Appears in IEEE Computer, March 1994.

Container Shipping: Operating System Support for I/O Intensive Applications

Joseph Pasquale and Eric Anderson

Computer Systems Laboratory
Department of Computer Science and Engineering
University of California at San Diego
La Jolla, CA 92093-0114

P. Keith Muller

AT&T Global Information Solutions
Decision Enabling System Division
San Diego, CA 92127

{pasquale, ewa, muller}@cs.ucsd.edu

Abstract

We present an operating system facility for efficiently transferring very large volumes of data between multiple processes and I/O devices, benefitting I/O intensive applications such as multimedia (e.g. video and image) and scientific computing. The execution of these programs often create an *I/O pipeline*, a set of processes that repeatedly carry out the following steps: a large data block (on the order of Mbytes) is input from a device, is transferred between and possibly modified by the processes, and eventually is output to a device. Each transfer may require a protection domain crossing, often with significant cost in CPU time, memory bandwidth and space, and bus bandwidth. Our "container shipping" inter-domain transfer facility involves no physical copying between domains, and only incurs cost proportional to the amount of data actually accessed, which is often a small fraction of the amount of data transferred, leading to increased throughput and reduced delay.

This work was supported in part by the Sequoia 2000 Project, DEC, IBM, NCR, TRW, and the National Science Foundation.

1.0 Introduction

A new generation of I/O devices with data rates ranging from 10 to 100 Mbytes per second are becoming available for personal computers and workstations. These devices include human interaction devices for video capture and display (and audio record and playback), high-capacity storage devices, and high-speed network communication devices. Along with continual improvements in processor, memory, and bus technology, these devices have enabled I/O intensive applications for desktop computing that require input, processing, and output of very large amounts of data. In this paper, we focus on an important aspect of operating system support for these types of applications: efficient transfer of large data objects between the protection domains in which processes and devices reside.

A rapidly growing class of I/O intensive applications is multimedia computing. In particular, we are referring to applications that acquire or present video (or image) and audio streams, possibly transforming them in novel ways under programmer control. These applications are often distributed and interactive, imposing real-time constraints for the delivery of large volumes of data being transported over potentially long distances. Examples include video teleconferencing with shared workspaces, remote scientific visualization and sonification, and distributed virtual reality.

Multimedia computing applications often require the manipulation of images whose sizes typically range from 1 to 10 Mbytes, but they can be larger. For example, an uncompressed High Definition Television (HDTV) frame requires 6 Mbytes, and a high-quality computer-generated video frame from a motion picture such as *Jurassic Park* requires 36 Mbytes. While compression can sometimes reduce these sizes by one to two orders of magnitude, usually processing must be done on uncompressed data.

Scientific computing and visualization are even more demanding. These applications often operate on data in the form of sequences of very large images. Examples include AVIRIS (Advanced Visible and Infrared Imaging Spectrometer) data requiring 140 Mbytes per image, and Landsat (Land Satellite) data requiring 278 Mbytes per image.

Generally we can structure these I/O intensive applications as multiple processes that transfer very large data objects among themselves and to file, database, network-protocol, and window servers. The dynamic computation structure formed by these interactions is typically an *I/O pipeline,* which we describe as a repeated activity in which

(1) a process inputs a large data object;

(2) the data is then sequentially transferred between the domains of various processes, each of which may read or modify varying portions of the data; and

(3) a process outputs the possibly modified data.

Processes that execute application-specific code may require access to some or all of the data. On the other hand, processes that control data transfer from and to I/O devices by direct memory access typically do not need to access any data, or may need access to only a small portion.

Many operating systems are very inefficient in transferring large amounts of data between domains. A prime example is Unix, which requires the physical copying of data between protection domains - for example, the kernel and user processes. Physical copying is detrimental to the performance of operating system and system-related software. This is most evident with network-protocol software implementations, where physical copying can consume a significant fraction of processing time for large packets [4].

Some operating systems use virtual transfers to try to avoid physical copying - that is, they

remap pages between virtual address spaces. Virtual transfers are one to two orders of magnitude faster than physical copying, but they can still lead to physical copying in certain cases. Furthermore, the cost of updating the state of the virtual-memory-management system (hardware and software) can be significant when the data transferred is 10 Mbytes or more.

Most operating systems, even when they try to avoid physical copying, offer a data-transfer model that assumes a need for complete accessibility to all transferred data. This assumption is too strong for most I/O pipelines and leads to overheads that can otherwise be avoided. Our design for an interdomain transfer facility (which was inspired by the "container-shipping" solution from the cargo-transportation industry) is based on virtual transfers and avoids all unnecessary physical copying. It is optimized for the data-access and transfer patterns of I/O pipelines. More conventional higher level interfaces (for example, Unix read and write) can be built on top of this facility to support non-I/O-intensive applications where physical copying is not a performance bottleneck.

After we present the I/O-pipeline model, we analyze issues relevant to the design of an operating system interdomain data-transfer facility. Then we present our design for such a facility.

2.0 The I/O Pipeline Model

An I/O pipeline is a model of a dynamic computation structure consisting of a sequence of domains: an *input* domain followed by one or more *intermediate* domains, and an *output* domain. A *domain* contains one or more processes, an address space in which the processes execute, a set of capabilities available only to processes in that domain (thus, the domain defines a boundary of protection), and various physical resources (for example, pages of physical memory). The address spaces of all domains may be independent of each other, or they may be separate regions in a universal address space. The input and output domains also contain input and output devices (or classes of devices); the input and output domains' processes act as device drivers. We assume devices transfer data at high speeds directly into physical memory allocated to their respective domains.

Data flows through the I/O pipeline by a sequence of interdomain data transfers. A *transfer* of data from one domain (called the *source*) to another (called the *destination*) makes that data available to processes in the destination domain. *Transfer models* (discussed in the next section) define whether the transferred data remains available to the processes in the source domain, and, if so, the dependencies between domains if the data is modified. Availability may be in the form of *accessibility* through the address space using memory load and store instructions, or in the form of a *capability* that defines permissible abstract operations on the data (such as make the data accessible or transfer the data to another domain). We reserve the term *access* to mean reading or writing the data through the address space. We specifically distinguish between the ability to transfer data and the ability to access it.

