References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The interaction of architecture and operating system design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[4] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation lookaside buffer consistency: A software approach. In *Third Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 113–122, Boston, Massachusetts (USA), Apr. 1989. ACM.

[5] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3), March 1988.

[6] R. Fitzgerald and R. F. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[7] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 68–80. Association for Computing Machinery SIGOPS, October 1991.

[8] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[9] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[10] J. C. Mogul. Network locality at the scale of processes. In *Proceedings of the SIGCOMM '91 Conference*, pages 273–284, Zuerich, Switzerland, Sept. 1991.

[11] S. J. Mullender, G. V. Rossum, A. S. Tanenbaum, R. V. Renesse, and H. van Staveren. Amoeba – A Distributed Operating System for the 1990's. *IEEE Computer Magazine*, May 1990.

[12] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Usenix 1990 Summer Conference*, pages 247–256, June 1990.

[13] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, Feb. 1990.

[14] *UNIX System V Streams Programmer's Guide*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

modified a version of the Mach kernel that avoids communication with the fbuf pager when a newly allocated fbuf is first accessed. This modification significantly reduces the per-page cost of accessing a newly allocated fbuf, from 580μsec to 190μsec.

The row labeled "IOData transfer" shows the fbuf-related cost of passing an IOData object to another domain. The reported number does not include the cost of the IPC call used for sending the reference, since that cost is dependent on the IPC facility used. Using Mach IPC with RPC stubs, the combined cost is 134μsecs. As can be seen, there is a fixed cost for passing an IOData object, independent of the number of constituent fbufs and their cliques.

The IOData access costs are reported in the row labeled "IOData access". There is a fixed cost for accessing a IOData object. Additionally, a cost is incurred for each fbuf in the IOData object. Finally, if the accessing domain is not a member of an fbuf's current clique, then a per-page cost is associated with referencing the pages of this fbuf. This per-page cost is caused by page faults that occur as a result of the Mach kernel's lazy update strategy for machine-specific virtual memory data structures. That is, when a domain accesses a shared page for the first time, a page fault occurs, even when the page is in main memory and another domain has previously accessed it.

As we have seen, a significant additional cost must be paid whenever an fbuf must be allocated and/or transferred to a new domain. The clique allocator's caching strategy should eliminate this cost in most cases; however, we cannot yet quantify this effect. Another problem is that there are situations when the set of domains that will be allowed access to an fbuf is not known at the time of the fbuf creation. This can happen, for example, when a device driver for a DMA device must pre-allocate an fbuf for incoming packets. Note, however, that this pre-allocation can be removed from the critical path of a data transfer.

Another cost component we cannot yet quantify is that of reclaiming physical memory mapped to deallocated fbufs. This cost does not occur in the critical path of a data transmission, and can be delayed at the cost of temporarily consuming physical memory. Since we expect traffic patterns for high-bandwidth applications to be bursty, it should be possible to perform this "cleanup" during low-traffic periods.

7 Conclusion

We have described a new facility for transferring data across domain boundaries on shared-memory machines. Our approach can be thought of as a dynamic shared buffer mechanism in as much as the set of buffers that is shared among a collection of domains changes over time based on usage. We avoid the security limitations of other shared buffer schemes by randomly selecting buffer addresses from a large virtual address window. In this way, a buffer's address can be viewed as a software capability, and by passing this capability from one domain to another, the data in the buffer is effectively transferred across domain boundaries.

Experiments show that a data transfer facility based on this approach performs significantly better than traditional copy-on-write solutions. Our analysis suggests that the reason for this better performance is that unlike existing techniques, our facility does not depend on (1) kernel involvement, or (2) fast execution of virtual memory mapping changes.

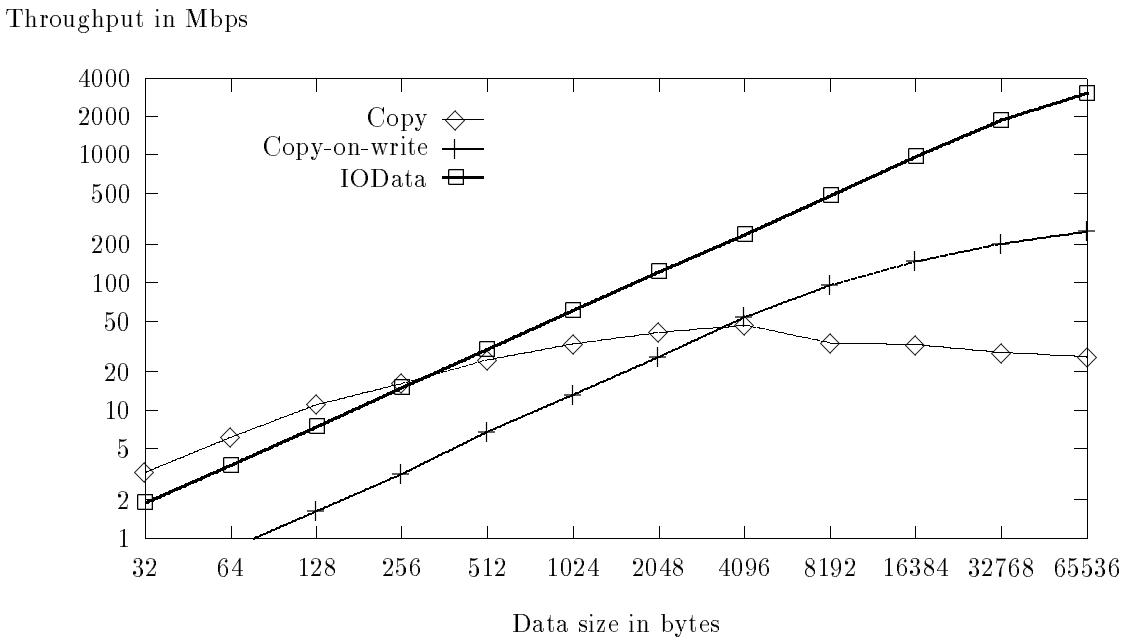

Figure 5: Mach Cross-Domain Throughput

6.2 Cost Analysis

Table 1 gives a detailed breakdown of various costs relating to the creation and transfer of IOData objects in the prototype. The row labeled "IOData creation" reports the measured cost for creating an IOData object. The fixed cost is due to the IOData object creation and initialization—it is also the cost of creating an empty IOData object. If the IOData is allocated with an initial content, then an fbuf is required which incurs a per-fbuf cost, depending on whether an fbuf can be reused or an fbuf allocation is necessary. A newly allocated fbuf does not have any physical memory mapped, which causes a per-page cost for taking a page fault upon first access. We assume that a reused fbuf still has physical memory.

