doing them asynchronously. This can certainly be done for unmappings due to page-out and address space deletion, since latency is not a critical factor in these cases. However, we cannot apply this technique directly to the unmapping of pages for IPC, since latency may be critical. However, as will be shown in Section 3, there are cases where IPC unmapping can be done asynchronously. Furthermore, when synchronous unmapping must be done it is often possible to invalidate only a small subset of TLB's.

3. THE DASH DESIGN FOR LOCAL DATA MOVEMENT

3.1. Overview

At a high level, the DASH distributed system has a client/server structure. A server may be implemented as a user-level process that communicates with local and remote clients using message-passing style IPC. This paper is concerned only with local data movement. This, and synchronization, are the major components of local user-level IPC. Network user-level IPC involves movement of data from user to kernel space at the sending end, and from kernel to user space at the receiving end. The mechanisms for

kernel/kernel network communication are described elsewhere [AFR87].

In DASH, the concepts of *process* and *virtual address space* are decoupled; multiple processes may execute in a single address space. A DASH virtual address space is divided into three regions: 1) a *general region* for code and data private to a space; 2) a *shared-segment region* for shared read-only programs and libraries, and 3) an *IPC region* for data transfer between spaces. Of these, this paper is concerned only with the IPC region; we will use *VM system* to refer only to the portion dealing with the IPC region.

The local user IPC system involves the following layers:

- A *message-passing (MP) system* providing operations to allocate, access, send, receive and deallocate messages. It is implemented by a user-level library that handles some operations itself and traps to the kernel to perform others.
- A *protected shared memory* facility that supports *ownership* of pages in the IPC region. This facility is used by the MP system for data movement, and can also be used directly by user processes via system calls (which are themselves implemented as MP operations).
- The *Logical VM mapping* of the VM system that is machine-independent and supports the protection of shared memory.
- The *Physical VM mapping* of the VM system that is machine-dependent and underlies the logical VM mappings. It encapsulates mechanisms for dealing with TLB inconsistency.

This structure is depicted in Figure 2.

3.2. The DASH Message Passing System

A *message* in DASH is the abstraction of an array of *n* bytes. Messages can be accessed only through a fixed set of operations (allocation, deallocation, send, receive, append, duplicate access, and so forth).

A message might not be physically contiguous; the actual representation is hidden by the operations. In the current implementation, a message is represented by 1) a *header* containing general information, 2) a set of *descriptors*, and 3) an optional data part. Each descriptor points to part of the message data; this can be a pointer to the data part of the message header, a random-size "small-block", or an IPC page.

The MP system defines the notion of *ownership* of messages by address spaces. Ownership is acquired by allocating, receiving, or duplicating a message, and relinquished by sending or deallocating it. A process may access a message only if its address space owns the message.

The receiver of a message may or may not *trust* the sender to not modify the message data after it has been received. The MP `send()` has a `trusted` flag that is set when the sender believes that the receiver trusts it. The MP `receive()` operation has 1) an `immediate_access` flag that is set if the receiver intends to access the message's IPC page data, and 2) a `trust_sender` flag that is set if the receiver trusts the sender. These flags allow the MP operations to be performed with increased efficiency, as will be discussed.