Figure 1 illustrates these concepts and shows how data is transferred through an I/O pipeline. A process in the input domain acquires data from the input device by direct memory access (DMA). That process (or another belonging to the same input domain) then transfers the data to an intermediate domain. The intermediate domain (and more generally, the destination domain of any transfer) may be known statically or determined dynamically. Processes in the intermediate domain may need to access none, a portion, or all of the transferred data. When all processes in the intermediate domain are done with the data, it is transferred to the next domain in the I/O pipeline. Each time the data is transferred, it may contain more or less data than what was previously received. Finally, when the data is transferred to the output domain, a process in that domain acti-

vates the output device that obtains the data by DMA.

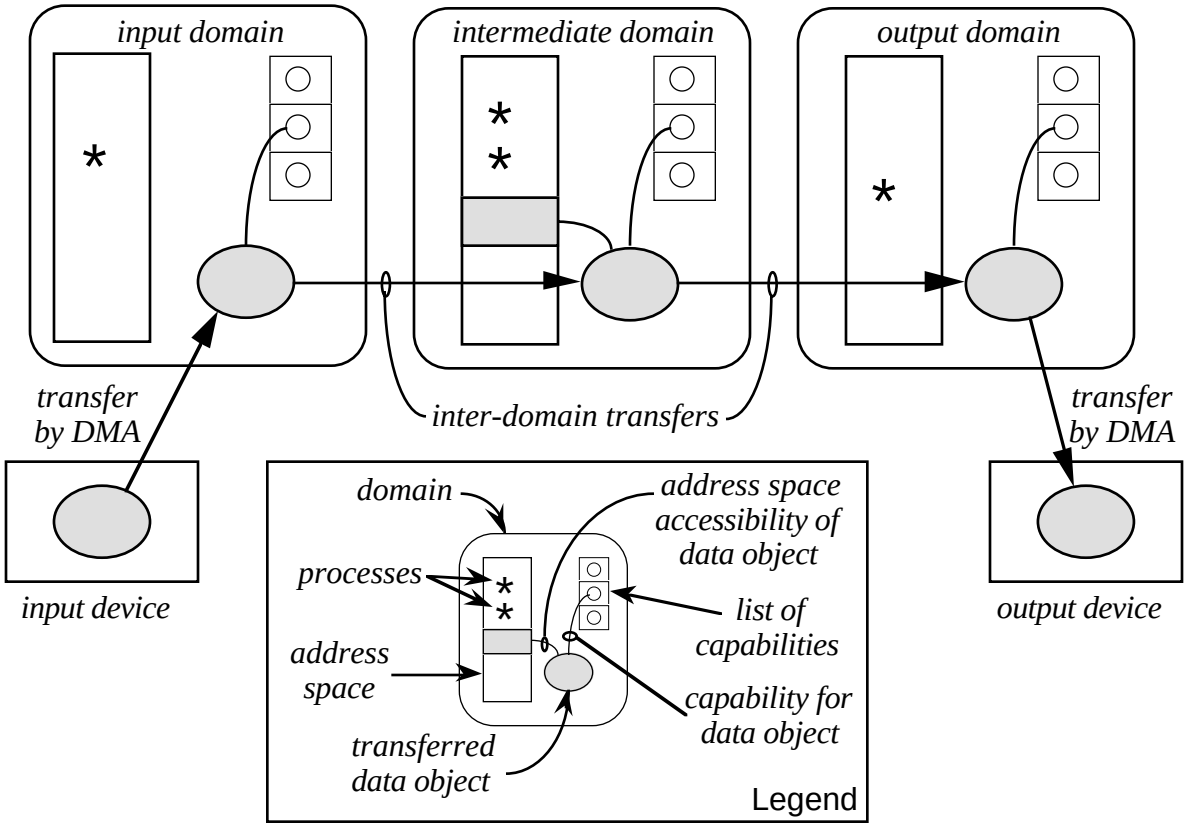

FIGURE 1. An I/O pipeline with three domains. An input device places data in the input domain by direct memory access. The data is then transferred from the input domain to an intermediate domain. (Only one intermediate domain is shown, but there can be a sequence of intermediate domains). The data is finally transferred to an output domain, from which an output device acquires the data by direct memory access. After a transfer, the data is available, either through the use of a capability or through accessibility via the address space, by processes in the destination domain. The data may or may not be accessible to the source domain, depending on the transfer model. The input and output domains here each have a single process (to execute device driver code), with the data available to it via a capability. The intermediate domain has two processes, with the data accessible to them via the address space.

There are no restrictions on concurrency within or between I/O pipelines. Thus, multiple transfers may be in progress at different stages of any single I/O pipeline, and a domain (and any processes associated with it) may be participating in multiple I/O pipelines.

As an example of an I/O pipeline, consider a video browser that requests video frames from a file server and then sends them to a window server. The file server makes requests to a disk-block server to obtain data from the disk device. The window server makes requests to a display server that controls a frame buffer's contents. Each server, as well as the video browser, is encapsulated by a domain. The I/O pipeline consists of, in sequence, the disk-block server, file server, video browser, window server, and display server. No process requires access to - the ability to read or write the contents of - the video-frame data being transferred. Each simply specifies that some portion (usually all) of the data gets forwarded to the next domain. For example, the window server may need to clip various portions of the video frame and therefore transfer only parts of it to the display server.

If we replace the video browser with a real-time video editor, the video editor might read or modify all or part of a video frame based on user directives. At the beginning of the I/O pipeline we could place a network server to remove protocol headers from network packets and a network-protocol server to combine packets into video frames. Both servers are examples of domains whose processes require access to a small portion of the transferred data - namely, protocol headers.