		fixed cost	per-fbuf cost	per-page access penalty
IOData creation	fbuf reused	4	8	—
	fbuf allocated	4	27	580 (190)
IOData transfer		31	—	—
IOData access	within clique	2	5	—
	out of clique	2	5	143

Table 1: Basic costs of the prototype in μsecs

In handling page faults on a newly allocated fbuf, the kernel sends a message to the fbuf pager, requesting the missing data. The fbuf pager responds by instructing the kernel to provide a zero-filled page. The cost of the communication between kernel and fbuf pager acounts for almost two thirds of the cost incurred by these page faults. This is the main limitation of our layered prototype. We have

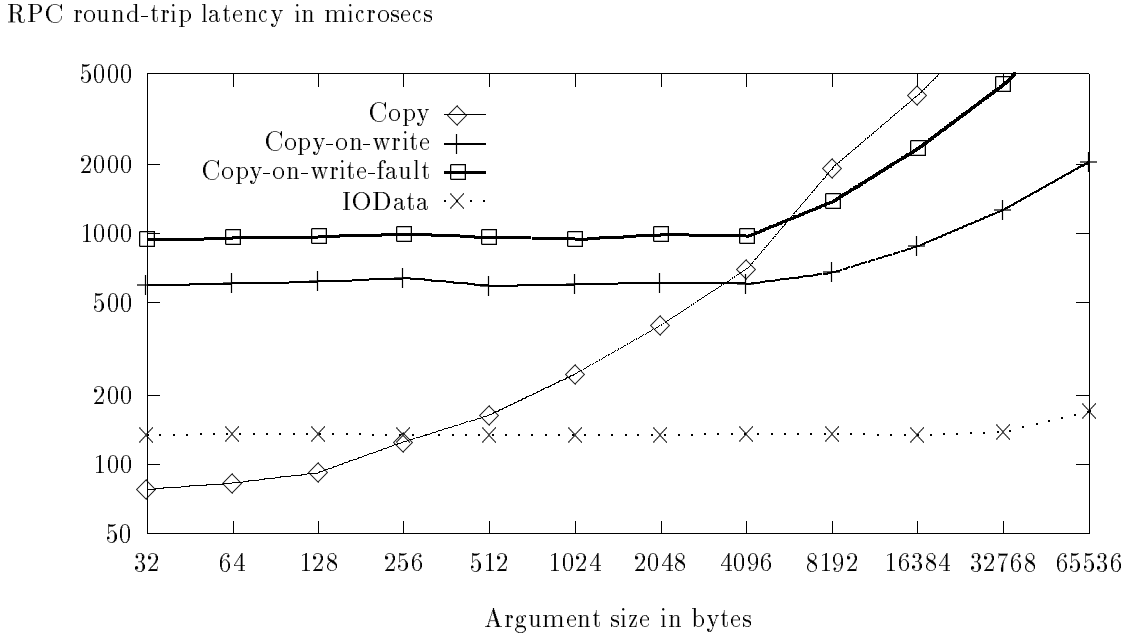

Figure 4: Mach cross-domain RPC performance

Two data copies occur in the case of a copy argument—from the caller's domain into the kernel, and from the kernel into the callee's domain. Nevertheless, due to the high basic cost of copy-on-write, passing an argument by copy is cheaper for argument sizes of up to 3,500 bytes. There is also a significant per-page cost for large copy-on-write arguments, due to page faults caused by Mach's lazy update strategy for machine-specific page tables.

The latency for calls using an IOData argument is data-size independent up to a data size of 32KBytes. The small data-dependent cost for larger data sizes is due to TLB misses, which are handled in software in the MIPS architecture. The cost of IOData-based RPC breaks even with the cost of a copy message at a data size of approximately 250 bytes. For smaller data sizes, using an IOData is more expensive due to the overhead in creating and destroying the IOData object and the associated fbuf.

Figure 5 shows the calculated throughput of each of the above tests based on the measured latency numbers. Again, both axes use a logarithmic scale. As can be seen, IOData objects result in significantly better throughput than copy-on-write—241Mbps versus 54Mbps at 4KB transfer sizes, respectively. Copy-on-write incurs a data-dependent cost due to page faults, which causes its throughput curve to level off. IOData objects support gigabit throughput rates for a data size greater than 16KB. Importantly, note that this does not imply the the constituent fbufs that implement the IOData object are themselves 16KB; the constituent fbufs can still be the same size as network packets. With copy-on-write, on the other hand, the size of a transferred data unit is limited by the network packet size, unless it is possible to combine multiple network packets into a contiguous buffer.

In other words, the proposed application interface based on IOData objects avoids the copy from the application buffer to a system buffer on the send side, and from a set of system buffers to an application buffer on the receive side. The IOData abstraction supports operations that allow copy-free processing of data in the communication subsystem. Finally, the fbuf facility, used in our implementation of the IOData abstract data type, avoids copies across domain boundaries.

Note that the proposed application interface can coexist with the traditional UNIX read/write interface, which is maintained for backward compatibility. Applications that depend on high bandwidth network communications are only now being implemented, and can just as easily be written to take advantage of the new interface.

6 Performance

This section presents some preliminary performance results on our prototype fbuf facility. All measurements were performed on a DecStation 5000/200, which uses a MIPS R3000 processor running at 25Mhz, and has a 64KB instruction cache and a 64KB write-through data cache. The system had 16MBytes of main memory and ran the Mach3.0 kernel version MK69 with the UNIX single server version UX28.

Unless otherwise stated, all measurements were taken by repeating a sequence of operations N times, measuring the elapsed time using the kernel's host_get_time() primitive, and dividing the result by N. In each experiment, N was chosen so that the elapsed time was at least one hundred times the real-time clock granularity, which is 15.625msecs. To eliminate interference from other system activities, all threads executing on behalf of the test ran at priority zero (highest), using a fixed-priority scheduling discipline.

6.1 Fbufs and Mach IPC

Figure 4 shows the data-transfer performance of Mach IPC between two user-level tasks; both axes use a logarithmic scale. RPC stubs for the calls were generated using the MiG stub generator. Run-time type checking in the RPC stubs was turned off. Each line shows the latency of a specific test on a set of data sizes ranging from 32 bytes to 64KB. The tests perform the following actions:

Copy: The caller writes one machine word per page of a pre-allocated, page-aligned buffer, and passes the buffer as a copy argument to the RPC. The callee reads one word in each page of the received argument and returns.

Copy-on-write: Like copy, except the buffer is passed as a copy-on-write argument.

Copy-on-write-fault: Like copy-on-write, except the callee *writes* one word in each page of the received buffer, forcing a physical copy of the argument.