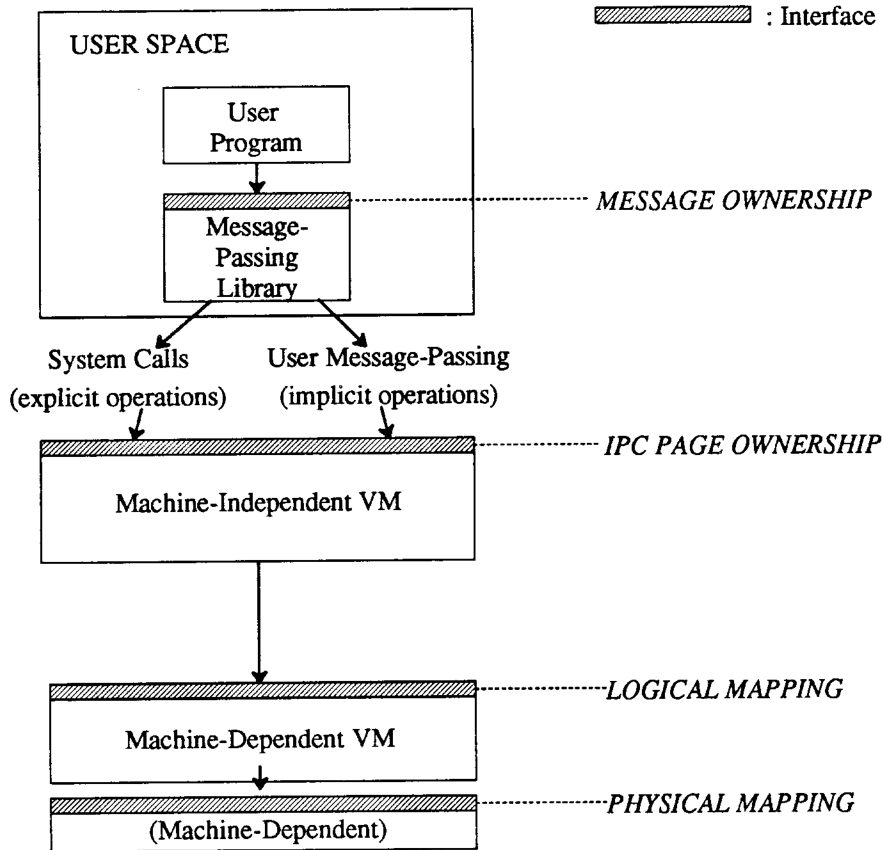


Figure 2: Logical Levels and Software Structure.

3.3. The IPC Region: Protected Shared Memory

The *IPC region* is a special part of all address spaces (user and kernel) and is composed of *IPC pages*. All data to be moved between address spaces without copying must be placed in IPC pages. The IPC region appears at the same address in all spaces, and IPC pages are not mapped to different physical pages in different spaces (this eliminates the need to modify page pointers on remapping). However, different spaces may have different VM protection on IPC pages. Allocation of IPC pages is handled centrally by the kernel; hence there is no need to coordinate with user-level allocation.

An IPC page may be *owned* by an address space. An IPC page is associated with a physical memory page while any space owns it. IPC page ownership can be gained or given up *explicitly* (by `get_ownership()`, `duplicate_ownership()`, and

`release_ownership()` system calls) or *implicitly* (by MP operations; see below). More than one space may own a page simultaneously, and a space may have multiple ownerships of a single page. Ownership has a Boolean attribute: `read/write` or `read-only`. When ownership of a page is held by more than one space, it is read-only to all of them. Processes are expected to obey the following rule for IPC pages:

> **A process should only (directly) access pages it owns, and should only write to pages for which it has a read/write ownership.**

Violations of this rule can result in (1) an exception to the violator, or (2) non-deterministic behavior that can damage other processes in same space (e.g., overwrite the data in messages they read or write). However, the rule is not always strictly enforced. Depending on the flags present in the MP operations, the MP system may defer unmapping the page from the sender's space. Hence a process's hardware mapping may contain a mapping of an IPC page that is no longer owned.

The relationship of IPC page ownership to message ownership is as follows. Ownership of a message implies ownership of the message's IPC pages. Message descriptors for IPC pages have a flag indicating whether the ownership is read-only. At the MP level, a message is always writable; the `write()` operation will check the flag, and make a copy in software if needed.

Send operations relinquish ownership of IPC pages referred to in the descriptors of the message being sent; receive operations gains ownership of IPC pages referred to in descriptors of the received message. Page ownership is encapsulated in MP operations; a process that uses only MP has to concern itself only with message (not page) ownership.

The IPC region can be viewed as *protected shared memory*. It is *shared* in the sense that an IPC page appears at the same virtual address in every space. It is *protected* because the VM system can, on request, ensure that an IPC page owned by a space cannot be accessed from processes in other spaces. Since the IPC page exists at the same virtual address in all spaces, the virtual page number does not change when it is transferred between spaces. The kernel changes only the VM protection of the page in the various spaces.

The IPC region's page ownership information is stored in an array (or hash table) indexed by virtual page numbers. Each entry contains (1) the address of the physical page associated with the IPC page, if any; (2) the total ownership count; (3) the read/write attribute, and 4) a list of spaces that own the page. The information for each space includes (a) the ownership count of this space and (b) the set of processors having this page in their logical mapping. The purpose of maintaining this set (which is represented as a bitmap) is to reduce the overhead of unmapping. Whenever a page is mapped on a processor, the corresponding bit of the bitmap is set. When it is necessary to unmap a page from a space, it is only unmapped on those processors present in the bitmap.