The types of I/O pipelines we want to support have characteristics important to our design:

- the data transferred between domains is not always accessed;
- the data is transferred after all processes in a domain are done accessing it;
- the amount of data transferred between domains is large (more than 1 Mbyte);
- the time for data to travel across the entire I/O pipeline is small and may be strictly bounded, say, to tens or hundreds of milliseconds; and
- the rate of transfer is high (more than 10 Mbytes per second).

3.0 Design Issues

In designing an interdomain data-transfer system, we considered transfer models, physical versus virtual-transfer implementation methods, and data structures. Our choices had a major impact on our system's performance, feasibility, and usefulness.

3.1 Models of Data Transfer

There are three models for transferring data between domains. In the *copy* model, data is copied from one domain to another - that is, the original still resides in the source domain and an exact copy resides in the destination domain. In the *move* model, data is removed from the source domain and placed in the destination domain. In the *share* model, after the data is transferred, processes in both the source and destination domains have access to the same data. Any modifications made to it by processes in one domain are visible to processes in the other. Figure 2 shows the different transfer models.

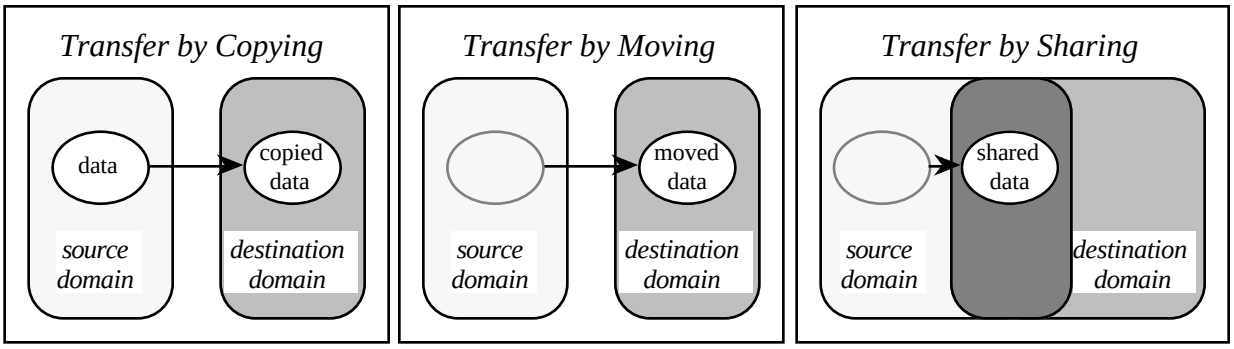

FIGURE 2. Models of data transfer. In the copy model, an exact copy of the data available in the source domain is available in the destination domain. In the move model, the data is removed from the source domain and made available in the destination domain. In the share model, the same data is available in both the source and destination domains.

Each model has certain disadvantages. In the move model, the source domain loses the data

after the data transfer. To avoid losing it, a process in the source domain would have to make a private copy before the transfer (or somehow arrange to have the data transferred back, making the loss temporary). The share model has the disadvantage that after the transfer, modifications on the transferred data by a process in one domain can affect processes in the other domain that depend on that data. Since these modifications are asynchronous (from the point of view of the other processes), explicit synchronization between processes in both domains may be required. Such coupling of the source and destination domains increases programming complexity and can propagate errors across domains. We describe the copy model's disadvantage in the course of our discussion of physical and virtual transfers.

3.2 Physical versus Virtual Transfer

How each transfer model is implemented greatly affects its performance with large data transfers. The major implementation issue is whether data transfers are physical or virtual. A *physical* transfer involves moving data in physical memory - that is, moving each byte (or word) of data from the source domain's physical memory to the destination domain's physical memory. A *virtual* transfer involves moving data in virtual memory. In other words, the transfer maps a region in the destination domain's address space to the physical pages (already mapped in the source domain's address space) that contain the data to be transferred. These physical pages are *transfer pages*. (We assume the page-based virtual-memory architectures used in most popular workstations. Similar arguments hold for other virtual-memory architectures, although the details may differ.)

Physical transfers generally apply to the copy model. They do not make much sense with the move model, since erasing the data in the source domain adds cost. When the physical memories of the source and destination domains are separate, physical transfers in the share model require an underlying process to keep the copies in the memories consistent.

Physical transfers promote flexibility. Since the transfer size granularity is the byte (or possibly, the word), we can transfer data from any location and of any size to a destination space that can begin at any location and whose size is exactly the data's size. With virtual transfers, the transfer size granularity is the physical page. The transferred data must be contained in the one or more transfer pages exclusively; these pages can contain no other data. Since entire pages are the actual units of transfer, the data's destination address must be at the same relative offset from the page boundary as the source address. Moreover, the size of the destination space must be the size of the number of transfer pages, which is usually greater than the data's size (unless it happens to be exactly a multiple of the page size).

The flexibility provided by physically transferring data is overshadowed by its primary disadvantage: high overhead in time and space. Physically transferring a word involves two memory accesses to read and write. Memory-access instructions take significantly more time than non-memory-access instructions, and the gap is widening as RISC architectures continue to allow CPU speed to improve faster than memory speed [5]. While the time to transfer a single word is insignificant, the time to transfer a large data object (say, 1 Mbyte to 100 Mbytes) is significant. Multiply this by the number of domain transfers in an I/O pipeline, and the total transfer time can be many times the total computation time required for the data object. In an I/O pipeline, many of the domains do not require access to some or all of the data, making these costly physical transfers especially wasteful. Furthermore, the amount of physical memory used during a physical transfer is twice the size of the data object being transferred (enough to store at the destination while reading from the source). Transfers must be delayed if sufficient physical memory is temporarily

unavailable. Both of these factors degrade performance by increasing delay and decreasing throughput. Figure 3 shows how physical copying between kernel and user process domains occurs in Unix buffered I/O, leading to performance degradation for large data transfers.