IOData: The caller creates an IOData object consisting of one fbuf, writes one word in each page of the fbuf, and passes the IOData object as an RPC argument. In this case, an appropriate fbuf is reused, i.e., the page allocator is not invoked. The callee extracts the content of the IOData object, touches one word in each page, frees its (logical) copy of the IOData object, and returns. The caller, upon return from the RPC, destroys the IOData object. The RPC returns a hidden argument that contains fbuf deallocation notifications.

adjusting the length and offset fields. Concatenation (message reassembly) is achieved by creating a pair node that refers to the root nodes of the component trees. A cross-domain transfer of a message object is a special case of a logical copy. The reference count on the root node of the transferred tree is incremented in the sending domain, and then a cross-domain link node is created in the receiving domain, referring to that root node.

Tree nodes are deallocated when their reference count goes to zero. Nodes that contain references (i.e., non-leaf nodes) decrement the reference count on the referents upon deallocation. A special case occurs when a link-node's reference count goes to zero. Since the node's referent originates in a different domain, its reference count cannot be directly decremented. Therefore, the appropriate domain must be notified of the deallocation. In practice, multiple deallocations are "accumulated" before a notification is sent to the appropriate domain. Moreover, deallocation notifications are "piggy-backed" onto other messages whenever possible.

Note that all the details of handling fbuf trees (and in fact the whole concept of fbufs) is entirely hidden inside the implementation of the IOData abstract data type. Programmers exclusively create, destroy, manipulate, and pass IOData objects as arguments in IPC; they do not directly manipulate fbufs.

5.2 Application Interface

Although IOData objects could be hidden below a traditional application interface (like the UNIX read/write interface), in the interest of avoiding *all* data copies, we propose an alternative programming interface that exposes the IOData abstraction to the application. The problem with the UNIX read/write interface is that (1) it has copy semantics, and (2) it requires data to be stored in a contiguous buffer. During a write system call, data must either be copied, or the system call must not be allowed to complete until the system is done with the data, which may take as long as a network round-trip time. A read also requires data to be copied, since the application specifies the address of a contiguous buffer in its address space.

We propose a write operation that takes an IOData object as an argument. The application creates an IOData object, filling it with data as it is produced, and then writes it, causing a logical copy of the data. The application continues to hold a reference to the data, but it may not change the contents of the object since it is immutable.[3] Once the last reference is relinquished, in both the originating and in other domains, the IOData object is destroyed.

Our proposed read operation returns a IOData object, obtained through a logical copy of the IOData object created by the system to contain incoming network packets. Recall that the data contained in an IOData object may not be contiguous. The application program can choose to "consume" the data in the IOData object in application specific data units; e.g., one field, record, or line at a time. The only time data needs to be copied is when a data unit spans multiple non-contiguous components (fbufs in our implementation) in the IOData object, and even then, only that specific data unit need be copied. This copying is transparent to the application, except for performance. The application program destroys the IOData object once it has consumed the data it wants.

[3] An incorrect or malicious application can modify the data content since the underlying fbuf is writable in its originating domain. However, the correctness of other domains is not affected because of the volatility assumption of fbufs.

mapped to deallocated fbufs. However, physical memory mapped to deallocated fbufs could be reclaimed more promptly if the systems page replacement algorithm were extended to use this information.

Finally, fbuf performance could benefit if the virtual memory system eagerly updated the page tables of all sharing domains for fbuf pages on architectures where this can be done efficiently. This could avoid the access faults that occur when an fbuf is passed to a new domain for the first time.

5 Using Fbufs

Fbufs provide a primitive facility for transferring data across protection boundaries. This facility can be used as a building block for constructing higher-level abstractions. This section describes such an abstraction, one that is tuned for copy-free communication processing. It also suggests an alternative application interface based on this abstraction.

5.1 IOData Object

Communication processing is more complicated than moving buffers across protection domains. Even within a single domain one needs to be careful about avoiding data copies. For example, a single message may be fragmented across multiple network packets. Since packets arrive asynchronously from the network, the receiving host is usually forced to receive each packet in a distinct buffer; in our case, a distinct fbuf. Rather than copy the fragments into a single contiguous fbuf, a message might be logically constructed from a collection of fbufs. We define an abstraction, called a *IOData object*, that supports this kind of buffer manipulation, and show how it can be implemented using fbufs.

The IOData abstraction is based on *x*-kernel *messages* [8] (and is similar to BSD *mbuf lists* [9] and System V *stream messages* [14]), except that it supports logical copying across protection domains. IOData objects are *immutable*, that is, they are created with an initial data content and cannot be subsequently changed. They support operations for logical copying, concatenation, splitting, and clipping. Additionally, there are special operations for the efficient addition and removal of network packet headers.

We have used fbufs to implemented the IOData object abstraction. Note that fbufs are writable only in their originating domain, which is consistent with the immutability of the IOData abstraction. This permits an implementation of logical copying—both intra- and inter-domain—that simply increments a reference count.

An IOData object is implemented as a *reference* to a binary tree. IOData objects are stored within a domain's private address space, e.g., its stack or heap. The tree itself resides entirely within the fbuf window, and consists of pair nodes, cross-domain link nodes, and leaf nodes. All nodes have a reference count, offset, and length fields. A pair node contains two references—for the left and right child, respectively. A link node has a single reference to a node that originates in a different domain. There are two kinds of leaf nodes: one consists of a single page containing both the node and the data area, and the other contains a pointer to a separate fbuf with an integral number of contiguous pages. A pre-order walk of the tree yields the data chunks that are logically contained in the IOData object.

When an IOData object is copied, the reference count in the root node of the referent tree is incremented. Splitting (message fragmentation) is implemented by a combination of logical copying and

pages can be flushed and reclaimed. It could inform the fbuf pager of the deallocation, but this would add the cost of a cross-domain IPC call to the cost of an fbuf deallocation. The kernel's page-out daemon would eventually reclaim the physical memory mapped to deallocated fbufs. However, since these pages were referenced immediately prior to their deallocation, the system's replacement policy will not select them for some time. During this time, other used pages may be selected for replacement at a cost to the system's performance.

To solve this problem, the fbuf pager periodically monitors the number of pages belonging to deallocated fbufs in each domain. If this number—reduced by the number of pages that were paged out by the kernel—is too high, the fbuf pager instructs the kernel to flush ranges of virtual memory that do not contain any allocated fbufs. The fbuf pager has knowledge only of the address of allocated fbufs, not of deallocated fbufs, and therefore does not know the number and size of deallocated fbufs in an address range. However, since fbufs are allocated uniformly in their address window, the fbuf pager can apply a statistical estimate for the number of "discardable" pages in a virtual address range of a given size, given the number of such pages in a domain's writable subwindow.

4.2 Optimizations

The prototype implementation just described is fully functional. We report on its performance in Section 6. This section explores how performance could benefit from an implementation that is more tightly integrated with the kernel's virtual memory system rather than layered on top of an existing system.