3.4. Logical VM Mapping and Unmapping

The IPC page ownership mechanism uses logical VM mapping and unmapping operations to enforce page ownership rules. These operations are part of the interface to the machine-dependent part of the the VM system implementation (this part has a machine-independent interface).

The interface provides a single function that synchronously adds a mapping on the calling processor. This is used both for `receive()` operations with the `immediate_access` flag, and on page faults resulting from initial access to pages received without this flag.

An IPC page is logically unmapped from a space when its last ownership by that space is relinquished (i.e.,on `send()` operations or when the page is deallocated). When the ownership count of a page goes from 1 to a larger number, the page protection is changed from read/write to read-only; this is also considered an unmapping operation for the discussion here. The interface provides three logical unmapping operations. One of the parameters of each operation is a bitmap representing the set of processors on which the page to be unmapped.

- `synchronous_unmap()`. For each processor in the bitmap, issue an interprocessor request (typically using an interprocessor interrupt mechanism) to physically unmap the page on that processor, then busy wait until all requests are done.
- `fast_asynchronous_unmap()`. For each processor in the bitmap, issue an interprocessor request to physically unmap the page on that processor, and return immediately.
- `slow_asynchronous_unmap()`. For each processor in the bitmap, insert the page in an *unmap queue* for that processor, and return immediately. The pages will be physically unmapped on each processor at some future point (e.g., when there is a context switch or when a watchdog timer expires).

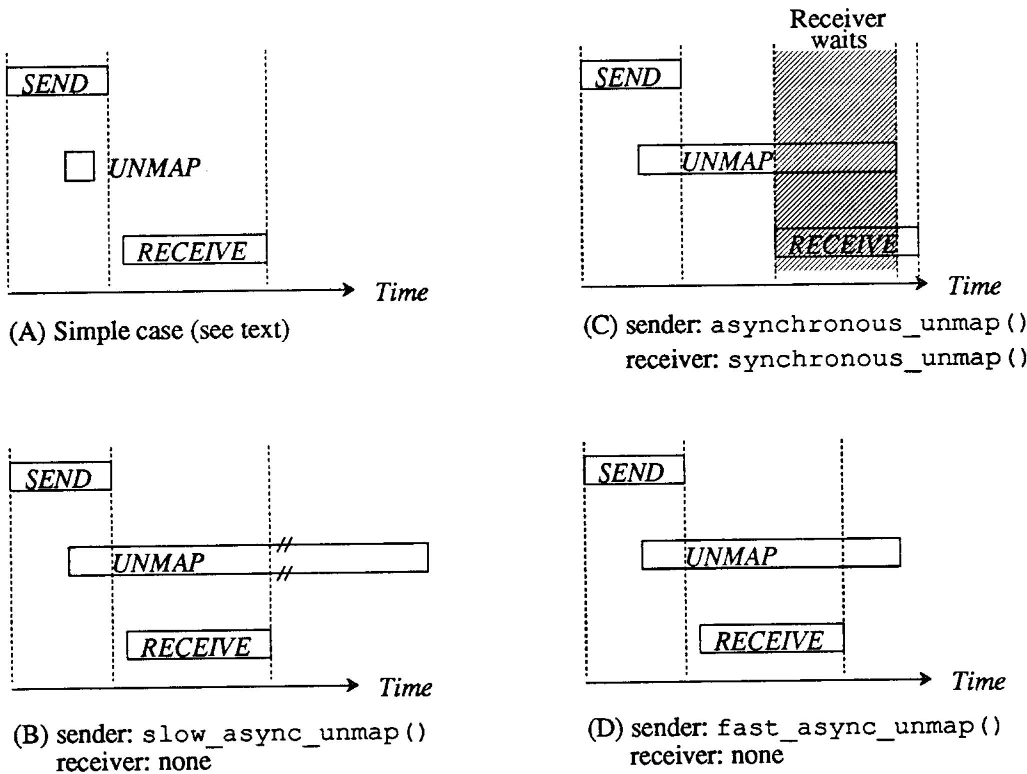
Figure 3 illustrates the possible timing relationships of `send()`, `unmap()`, and `receive()`.