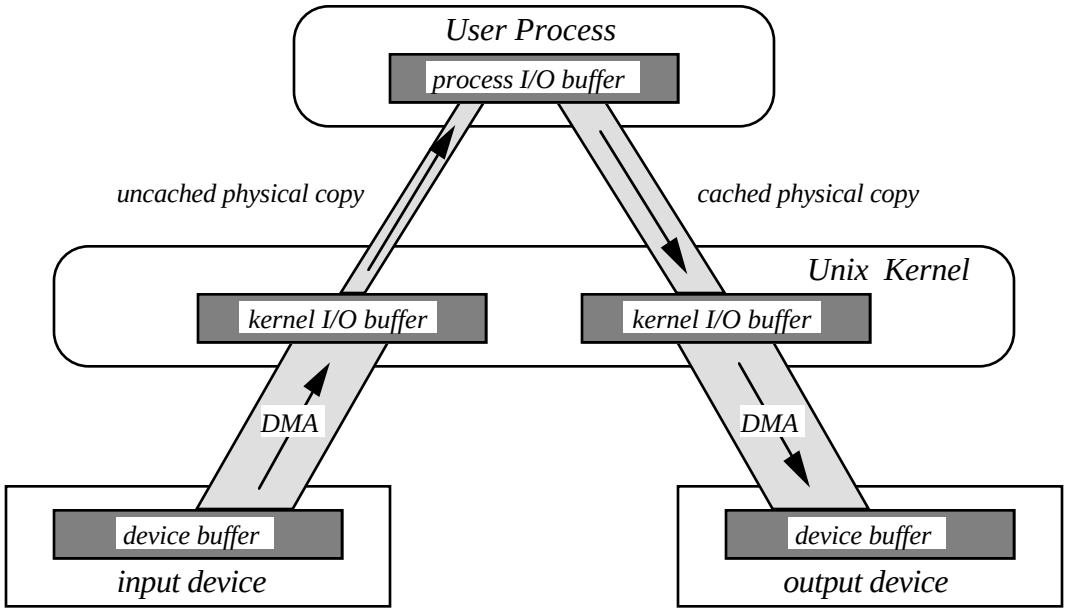

FIGURE 3. Physical copying in a simple Unix buffered I/O pipeline. A process reads from one device, possibly accesses the data, and then writes to another device. The relative widths of the data paths reflect data rates measured on a DECstation 5000/200: a narrow data path for physical copying of uncached data (14 Mbytes per second), a medium data path for physical copying of cached data (40 Mbytes per second), and wide data paths for direct memory access transfers (93 Mbytes per second). Overall throughput is limited by the uncached physical copy rate which is the bottleneck of the entire pipeline. The overall delay is increased by the multiple transfer times to copy the data between the domains. Physical copying was measured using the `memcpy` C library routine.

If physical transfers are too costly, the alternative is virtual transfers. We implement a *virtual copy* by mapping a region in the destination domain's address space to the transfer pages while not affecting their mapping in the source domain's address space. If processes in both domains only read the data, a virtual copy is as good as a physical copy. If a process tries to modify the data, a physical copy of the page containing the data is made so the modifications do not affect the data in the other domain. Unfortunately, implementations utilizing this copy-on-write mechanism are often complex compared with implementations based on simple physical copying. Furthermore, servicing a copy-on-write fault and physically copying the data are especially wasteful when the source domain does not need the data, as is often true in I/O pipelines.

A *virtual move* unmaps the transfer pages from the source domain's address space and maps them into a region in the destination domain's address space. Unlike in virtual copying, no copy-on-write mechanism is necessary, making this scheme simple and efficient. If a source-domain process does not want to lose the ability to access the transferred data, it explicitly makes a copy before transferring, thus incurring the cost of an expensive physical copy only when necessary. Unfortunately, a process does not always know a priori whether it needs a copy and may make unnecessary physical copies. An alternative solution when this is a potential problem is for source- and destination-domain processes to arrange for transferred data to be transferred back in its original form, thus avoiding physical copying. Figure 4 shows the differences between physical copy-

ing and virtual moving.