The obvious target for optimization is the allocation of new fbufs since this is the operation that requires involvement of the virtual memory system, namely the allocation of physical memory. The page fault resulting from the first access of a new fbuf causes the kernel to send a message to the fbuf pager, requesting the missing page. The fbuf pager replies with a message notifying the kernel that no data is available. Finally, the kernel zero-fills and allocates a physical page. The messages exchanged between the kernel and fbuf pager are asynchronous and are thus not subject to the optimizations that the Mach kernel applies to synchronous IPC (RPC) messages. Consequently, the cost of an fbuf allocation is dominated by the page fault handling and is quite high. Linking the fbuf pager into the kernel domain would not significantly improve this situation, since all communication between kernel and fbuf pager would still occur via Mach IPC.

A significant improvement can be achieved by adding support to the VM system so that the memory object backing the fbuf window is handled as an object with special semantics. A very effective optimization is to have the kernel, upon a fault within the fbuf memory object, check the allocation status of the page before sending a request message to the fbuf pager. In particular, a request message to the fbuf pager is only necessary if the non-resident page is both allocated and currently paged out. We have made this modification to the Mach kernel, and report the resulting performance improvement in Section 6.

Another possible optimization concerns physical memory management for the fbuf memory object. Currently, the kernel's page replacement algorithm does not explore the available information on the allocation status of physical pages mapped to fbufs. Replacement decisions are instead based on a global LRU approximation. The fbuf pager uses a flushing heuristic to prevent the waste of memory resources

mapped into address spaces. When a page is referenced that is not in the cache, it contacts the appropriate memory manager to obtain the data. When the kernel decides to evict a modified page from the cache, it sends the data to the appropriate memory manager for disposal. There is a default memory manager that provides initially zero-filled pages and stores paged-out data in a special file or disk partition. It implements the semantics commonly expected from generic virtual memory.

The Mach kernel allows external memory managers to be added to the system. Such a memory manager can implement abstract memory objects with specialized semantics. We use this feature to implement the special semantics of the fbuf window based on the standard Mach virtual memory system.

The semantics of the virtual memory associated with the fbuf window is unusual in two important respects. First, the physical pages associated with a deallocated fbuf can be flushed from the main memory cache, although the associated virtual address range has not been deallocated. In other words, the pages can be unmapped from the shared window and their contents can be discarded. Moreover, if the kernel decides to page out an allocated, but currently unused fbuf, its data can also be discarded. Second, referencing an address within the fbuf window that does not refer to an allocated fbuf causes an exception to be raised in the offending thread.

Our external memory manager, called the *fbuf pager*, maps the fbuf window that it backs into its own address space. The domains' page allocators place the data structures that describe their set of allocated pages into an fbuf, and notify the fbuf pager of its address. The fbuf pager therefore has knowledge of the allocation status of each page within the fbuf window. Additionally, the fbuf pager has access to information about the number of previously used/deallocated pages in each domain's writable subwindow.

When the fbuf pager receives a request from the kernel for pages that are not currently resident in main memory, it looks up the allocation status of the pages. If the pages belong to allocated fbufs and the fbuf pager has no data for them (i.e., the fbuf was newly allocated), the pager instructs the kernel to provide zero-filled pages. If the pages belong to allocated fbufs and the fbuf pager has data for them (i.e., they were previously paged out by the kernel), then the appropriate data is provided to the kernel. Finally, if the requested pages do not belong to an allocated fbuf, then the kernel is instructed to raise a memory failure exception in the offending thread. A kernel request to dispose of a page that the kernel has decided to page out causes the fbuf pager to either discard the page (if it belongs to a free or deallocated fbuf), or have the default pager write it to secondary paging storage.

Mach's virtual memory system is layered into machine-dependent and machine-independent portions. The machine-independent portion reflects the state of the VM system at all times. The machine-specific state (e.g., the hardware page tables) is updated in a lazily evaluated fashion. Specifically, whenever the protection of a virtual memory region is lowered, the machine-specific page tables are not updated until a page fault occurs. If the requested access is found to be legal, the page tables are updated accordingly for the faulted page. In the case of the fbuf facility, this lazy evaluation strategy leads to additional page faults when an fbuf is passed to a domain that has never before accessed that fbuf. However, the resulting cost is mitigated by the effects of clique caching. That is, whenever an fbuf can be reused in the exact clique requested, no page faults occur.

Remember that the clique allocator in each domain deallocates pages (at the level of the page allocator) if it has too many cached fbufs. The page allocator removes the corresponding pages from the set of allocated pages, but has no means to inform the kernel that the physical memory associated with these

an application domain. The originator of such packets is the kernel domain, which is implicitly trusted by all other domains. Thus, no problem arises for received network data.

Now consider buffers that contain application-generated data, that is, fbufs originating in a user domain. The application domain may pass the buffer to the kernel, possibly through a network server. Note, however, that network and OS software rarely *interprets* user data. If some processing of the user data is required, like checksum calculations or encryption, the data-transforming function can be designed to not depend on the stability of the data. We say "not depend" in the sense that a sporadic change to the data should not lead to the failure (crash) of the communication subsystem; incorrect output data only affects the malicious/incorrect user that modified its data after sending it.

4 Prototype Implementation

A prototype of our fast buffer facility has been implemented on top of the Mach 3.0 microkernel. We have chosen this platform for several reasons: to ascertain whether fbufs can be implemented in a portable way using only the standard virtual memory and IPC services provided by a modern microkernel; to evaluate the performance of an fbuf implementation based on these standard services, thereby giving us a lower bound on the performance of a more tightly integrated implementation; and to permit a more expedient implementation. This section briefly describes the prototype implementation, and suggests some optimizations that would be possible in a more tightly integrated implementation.

Since workstation-type machines that support 64-bit address spaces are not yet available, we implemented the prototype on a DecStation 5000/200. The DecStation's MIPS R3000 processor supports 32-bit virtual addresses, of which only 2^{31} bytes are available per domain. This is unfortunately not quite enough for a practical implementation of the fbuf facility: either the fbuf window is partitioned into a small number of writable subwindows (meaning a limited number of domains can be supported), or each writable subwindow is of a limited size (meaning that it is easier to guess a valid fbuf address). While these limitations are of a practical concern, they do not affect our ability to implement and evaluate the performance of a prototype.