There are three simple cases for unmapping:

- If a page is physically mapped into a space only on the current processor (the one where the `send()` operation is being executed), it is unmapped using `synchronous_unmap()` on the `send()` operation (Figure 3A). This can be done instantly (before the `send()` operation returns).
- If the sender is the kernel (or a universally trusted ''superuser'') then no physically unmapping is done since the sender is trusted.
- When an IPC page has multiple owners, its protection is read-only. Such a page can always be unmapped using `slow_asynchronous_unmap()` on the `send()` operation (Figure 3B). The receiver does not have to wait for the completion of unmapping.

In the general case, an IPC page may be physically mapped on processors other than the current one. This occurs if the owner process migrates between processors, and references the page from different processors. The mapping may then be present in multiple TLB's. However, this will hopefully not be a common case; process scheduling should favor keeping a process on a single processor to promote memory access locality for improved cache performance.

Suppose an IPC page is mapped on processors other than the current one, as shown in Figure 3C or 3D; on `send()` the page is unmapped using `slow_asynchronous_unmap()` or `fast_asynchronous_unmap()`. The choice is determined by the `trusted` flag of the `send()` operation. If the receiver



(A) Simple case (see text)

(C) sender: `asynchronous_unmap()`
receiver: `synchronous_unmap()`

(B) sender: `slow_async_unmap()`
receiver: none

(D) sender: `fast_async_unmap()`
receiver: none

Figure 3: Timing Diagram for `send()`, `unmap()`, and `receive()`.

does not trust the sender (Figure 3C), it has to wait until the unmapping is completed. Note that the sender may return while the unmapping is in progress; the unmapping needs to be done only before the receiver starts using the data. If the receiver trusts the sender (Figure 3D), it can use the data before the unmapping is done.

Figure 3C is the only situation where the receiver has to wait for the completion of an `unmap()` operation. The receiver waits by issuing a `synchronous_unmap()` request. The implementation of this function checks whether interprocessor requests are already active, issues them if necessary, and waits for completion. If there is a delay between the `send()` and `receive()` operations because of application or system overhead, the unmapping may have already been done when the `receive()` operation starts and the `synchronous_unmap()` is not necessary.

In unmapping for deallocation and page-out operations, the latency is not critical and the unmapping can be done asynchronously. The virtual IPC page and the physical page are not available for recycling before the unmapping is done.

3.4.1. Tolerating TLB Inconsistency

The diagrams in Figure 3 can be divided into three categories:

- The `send()` returns after the `unmap()` is done (Figure 3A).
- The `unmap()` is completed after the `send()` returns, but before the `receive()` returns (Figure 3C).
- The `unmap()` has not been completed when the `receive()` returns (Figures 3B and 3D).

In the last two cases, the page ownership rules are potentially unenforced for a certain period, allowing the sender to access an IPC pages while it does not own it. However, this will not cause any security violations. In Figure 3C, the sender may modify the data it has given away, but only before the receiver reads it. The effect is the same as if the sender modified the data first and then gave it away; once the receiver starts using the data, it cannot be changed by the sender. In Figure 3B, the sender's old protection is read-only. It cannot destroy the data and can only read data that it already knows. In Figure 3D, the inconsistency potentially allows the send to modify the IPC page after the receiver has received it. However, this case occurs only if the receiver has specified that it trusts the sender, and therefore is not a security violation.

4. RELATED WORK

The DASH design for data movement by VM remapping incorporates a number of new and old ideas in operating systems and distributed systems. Other projects have addressed some of the issues discussed here. This section surveys this work and compares it to the DASH design. Our discussion is divided into the following areas: 1) *IPC semantics*, 2) *Virtual memory abstraction*, and 3) *TLB inconsistencies*.

4.1. IPC Semantics

Accent is a message-based operating system, described briefly in Section 1 [Fit86, FiR86, RaR81]. Seeing the poor performance of the memory remapping operations cited there, the Accent developers suggested that a larger page size and implementing the VM operations in microcode could improve performance. The copy-on-write scheme for the IPC call-by-value semantics is provided by a fault handling routine in the Accent kernel. In DASH, the copy-on-write mechanism is at the user-level; no page fault is incurred when a `write` message access operation needs to make a copy of a page.