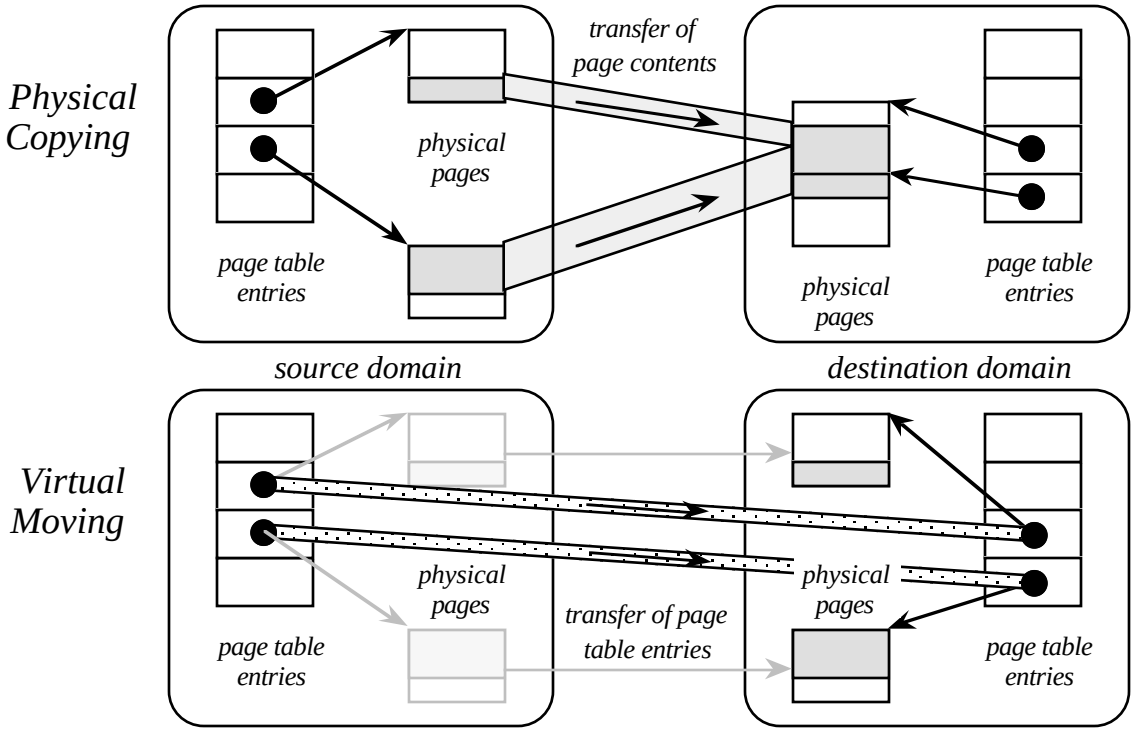

FIGURE 4. Physical copying versus virtual moving. Physical copying (top) is based on physical transfers. A region starting at an arbitrary location in the virtual address space can be transferred to an arbitrary location in the destination domain's address space, because the contents of physical pages are copied byte by byte (or word by word). Virtual moving (bottom) is based on virtual transfers. The contents of page-table entries are copied to the destination domain (and then removed from the source domain in the case of virtual moving). Thus, the granularity of data transfer is the page.

Under the share model, regions in address spaces of both the source and the destination domains are mapped to the same transfer pages. This scheme has the advantage of virtual copying in that the source domain does not lose the data and does not incur the cost of copy-on-write, but disadvantages arise from the implicit coupling of the source and destination domains.

3.3 Organization of Transfer Data

Data transferred between domains must be organized in a way source- and destination-domain processes agree upon. The organization can affect performance by forcing physical copies, either in preparing for or during the transfer. Of the three organizations we consider here - unstructured, structured, and semistructured - the simplest and most common is an unstructured array. Only raw data is transferred, and no other information, such as a set of pointers, imposes further structure.

However, transfer data is often not in the form of a single array. It may be stored in numerous pieces organized by a more complicated data structure such as a tree. Consequently, before it can be transferred, it must be *linearized* - physically copied into a single array containing only raw data. Thus, if processes agree on the unstructured organization for transferring data, physical copying generally takes place in preparation for the transfer.

Structured data has some special organization and may include pointers to define its structure. In particular, the transferred data will not be in a single array, so destination-domain processes

must have methods to access the data according to its special structure. Generally, potential receivers do not know these methods, which must also be communicated. Furthermore, if the data is embedded with pointers that are virtual addresses, they could require translation in a new address space. (If all domains use the same address space, this translation is unnecessary.)

Semistructured data consists of pointers and raw data arrays, with each pointer (possibly) referring to a raw data array. This organization avoids the disadvantages of unstructured and structured organizations. It does not have the single-array requirement of unstructured data, and so linearization is not necessary. While semistructured data has associated pointers, these are separate from (not embedded in) the raw data and can quickly be located for possible translation and separate transference. Furthermore, the access method is well known and does not have to be specially communicated.

A semistructured data organization has additional benefits. We gain the flexibility of locating raw data at arbitrary locations within pages that contain no other data, which we can then use as transfer pages for virtual-transfer methods. Moreover, semistructured data is well matched to the scatter/gather DMA programming requirements of most high-speed devices. The device controller can output semistructured data directly using gather DMA. On input, if there is an advantage to generating semistructured data, the controller can do it with scatter DMA. Physical copying is reduced because there is no need to linearize, as is the case with network protocols that require fragmentation and defragmentation of messages.

4.0 The Container-Shipping Transfer Facility

We formulated three design goals for our interdomain data-transfer facility:

(1) *Performance*. The facility should provide the highest possible performance for I/O pipelines. The transfer mechanism should be efficient and minimize data-transfer overhead.

(2) *Simplicity*. The design should be simple to implement, and its concepts and usage semantics easy to understand.

(3) *Safety*. The design should not discourage the use of separate protection domains, because they are a valuable structuring principle for building reliable systems.

4.1 Design Principles

To achieve these goals, we based our design on four principles for the transfer of very large data objects through I/O pipelines.

(1) *Transfers should not cause physical copying, directly or indirectly.*

Because physical copying is the prime source of inefficiency, our first principle led us to use virtual rather than physical transfers. We also avoided virtual *copying* because it can indirectly cause physical copying. Unstructured data organization would require physical repositioning of non-page-aligned data, so we chose a semistructured data organization. This gave us the flexibility of pointing to where the data is within transfer pages while avoiding the structured organization's need to communicate access methods.

Although significantly less expensive than physical transfers, virtual transfers can still be quite costly when remapping very large data objects. For example, the 278-Mbyte Landsat image mentioned earlier would require 68,000 remaps per interdomain transfer on a DECstation 5000/200

(the workstation on which we did our measurements) which has a 4-Kbyte page size. Running a modified Mach 3.0 kernel on that workstation, Druschel and Peterson achieved a minimum of 22 microseconds per remap, although they point out that a remap time of 42 to 99 microseconds is more realistic [2]. Even using the optimistic remap time of 22 microseconds and assuming four interdomain transfers, the cumulative time to simply transfer the Landsat image through the I/O pipeline (without it even being accessed) is 6 seconds, *and this is avoiding physical copying*! Clearly, this is a severe performance penalty, and safety may suffer if it encourages partitioning software systems into fewer domains to avoid long I/O pipelines or to simply have all domains share memory.

(2) *The cost to transfer data should not depend on the cost to access it.*

In most systems, data transferred to a domain is automatically accessible to those domain processes. Because many domains in an I/O pipeline have no need to access the data, we separated the transfer and access mechanisms. Data transfer can be implemented very cheaply by simply passing a capability. The data need not be accessible in either the source or destination domain. The cost of a transfer then becomes insignificant relative to other basic overheads such as domain-switching time and is independent of the amount of data transferred. If transferred data is never accessed in any pipeline domain, the delay should be roughly the time needed for input and output transfers by DMA, and the maximum throughput could theoretically be half the DMA rate (assuming a shared I/O bus and no bus contention).

This estimate is, however, highly optimistic because a process must still execute between inputting and outputting. Thus, performance will be adversely affected by context-switch overhead and scheduling delays. An alternative approach is to remove the process from the I/O loop, creating a direct "in-kernel" data path between I/O devices [3].