4.1 Using Mach Virtual Memory

A virtual address space in Mach consists of a sparse set of regions. A region is a contiguous range of virtual addresses, and is associated with, and backed by, an abstract memory object. The semantics of a region—as seen by a program that accesses virtual memory within the region—is controlled by a memory manager, which implements abstract memory objects. An abstract memory object represents the non-resident state of the memory region it backs. As an example, imagine a memory manager that implements disk files as abstract memory objects. Using this memory manager, an application can map a region backed by a disk file into its address space. The semantics, as determined by the memory manager, may be as follows. Upon reference of a page, the appropriate data is read from the disk file and mapped into the region. Modified pages are eventually written back to the disk file. This example memory manager effectively provides memory-mapped disk files.

The Mach kernel uses main memory as a page cache for the various abstract memory objects currently

extremely small.

Consider the following. If W_{wr} is the size of a domain's writable subwindow, and W_{alloc} is the number of bytes allocated to fbufs, the probability of success of a single "trial" access within the subwindow is

$$p_{succ} = \frac{W_{alloc}}{W_{wr}}$$

For example, assuming 8 Mbytes worth of allocated fbufs in a domain, a writable subwindow of size 2^{50} bytes results in $p_{succ} = 2^{-27} < 10^{-8}$. Allowing a maximum of 1024 domains, each with a writable subwindow of size 2^{50}, the size of the entire shared fbuf window is 2^{60} bytes, which takes up only $\frac{1}{16}$th of a 64-bit address space.

A "wrong guess"—i.e., an access to an unallocated page of the shared fbuf window—results in an access fault, to which the system reacts by raising an exception in the offending domain. On a fast machine, the system must delay this exception long enough to make it infeasible for a user program to find an allocated fbuf using trial and error within a reasonable amount of time[2]. In particular, assuming that a domain starts a linear search by trying to access each consecutive page, starting at a random point within a domain's subwindow, the expected number of trials to find an allocated buffer is given by

$$E[trials] = \frac{W_{wr}}{2W_{alloc}}$$

Using the numbers from the above example, we get $E[trials] = 2^{26}$. Assuming that each trial takes only 100 milliseconds—i.e., the system delays the delivery of the exception for 100msec—we obtain an expected time to find an allocated fbuf of 77 days! Note that delaying an access fault exception for 100msecs does not interfere with the ability to debug a faulty program, which is the only "legal" use of this exception we can think of.

While we recognize objections to a communication system that is only probabilistically secure (rather than provably secure), it is important to understand that the proposed facility is designed to accommodate data delivered to a workstation by a high-speed network. This data has already been vulnerable to unauthorized reading while on the network. Also note that the software capabilities used by the fbuf facility are similar to the capabilities used in systems like Amoeba [11]. Finally, should stronger security be required, more traditional IPC mechanisms are likely to still be available.

3.4 Protection

The protection problem with shared memory buffers is that of preventing separate domains from unintentionally or maliciously interfering with each other. Because fbufs are only writable in their originating domain, interference of a receiving domain with the originating domain is ruled out. However, the originating domain can potentially interfere with a receiving domain. In general, a receiving domain must protect itself from such interference by assuming that the data contained in a received fbuf is *volatile*.

Fortunately, this is of limited practical consequence. Consider how fbufs might be used. Arriving network packets are placed into an fbuf, perhaps handed to a network server, and eventually delivered to

[2]Actually, the system must limit the rate at which this exception is delivered throughout the system, in order to prevent multiple domains from conspiring to find an fbuf

to choose a previously unused address. If an fbuf is deallocated at this level, its physical memory can be reclaimed.

The page allocator maintains the set of pages occupied by allocated fbufs within a domain's writable subwindow. When asked to allocate a new fbuf, the page allocator draws a random address A within the writable subwindow. Let L be the size of the requested buffer (corresponding to an integral number of pages). If the range of addresses A to $A + L$ overlaps an already allocated page, then A is rejected, and a new random address is drawn. When an fbuf is deallocated at the level of the page allocator, the corresponding pages are removed from the set of allocated pages. Any physical memory associated with the deallocated buffer can be reclaimed; whether the page allocator takes explicit action to do so is implementation specific.

The page allocator is sufficient for a correct implementation of fbufs. However, using a new fbuf allocated by the page allocator always causes a fault when the buffer is first accessed. An obvious optimization is to reuse fbufs, but naive reuse is not possible since the address of a previously used buffer may have been revealed to other domains, causing a security breach. The solution is to keep track of the set of domains that have knowledge of a particular fbuf's address, i.e., hold a capability for the buffer. We call such a set of domains a *clique*. An fbuf is said to *belong* to a clique if its address is known to exactly the members of that clique. Because of spacial locality in interprocess communication, it is likely that an fbuf that was sent to a certain set of domains can be reused for data that travels the same path.

Thus, the second-level allocator, called the *clique allocator*, avoids physical memory reallocation by exploiting spacial locality in interprocess communication to reuse fbufs. The clique allocator maintains a cache of previously used buffers according to their cliques. It implements a policy to decide when buffers should be flushed from the cache and deallocated at the level of the page allocator. This policy must ensure that (1) enough fbufs are cached for each clique in order for the cache to be effective, and (2) not too much physical memory is being claimed by currently unused fbufs. Individual fbufs are deallocated when there are too many on a free-list. Additionally, all buffers of a particular clique may be deallocated, if fbufs in that clique have not been used recently. Information from the virtual memory system about physical memory availability may also be used in the decision, if that information is available.

A program that requests a buffer may specify a clique, i.e., a set of domains that will be allowed access to the buffer. The default clique is the one containing only the current domain. The clique allocator can honor the request by reusing an fbuf of the requested clique, or any subset thereof. Only if no appropriate buffer can be found is the page allocator invoked to allocate a new fbuf. In other words, the page allocator is the "freelist" of last resort; it produces buffers that belong to the clique containing only the current domain.

3.3 Security

The security problem posed by a globally shared region—i.e., how to prevent access of an unrelated domain to a page whose content should not be disclosed—is addressed as follows. Individual fbufs are allocated sparsely at uniformly distributed, random addresses. Given a large enough virtual address window, the probability that an unrelated domain can "guess" the address of an allocated buffer is

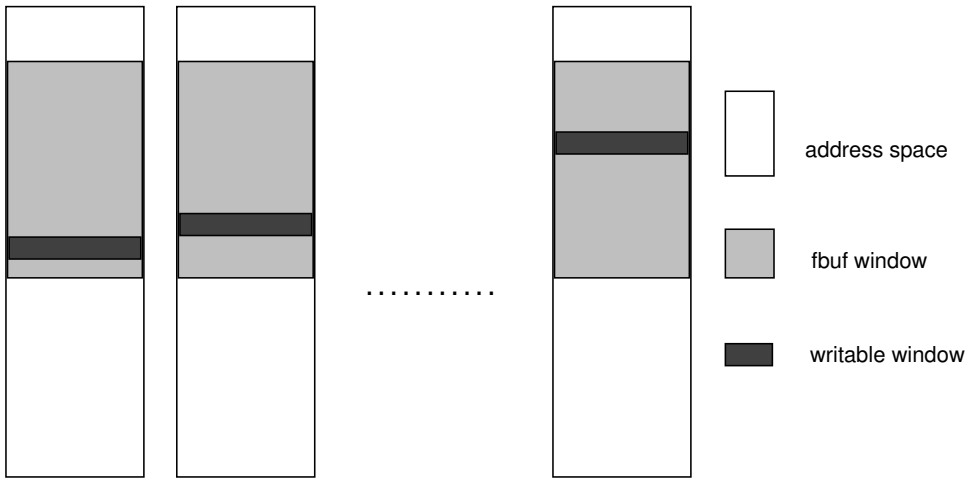

Figure 3: Fbuf Window

Each domain allocates fbufs within its domain's writable region of the fbuf window. Thus, buffer allocation in separate domains is independent and does not need to be synchronized. This benefits performance, particularly on shared memory multiprocessors.