Mach is a descendant of Accent [RTY87] and inherits its call-by-value/copy-on-write semantics. There has been an attempt at improving Mach's message passing by dedicating one processor of a shared-memory multiprocessor as an IPC coprocessor [WeT86]. The coprocessor copies a message in parallel with the application processing of the sender and the receiver. It was assumed that both the sender and the receiver had other work to do while copying was in progress. However, the performance results were not encouraging and sometimes were worse [WeT86].

Both Accent and Mach use the copy-on-write mechanism for message passing as well as for address space inheritance for forked processes. This may reduce physical memory requirements. DASH does not have address space inheritance; its primary method of allowing sharing is the shared-segment region of an address space. Because it is used only for IPC, the copy-on-write mechanism in DASH can be simpler.

Accent and Mach also differ from DASH in the way they treat the IPC page in the sender's address space. They keep the page mapped, but change the protection to read-only. In DASH, the sender loses ownership of the page it sends. This is based on the assumption that a process will usually not need a copy of the pages it is sending, and if it does, it must duplicate them.

4.2. Virtual Memory Abstraction

Shared memory is an alternative to message passing for communication. It is attractive because it requires no explicit memory copying and does not rely on dynamic page remapping. Li has proposed a shared virtual memory in which a single virtual address space is shared by a set of loosely-coupled processors [Li86]. The intention is to provide the benefits of a global, single level store for a single level process accessing remote data. Such benefits do not immediately carry over to the communication between two processes, because of the synchronization necessary. His ideas are extensions of those found in the Apollo Domain system [LLD83].

Shared memory does have its problems. The abstraction was not originally proposed for remote communication. Synchronization primitives are still necessary. Communicating processes must be well-behaved and not access data they should not. In the client/server model of communication, this latter point is difficult to guarantee because clients are not always trustworthy. Li has solved some of these problems in the VM mapping manager, transparently to user programs.

DASH solves these problems by having a high level message passing abstraction and introducing protection to the shared memory model. The goal is to keep the performance benefit of shared memory while overcoming its undesirable features. The message passing system can be viewed as synchronization primitives for dynamically protected shared memory. The primitives are integrated with the VM system. Instead of depending on processes to be well-behaved, the semantics of the IPC primitives define page ownership, and the VM system enforces the ownership rule.

4.3. TLB Inconsistencies

Caches form part of the solution to TLB inconsistencies in two recent projects. SPUR and VMP have virtual address caches; the performance of address translation is less critical than that for processors having physical address caches. Both regard the TLB's as caches of page table entries and use cache coherency protocols to handle TLB inconsistencies. TLB entries and user data share a cache and complete for space. The cache coherency protocols for SPUR are implemented in hardware [Rit85, WEG86], whereas those of VMP are implemented mostly in software [CSB86].

Another approach to TLB inconsistency is to have a single TLB shared by all processors [HeH86]; no inconsistencies can arise. However, this may introduce a performance bottleneck as the number of processors increases.

With a view towards porting DASH to many different computer systems, we make minimal architectural assumptions. We assume one TLB per processor, with consistency managed by software. This is the model used by most commercial systems, e.g., VAX 8800 [FHM87]. DASH tolerates TLB inconsistencies, and handles them efficiently when they do occur.

5. SUMMARY AND FUTURE WORK

We have argued that memory copying will increasingly be a bottleneck in high performance network communication. Memory remapping can eliminate copying but has its own problems: high VM overhead and TLB inconsistency on shared-memory multiprocessors. The goal of this research is to minimize the overhead of VM remapping while addressing the TLB inconsistency problem.

Our solution is an integrated IPC design involving message-passing, protected shared memory, and low-level VM mechanisms. The message-passing system defines message ownership, which implies IPC page ownership. The sender relinquishes the ownership of the message being sent. The IPC region is shared by all spaces with the ownership rules enforced by the VM system. The expected performance benefits of our design are described in the following paragraphs.