Avoiding access costs should not be limited to situations where no data is accessed. Ideally, the cost should vary based on the fraction of transferred data actually accessed, rather than the total amount potentially accessible. Lazy page mapping is a way of achieving this proportional cost. When a process attempts to access a page of transferred data, a fault occurs and the fault handler then maps the page. However, lazy page mapping introduces costs that can be significant: fault-setup overhead for all transferred pages and fault-handling overhead for all accessed pages. Furthermore, the complexity of lazy evaluation schemes conflicts with our next principle.

(3) *Favor a simpler mechanism that lets the programmer create an optimization rather using a more complex mechanism that has the optimization built in.*

According to this principle, we should provide the programmer with a mechanism for runtime specification of the portion of data to be accessed. A semistructured data organization allows selective mapping of different unstructured components and gives the programmer the right level of abstraction to control the portion of data made accessible. This places the burden on the programmer, but it simplifies the implementation and still permits optimizations.

(4) *Avoid mechanisms that allow processes in separate domains to affect each other's state in an uncontrolled way.*

This principle dictated that we use the move model. (We had already eliminated the copy model because of its inefficiency.) The share model, from a user's perspective, adds the complexity and inconsistencies that arise from asynchronous updates.

4.2 Container Shipping

Our design principles led us to use the move model and virtual transfers, a semistructured data organization, and separate transfer and access mechanisms. Much of the inspiration for our implementation of these design decisions came from cargo transportation. The problems encountered in moving cargo efficiently are similar to those we are trying to solve, and many were solved by "containerization," an important advance of the 1960's. Containerization is the use of standardized fixed-size containers for loading, transporting, and unloading cargo. The central idea is that independent entities need not agree on the size of the item(s) being shipped, since they have already agreed upon the container around which transport has been optimized. These optimizations include efficient local positioning of containers for storage, loading, and unloading of items; efficient transport of containers between the docking station and the vehicle of transportation (ship, railroad car, or truck), and efficient positioning of containers on the vehicle.

Adopting cargo-transportation terminology, we call our interdomain transfer facility *container shipping* and call the basic objects pallets and containers. A *pallet* is a data space in contiguous virtual memory in units of pages and corresponds to its physical counterpart, a portable standard-sized platform on which items to be shipped are placed. A pallet can contain *valid* data, which is an unstructured data block. The data block begins at some offset (possibly 0) from the beginning of the pallet and has some length that does not go beyond the end of the pallet. As Figure 5 shows, a *container* is an ordered set of pallets and corresponds to its physical counterpart, a receptacle that holds real pallets.

Container with a 1 page pallet and a 2 page pallet.

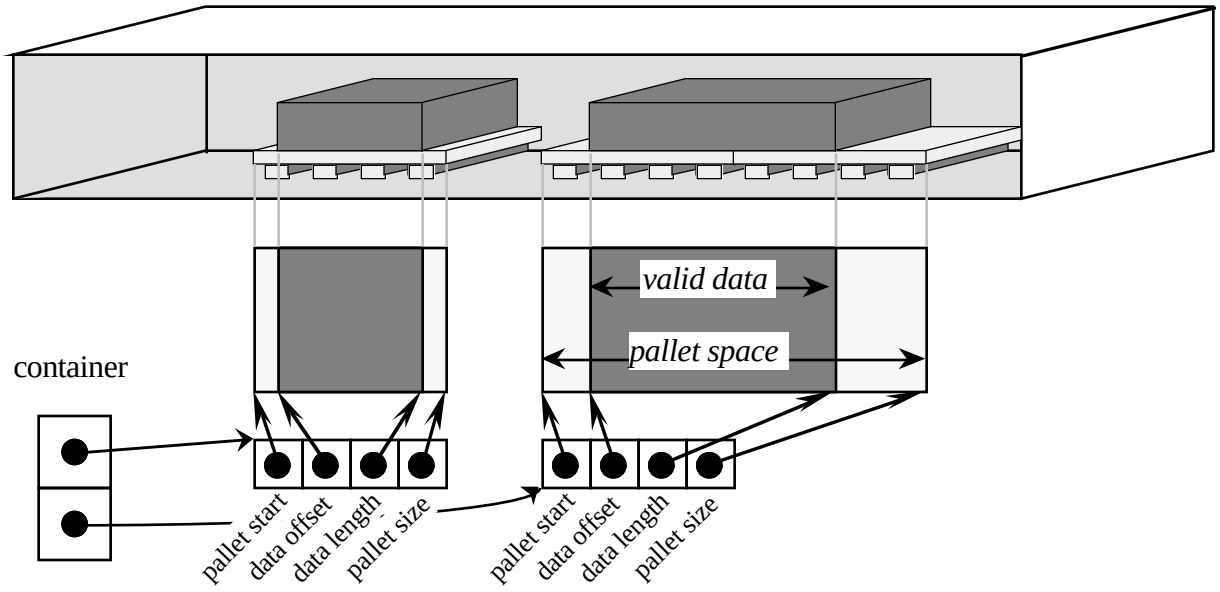

FIGURE 5. Pallets and containers. On top is a conceptual view of a container with two pallets, and below it is its data structure. Pallets come in different sizes in units of pages. Valid data on a pallet begins at an offset from the start of the pallet space and has a length that must not cause the data to go beyond the end of the pallet space.

Pallets get unloaded from and reloaded to containers, and containers get shipped between domains. Pallets are the units of *access*, and containers are the units of *transfer*. The data structure of a container with pallets has a semistructured organization, where each component of unstruc-

tured data is individually selectable for access: Each pallet has its own set of physical pages.

Our system provides eight operations on containers:

- *Alloc* allocates a new container;
- *Free* frees a previously allocated container;
- *Fill* requests that a device deposit data into a container;
- *Empty* requests that a device retrieve data from a container;
- *Unload* removes pallets from a container, making their contents accessible;
- *Reload* places a previously unloaded pallet back into a container, making its contents inaccessible;
- *Ship* transfers a container to another domain; and
- *Info* retrieves information about the container.