The access fault following the allocation of a new fbuf can frequently be avoided by exploiting spacial locality in interprocess communication [10]. When an fbuf is freed, it is put on a list of free fbufs associated with the set of domains that have knowledge of its address (i.e., hold its capability). When a program requests an fbuf, it specifies the set of domains that will be allowed access to the buffer. This set typically includes domains that are either trusted, or will eventually receive the fbuf. The allocator can then return any free fbuf that is known to any subset of the specified set of domains. Fbufs reused in this way generally already have physical memory associated with them, and can thus be used without generating a fault. Allocation of a new fbuf is necessary only when all the appropriate free-lists are empty.

Fbufs act like group-wise shared buffers, except that the group of sharing domains can be extended simply by passing the buffer's capability to a new domain. One can therefore view fbufs as *dynamically shared buffers*. Virtual-to-physical memory map manipulations occur only when (1) a new fbuf is created, which has as yet no physical memory mapped, and (2) when the kernel decides to reclaim physical memory associated with an unused fbuf. Note that (1) occurs only when no fbuf can be reused, and (2) never happens in the critical path of a data transfer. Thus, allocating, transferring, and deallocating an fbuf requires no kernel involvement in the common case. A page fault and subsequent allocation of physical memory is required only when a new fbuf allocation is necessary.

3.2 Allocator and Cliques

The fbuf facility has a two-level allocator in each domain. At the base is a *page allocator*; it allocates a new fbuf that is not known to any domain. The page allocator uses a uniform random number generator

since determining this set of domains may require the inspection of several layers of protocol headers.

3 Design

In order to scale well with increases in processor performance, the inter-domain data transfer facility must avoid operations that are unlikely to scale with the overall performance of future processors. These operations include:

- data copying operations that depend on memory bandwidth while exposing little locality of reference;
- virtual memory map manipulations, especially ones that entail TLB and virtual cache coherence/consistency actions; and
- other operations that require a switch to supervisor mode.

The shared memory approach shows the most promise in this regard, since it depends on neither memory bandwidth, nor any privileged operations. It is, however, burdened with protection and security limitations. This section describes a new facility that offers the performance of the shared memory paradigm while retaining the flexibility, protection, and security of the message-passing paradigm. This facility, called *fast buffers* (fbufs), takes advantage of a very large virtual address space, a feature provided by next-generation processor architectures.

3.1 Fbuf Window

An fbuf is a contiguous memory buffer allocated from a large virtual address space window. This window, called the *fbuf window*, is mapped into every domain of a single shared memory machine at the same virtual address. Each domain has read and write permissions to a distinct subwindow, called the domain's *writable subwindow*, and read-only access to the remaining portions of the fbuf window. That is, any page within the fbuf window is mapped for read/write access in exactly one domain, and read-only in all other domains. Figure 3 illustrates the fbuf window in a system with a separate address space per domain.

Each domain allocates individual fbufs exclusively within its writable subwindow. Once allocated, an fbuf is "transferred" to another domain by sending its address to the recipient domain. Fbufs are allocated sparsely at uniformly distributed, random addresses within a domain's writable subwindow. Given a sufficiently large fbuf window, it is infeasible for an unrelated domain to "guess" the address of an allocated buffer. Thus, an fbuf's address is in effect a *software capability*.

Initially, no physical memory is allocated to the fbuf window. When a domain accesses a newly allocated fbuf for the first time, a fault occurs and the system allocates physical memory to the buffer. The data contained in an fbuf can be sent to another domain simply by passing the fbuf's capability (i.e., essentially its address). Importantly, this transfer takes place without direct kernel intervention. The receiving domain can directly access the buffer, and pass its capability on to other domains. Note that only the originating domain has write access to the fbuf; all other domains have read access only.

and TLB consistency actions on a multiprocessor [4]. If the architecture uses a virtually addressed cache, cache flushes and cache consistency actions may also be required. Third, the kernel must find a hole of the appropriate size in the receiving domain's address space. In general, it may not be possible to map the shared pages at the same virtual address in each domain. This leads to virtual address aliasing, which complicates invalidation and consistency requirements for TLBs and virtually addressed caches.

Moreover, kernel traps, page faults, TLB operations, and cache operations are like data copying in that they have not scaled with the dramatic integer performance improvements on recent RISC architectures [2]. This leads us to believe that operating systems may have to be less aggressive in the use of facilities that rely on fast execution of these operations.

Page remapping is another way to avoid data copying using virtual memory operations. For example, the V System uses this technique [5]. Although potentially more efficient than copy-on-write, page remapping is not as general: it does not preserve copy semantics, since a transfer implicitly deallocates the data from the sending domain. More importantly, page remapping suffers the same fundamental limitations as copy-on-write: (1) it requires kernel involvement, and (2) it relies on fast execution of virtual memory operations.

2.3 Shared Memory Buffers

Another copy-avoiding technique for cross-domain data transfer is the use of buffers shared between the communicating domains—e.g., kernel and user domains. This technique is used, for example, in the TAOS operating system [13]. The design space for shared buffers has several dimensions: size of the shared memory window, pair-wise/group-wise/global sharing, static/dynamic allocation, and read-write/read sharing with one writer.

Ideally, shared buffers could avoid both data copying and virtual memory map manipulations. On the sending side, an application produces data and places it into a shared buffer, hands the buffer off to the operating system, and the network interface driver transmits directly from the buffer. On the receiving end, the driver stores data into a shared buffer and hands it off to an application, which directly references the buffer until it has consumed the data. In practice, however, several problems prevent this ideal scenario.