In our message-passing model the `send()` and `receive()` operations have parameters that can be used by the implementation to reduce work. For example:

- The sender may specify whether it thinks that the receiver trusts it. If so (and if the sender is correct) an unmap operation is unnecessary.
- The receiver may specify whether it trusts the sender or not. When the sender is trusted (e.g., the kernel owner is always trusted), it is not necessary to wait for a previous asynchronous unmap operation to complete.
- The receiver may specify whether to physically map in message pages immediately, or to map them in on demand. In the second case, a page is mapped in by the page fault handler when the receiver first accesses the page. No mapping is done if the page is not accessed. This optimization may be significant for applications that forward messages (e.g., a file service that receives a block from a disk device and sends it to the network without accessing it). Message forwarding is a common communication paradigm when the operating system is organized as a set of user-level services.

The above optimizations can eliminate operations. When operations are necessary, our design allows them to be done efficiently. Synchronous unmapping is more expensive than asynchronous unmapping, and we avoid it when possible. For both types of unmapping operations, the VM system maintains the set of processors on which a page has been mapped, and only unmaps it from those processors.

In addition to the above performance advantages, the model of protected shared IPC pages for message-passing yields a design that is simpler and more easily implemented than some others. Message pages reside at the same address in all spaces, so there is no need to adjust page pointers on remapping. Messages need not be contiguous in memory, so allocation problems such as external fragmentation are avoided.

The design is now being implemented in a prototype distributed system being developed by the DASH project at UC Berkeley. Our first implementation is on a Sun 3/50 uniprocessor and we are planning to port it to a Sequent Symmetry series multiprocessor. The performance of the IPC system will be evaluated when the implementation is done, and we will evaluate different techniques for asynchronous unmapping. We will also study the interaction between our design and process scheduling of shared-memory multiprocessors, since the cost of unmapping depends on whether or not a process has migrated

between processors.

6. ACKNOWLEDGEMENTS

We thank the other members of the DASH project at UC Berkeley for useful discussions. We also thank Stuart Sechrest for his valuable comments on a draft of the paper.

Reference

[AFR87] D. P. Anderson, D. Ferrari, P. V. Rangan and S. Tzou, "The DASH Project: Issues in the Design of Very Large Distributed Systems", Technical Report No. University of California, Berkeley/CSD87/338, Computer Science Division, University of California, Berkeley, January 1987.

[CSB86] D. R. Cheriton, G. A. Slavenburg and P. D. Boyle, "Software-Controlled Caches in the VMP Multiprocessor", *Proceedings 13th Int. Symposium of Computer Architecture*, June 1986, 366-374.

[FHM87] S. J. Farnbam, M. S. Harvey and K. D. Morse, "VMS Multiprocessing on the VAX 8800 System", *Digital Technical Journal*, February 1987, 111-119.

[Fit86] R. P. Fitzgerald, *A Performance Evaluation of the Integration of Virtual Memory Management and Inter-Process Communication*, Ph.D. Dissertation, Carnegie-Mellon University, October 1986.

[FiR86] R. Fitzgerald and R. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent", *ACM Transactions on Computer Systems 4*, 2 (May 1986), 147-177.

[HeH86] J. L. Hennessy and M. A. Horowitz, "An Overview of the MIPS-X-MP Project", Technical Report STANCSL 86-300, Computer Systems Laboratory, Stanford University, April 1986.

[LLD83] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf, "The Architecture of an Integrated Local Network", *IEEE Journal on Selected Areas in Communication 1*, 5 (November 1983), 842-857.

[Li86] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Dissertation, YALEU/DCR-492, Yale University, September 1986.

[RaR81] R. Rashid and G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proceedings of the 8th Symposium on Operating System Principles*, December 1981, 64-75.

[RTY87] 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", *Proceedings of the 2nd Symposium on Architecture Support for Programming Language and Operating System*, October 1987.

[Rit85] S. A. Ritchie, ''TLB for Free: In-Cache Address Translation For a Multiprocessor Workstation'', Technical Report University of California, Berkeley/CSD85/233, Computer Science Division, University of California, Berkeley, May 1985.

[WeT86] J. W. Wendorf and H. Tokuda, ''An Interprocess Communication Processor: Exploiting OS/Application Concurrency'', Technical Report, Computer Science Department, Carnegie-Mellon University, May 1986.

[WEG86] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz and D. A. Patterson, ''An In-Cache Address Translation Mechanism'', *Proceedings 13th Intl. Symposium of Computer Architecture*, June 1986, 358-365.