Depending on the operating system into which container shipping is incorporated, the interprocess communication (IPC) facility may encapsulate some of the operations - for example, *Ship* - while in other cases all the operations may be kept separate from the IPC facility.

Although we expect there will be different ways of using container shipping, we describe a highly stylized use based on the I/O-pipeline model. Italicized words correspond to container operations.

In preparation of transferring data yet to be generated, a container is *allocated* with one or more pallets of specified sizes. Multiple pallets are used when multiple data items are expected to be selectively accessed by receiving domains (which the source domain knows by previous arrangement).

In the input domain, an input device *fills* the container by depositing valid data directly into its pallets by DMA. The interface lets the programmer specify per-pallet offsets and lengths indicating where valid data should be placed and how much space should be used in each pallet of the container.

The container is then *shipped* to an intermediate domain. The input domain loses possession of the container, and the intermediate domain gains possession. Processes in the intermediate domain may now select one or more container pallets to be *unloaded*. Unloading a pallet causes its space to be mapped into a free region of the domain's address space, making it accessible - readable and writable - to processes using ordinary memory-access instructions. The interface provides information about the location and size of this space and the location and size of the valid data within it.

A previously unloaded pallet may be *reloaded* into a container. This causes the pallet to be unmapped from the domain's address space, making it inaccessible (unless it is subsequently unloaded again). The interface lets the programmer respecify the location and size of the valid data in the pallet space.

When all processes in the intermediate domain finish with the container, it is shipped to the next domain. If this is another intermediate domain, the same activity of unloading and then reloading pallets of interest may occur. Or a container can be shipped without having any pallets unloaded or reloaded.

When the container reaches the output domain, it is *emptied* by an output device, which acquires valid data from the container's pallets by DMA. The interface lets the programmer specify per-pallet offsets and lengths indicating what parts of the valid data should be retrieved for out-

put.

If the container is no longer needed, it is *freed*: All resources associated with the container - for example, physical pages of pallets - are freed.

A variation on this pattern permits return of containers to the original allocator (known by pre-arrangement) rather than freeing. Often it is advantageous for the initiator of an I/O pipeline (which often does not correspond to the input domain) to be the allocator of a container. This ensures that it is configured properly, that is, allocated with a certain number of pallets, each of a certain size. The system then ships the container to an input domain where it is filled and then passes it through the rest of the I/O pipeline. The initial “filler” of a container need not be the device of an input domain. A process in any domain can allocate a container, unload its pallets (which are initially empty), write data on them, reload the pallets, and ship the filled container.

4.3 Discussion

We optimized the container-shipping design and implementation for data transfers according to the I/O pipeline model we presented earlier. Shipping is based on the move transfer model, which is easy to understand and implement because data is never shared. Data always belongs to one, and only one, domain. We decoupled shipping of containers from making their contents accessible, which can be done selectively. Consequently, the shipping of containers is a very cheap operation that consists of a simple update to a system-wide table defining possessions of containers by domains. It does not automatically incur the cost of mapping their contents, which is significantly more expensive.

When a process unloads or reloads a selected pallet, it is mapped or unmapped into the process address space. This *selective mapping* requires no physical copying, and page mapping is done only on data to be accessed. The programmer selects the pallets to be unloaded, so the mapping cost is incurred on the basis of need. A Fill or Empty operation on a container does not cause any of its pallets to be mapped into the input or output domain’s address space. If a process needs access, it must unload the pallets. The interface lets the programmer manipulate pallets and not pages (out of which pallets are constructed): Pallets are machine-independent while pages are machine-dependent.

The separation of transfer and access also lets us avoid page cleansing. In general, when a physical page is allocated to a domain different from the one it was previously allocated, it must be cleansed to maintain information security. Page cleansing is a very expensive operation. A zero must be physically copied into each word of the page. However, cleansing pages belonging to a pallet of a newly allocated container can be safely delayed until the first time the pallet is unloaded, because the pallet’s contents remain inaccessible until this time. Typically, an input domain process calls a *Fill* operation before any pallets are unloaded. If a pallet page is to be completely overwritten by an input device (using DMA), cleansing the page is unnecessary.

There is no security problem if the container is emptied. The Empty operation permits only valid data in the container to be output to a device. When a container is allocated, all its pallets have no valid data areas. A pallet can acquire valid data only by being unloaded (so that a process can modify the pallet’s data space) and then reloaded, or by being filled by Fill, which does its own page cleansing if necessary.

Typically containers of the same type (configured with the same number of same-sized pallets) continuously travel through an I/O pipeline. We can take advantage of this regularity. We optimize address space allocation by caching the association of unloaded pallets of a recirculating container with a region of the address space (as in the Fbufs caching technique [2]). Each time the

container is transferred to a domain, the unloaded pallets get mapped to the same regions (if possible). To apply this optimization to multiple containers of the same type traveling through an I/O pipeline, a process can specify that a container be allocated based on a previously allocated "prototype" container. We also cache the association of a container and the physical pages of its pallets. This avoids page cleansing in various common cases of container reallocation to the same domain.

The semistructured data organization lets us use scatter/gather DMA, which creates or accepts an array of pointers to data segments and their sizes. It also lets network protocols prepend headers to data on output and remove headers on input. Semistructured organization also lets us break large data objects into components and access only a few. For example, we can decompose a video frame into hierarchically coded subframes.

4.4 Performance

We conducted performance experiments on a DECstation 5000/200 running the Ultrix 4.2a operating system (a derivative of Berkeley UNIX) with support for our container-shipping facility. In our model, a domain with one process corresponds to an Ultrix process that includes its own address space.

We measured the I/O pipeline transfer plus access *(T+A)* time. For an I/O pipeline consisting of *n* domains, the *T+A* time consists of the time for *n*-1 interdomain data transfers, the time to access the data in one domain, and any software overhead (like user program execution, system calls, and domain switching). Note that the *T+A* time does not include physical I/O time, which is purely dependent on the speed of the I/O bus and I/O devices rather than on the software facility that we wanted to measure.