First, a read-write shared memory window poses a protection problem because an incorrect or malicious domain can interfere with the operation of another domain that shares the window. This problem can be solved by using a small shared window, which limits and localizes the amount of vulnerable data. A small shared window, however, makes shared buffers a scarce resource. This forces an application that cannot consume received data immediately to copy the data out of the shared buffer. Similarly, the shortage of shared buffers forces an interface driver that must retain the data for retransmission to copy the data into another buffer. Thus, determining the size of a shared memory window involves a tradeoff between protection and performance.

Second, if buffers are shared globally, then a security problem arises because all domains can access data stored in shared buffers. If, on the other hand, buffers are shared pair-wise or group-wise, it is necessary to place an incoming network packet into a buffer that is shared among the precise set of domains that will eventually receive the data contained in the packet. This is difficult to do, in general,

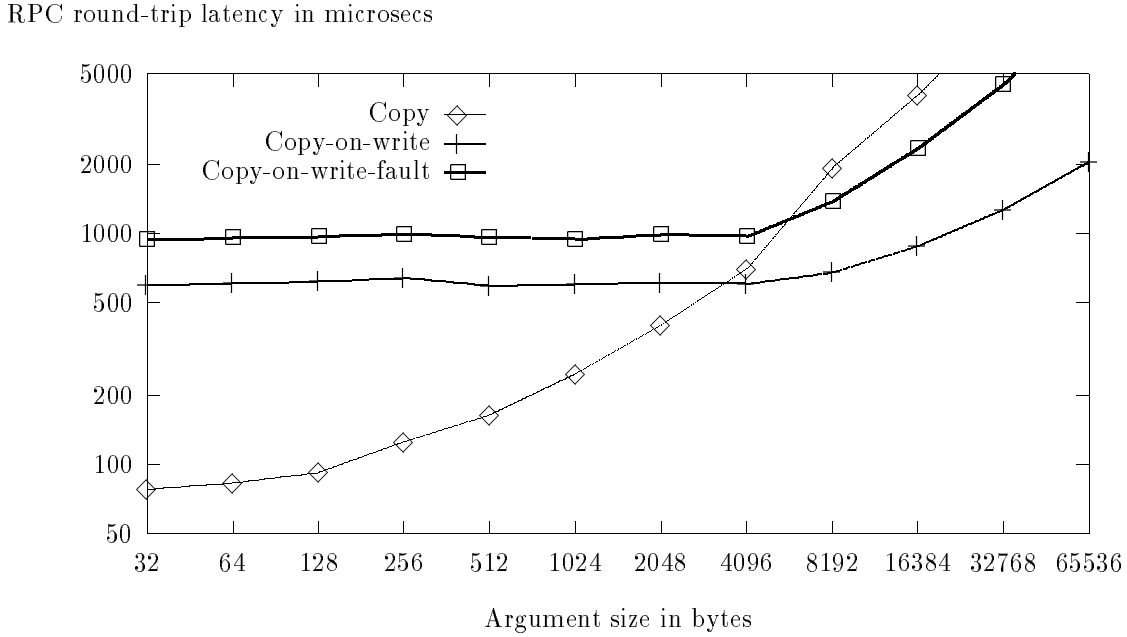

Figure 2: Mach 3.0 Cross-Domain RPC Performance

into each page of the received argument buffer, forcing a physical copy of the argument.

Unlike our discussion of copying, which focused on bandwidth, we are here reporting latency numbers. The reason is that the cost of copy-on-write data transfer is non-linear in the size of the transferred data. There is a basic cost per contiguous buffer, plus a per-page cost due to page faults when a buffer is accessed. Therefore, throughput depends on the size of the individual data units that are virtually copied. The effective throughput achievable with copy-on-write increases with the size of the data units. Using the measured latencies, the throughput is 54Mbps using 4KB data units and 255Mbps for 64KB data units.

While one would like to do copy-on-write transfers using as large of a data unit as possible, this is not always possible. On the receiving side of a network connection, for example, each data unit is likely to contain a single network packet. That is, because of the asynchronous nature of the network, each incoming packet must (in general) be received in a different buffer. Unless one is willing to pay the cost to copy these individual packets/buffers into a single large buffer, this means that the copy-on-write transfer unit is likely to be closer to 4KB than 64BK. Thus, the effective throughput rate using copy-on-write is is probably in the neighborhood of 54Mbps.

Admittedly, Mach's implementation of copy-on-write may not be optimal for the purpose of transferring network data across protection domains; an implementation of copy-on-write that is optimized for short-term sharing of individual virtual memory pages could improve performance. However, certain basic costs fundamentally limit the performance of any copy-on-write facility. First, since memory map changes require privileged operations, a kernel trap is necessary, even if a user-level facility such as URPC is used for control transfer. Second, the protection of the transferred data pages must be raised to read-only in the sending domain, which entails potentially expensive operations, such as TLB flushes,

Consider, for example, a DecStation 5000/200 connected to the Aurora gigabit network, which has a network bandwidth of 622Mbps. The DecStation's TURBOchannel bus supports DMA transfers at 800Mbps, allowing the network interface to transfer data into and out of main memory at network bandwidth. The system's memory-to-memory copy bandwidth for data not found in the cache is approximately 100Mbps, and 320Mbps for cached data. Using the above formula, the application sees an effective bandwidth of 100Mbps (16% of the available network bandwidth) in case (a), and 61.5Mbps (10% of the available network bandwidth) in case (b).

A second factor that influences user-to-user throughput is that data may need to be moved from one buffer to another so that some function can be performed on it; e.g., checksum, encryption, compression, presentation formatting. Even though the need to perform these functions reduces the effective user-to-user throughput rate, it is still beneficial to avoid as many copies as possible. It is also likely that many of these data manipulation functions will be provided in hardware on future high-speed network adapters.

2.2 Virtual Copying

Several recent operating systems support virtual copying of data across domain boundaries. Virtual copying using copy-on-write is a technique introduced by Accent [6] and used extensively in Mach [1]. The idea behind copy-on-write is to share pages that contain the virtually copied data by mapping them into multiple domains with read-only access permissions. A physical copy occurs only if a sharing domain tries to modify data while that data is still mapped in another domain. This technique can be used to delay and often avoid physical copying of data across domain boundaries while preserving copy semantics. The Mach system's interprocess communication (IPC) facility allows data to be sent both using copy-on-write and copying.

Measurements on a DecStation 5000 running Mach 3.0 (MK69) show that the high basic cost of sending data copy-on-write makes it cheaper to send data using copying for data sizes up to approximately 3500 bytes. The apparent reason for this surprisingly high cost is that Mach's copy-on-write facility was designed for long-term sharing of large virtual memory regions, as it occurs in the implementation of the UNIX fork primitive. Furthermore, Mach maintains the state of its virtual memory in a machine-independent form, and lazily updates machine-dependent page tables. This leads to additional page faults in the case of copy-on-write data transfer.

Figure 2 shows the performance of copy versus copy-on-write data transfers for various buffer sizes. Our measurements were obtained by performing a cross-domain RPC between two user tasks using MiG generated stubs.[1] The RPC function takes a single data argument of variable size, and returns no results. The basic cost for a null RPC that takes no arguments and returns no results is 59 μsecs. The line labeled "Copy" shows the latency for calls that copy the argument. Note the the argument is actually copied twice: from sending domain to a kernel buffer, and into the receiving domain. The latency for calls that pass the argument "Copy-on-write" was measured as follows: the sending domain repeatedly writes one word in each page of a (page-aligned) buffer and passes the buffer as an argument to the RPC, then the server function touches one word in each page of the received argument buffer and returns. The line labeled "Copy-on-write-fault" shows the latency for the case where the receiving domain writes one word

[1] Section 6 gives further details about how the measurements were performed.

but also to identify the issues guiding the design of the proposed fbuf facility.

2.1 Data Copying

In many existing operating systems, such as UNIX, I/O data is copied across the user-kernel boundary. Because I/O bandwidth used to be limited by storage devices and network links, copying costs did not add a significant overhead. However, recent technological trends have changed this situation. The bandwidth of high-speed networks is approaching the memory bandwidth of modern computer systems. Despite dramatic increases in processor performance, this makes the cost of even a single data copy significant. This is because data copying depends on memory bandwidth, which has not enjoyed similar improvements [12].

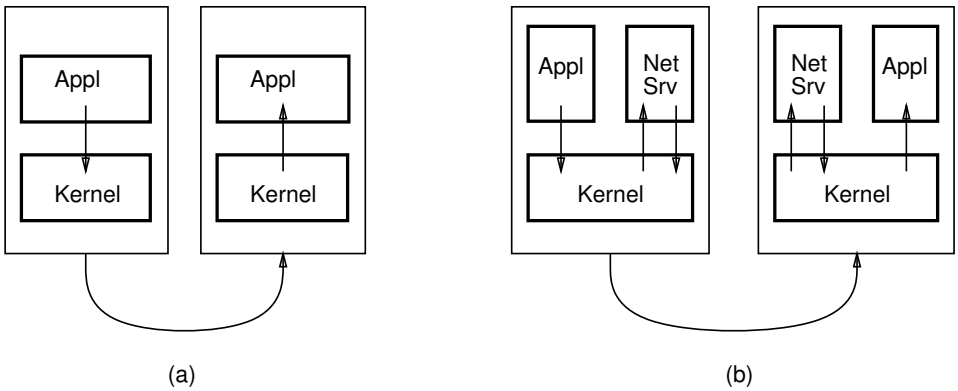

Figure 1: Simple Model of Domain Crossings

To understand the impact data copying has on end-to-end throughput, consider the simple model illustrated in Figure 1. On an monolithic operating system, one domain boundary must be crossed on each host (a). Similarly, on a small-kernel based operating system with a user-level network server, three boundary crossings are required on each machine (b). We are assuming that data arrives from the network and is placed in memory at the full network bandwidth without consuming CPU or interconnect resources. Furthermore, all protocol processing is ignored; i.e., this is a best-case analysis of achievable user-to-user throughput using copying.

In quantifying a machine's memory-to-memory copy rate, we have to consider cache effects. Generously assuming that all but the first copy on each machine is of cached data, the effective throughput rate X (up to the available network bandwidth) is given by

$$X = 1 \Big/ \left(\frac{1}{B_u} + \frac{n-1}{B_c} \right)$$

where n is the number of copies, B_u is the memory-to-memory copy bandwidth for uncached data, and B_c is the memory-to-memory copy bandwidth for cached data. Note that when $n = 1$ (i.e., a single copy on a monolithic system), the effective throughput is equal to the uncached copy bandwidth.

1 Introduction

One of the most critical issues in realizing the next generation of workstation applications (e.g., multimedia) is the design of the communication subsystem. We see the communication subsystem at the intersection of three important technology trends:

High-Speed Networks: Emerging network technology will soon offer sustained data rates approaching one gigabit per second to the workstation.

Host Architectures: Limited memory bandwidth and the complexity involved in managing the memory map hardware are making the memory architecture a dominant factor in workstation performance.

Operating Systems: The trend towards microkernel-based design leads to a system with the network devices, the communication protocols, and application programs being spread over multiple protection domains.

The challenge faced by the communication subsystem is moving data across multiple protection domains efficiently enough so that applications can take advantage of the bandwidth offered by the network. In other words, we are interested in turning good host-to-host performance into good user-to-user performance. This task is difficult because of the limitations of the memory architecture. Existing techniques for cross-domain data transfer—copying, copy-on-write, page remapping, and statically shared virtual memory—all suffer performance problems and/or other limitations.

We introduce a new facility for cross-domain data transfer, called *fast buffers* (fbufs), that is designed to take advantage of the performance characteristics of RISC architectures. The fbuf facility requires no special hardware other than support for large, sparse virtual address spaces, e.g., 64-bit addresses. Fbufs are largely independent of the IPC mechanism used for control transfer, and because they do not require the execution of any privileged instructions, they can be used with user-level IPC mechanisms such as URPC and memory mapped streams (MMS) [3, 7].

In addition to the fbuf facility itself, we also propose an alternative application interface. Although fbufs can be accessed using the traditional Unix read/write interface, the design of the fbuf facility suggests a new model of application/buffer interaction that permits copy-free network I/O. The proposed interface embodies this model.

This paper makes three contributions. First, it evaluates existing techniques for cross-domain data transfer in the context of modern RISC architectures; this discussion is given in Section 2. Second, it describes and evaluates the fast buffer facility for cross-domain data transfers; the design is given in Section 3, a prototype implementation is described in Section 4, and a performance study is reported in Section 6. Third, it proposes an alternative application interface that supports copy-free network I/O; this interface is presented in Section 5.

2 Analysis of Existing Techniques

This section examines existing techniques for cross-domain data transfer, namely data copying, virtual copying, and shared buffers. This is helpful not only to understand the limitations of these techniques,

High-Performance Cross-Domain Data Transfer

Peter Druschel and Larry L. Peterson[1]

TR 92-11

Abstract

We have designed and implemented an operating system facility for high-performance cross-domain data transfer on uniprocessors and shared-memory multiprocessors. This facility, called *fast buffers* (fbufs), is designed to take advantage of modern RISC architectures that support large virtual address spaces. Its goal is to help deliver the high-bandwidth afforded by emerging high-speed networks to user-level processes, both in monolithic and small-kernel based operating systems.

March 30, 1992

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by National Science Foundation Grant CCR-9102040 and DARPA Contract DABT63-91-C-0030