We created an I/O pipeline with five domains. The amount of data transferred was always 10 Mbytes, and a specified portion of it was accessed by physically copying it from one (uncached) location to another. We evaluated the performance of three types of transfer:

- *Physical copying*. Each transfer is a physical copy between domains.
- *Virtual moving*. Each transfer is a virtual move with full mapping of the transferred data in each domain.
- *Container shipping*. Each transfer is a virtual move, but only pallets that are accessed are unloaded (selectively mapped). We used a container consisting of five pallets, each holding 2 Mbytes of data.

Figure 6 shows the I/O pipeline throughput, computed as the amount of data being transferred divided by the measured *T+A* time as a function of the amount of accessed data. Container shipping completely outperforms physical copying by a factor of almost 4400 (15.0 gigabytes per second, or GBps, to 3.43 megabytes per second, or MBps) if none of the data is accessed, and a factor of 5 (13.9 MBps to 2.75 MBps) if all the data is accessed. Physical copying is very costly, with a *T+A* time ranging from 3.0 to 3.7 seconds, depending on how much of the 10 Mbytes of transferred data is accessed. However, container shipping is also better than virtual moving, especially when only a small amount of data is accessed. Container shipping outperforms virtual moving by a factor of 64 (15.0 GBps to 234 MBps) if no data is accessed, and by a factor of 5.8 (1.16 GBps to 201 MBps) if 100 Kbytes is accessed. Even if 1 Mbyte is accessed, container shipping is 57 percent better than virtual moving. As more data is accessed, the costly memory access time becomes

the dominating factor in limiting throughput.

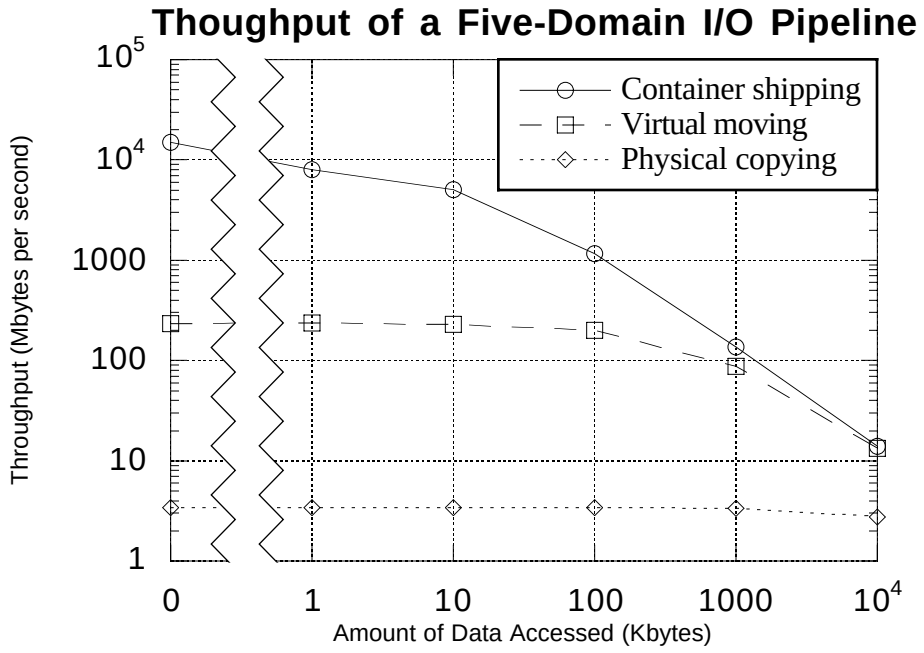

FIGURE 6. Five-domain I/O pipeline throughput when transferring 10 Mbytes of data. One domain accesses varying portions of the data. Container shipping (the container has five pallets, each containing 2 Mbytes of data) outperforms physical copying and virtual moving because cost depends only on what is accessed. Note that this is a log-log graph cut to show the 0 point.

One of the major problems with physical copying and virtual moving is that their performance does not scale well with the number of domains in the I/O pipeline. Unlike container shipping, physical copying and virtual moving both incur cost to make the data accessible *for each domain* regardless of how much, if any, of the data is to be accessed. This is evident from the left side of the graph in Figure 6 where their throughputs are well below that of container shipping when no data is accessed. Thus, container shipping is more scalable because it is relatively insensitive to the number of domains over which data is transferred, and a transfer requires little work unless the data is actually accessed.

5.0 Related Work

The use of virtual transfer techniques to avoid the performance penalty of physical copying is not new. Tenex [1] was one of the first systems to use virtual copying. Accent [6] generalized this concept by integrating virtual memory and IPC so that messages could be virtually copied.

We have especially benefitted from more recent systems that make extensive use of virtual transfer techniques: Tzou and Anderson's DASH interprocess communication design [7], and Druschel and Peterson's Fbufs interdomain transfer design [2]. Tzou and Anderson based interdomain transfers on virtual moving with optional lazy page mapping, and they addressed the difficult problem of consistency in the translation look-aside buffer when using virtual moving on a multiprocessor. Druschel and Peterson's interdomain transfer facility supports the move and share transfer models, a flexible semistructured data organization, and clever optimizations such as caching maps of transferred buffers. Implementations of both designs achieve high throughput.

6.0 Conclusions

Most operating systems do not support I/O-intensive applications well because they cannot transfer data efficiently. Our container-shipping design for an interdomain transfer facility uses virtual transfers based on the move model, and a semistructured data organization to achieve significant performance improvements. Its decoupling of the transfer and access mechanisms is visible to the programmer, who uses selective mapping only for parts of the transferred data that need to be accessed. Container shipping is well suited for transferring very large data objects through I/O pipelines in which each domain may selectively modify the transferred data and then forward it without further need for it.

Compared with previous systems, container shipping supports simpler and safer implementations with inherently scalable performance. A programmer who wishes to use the container-shipping interface directly to assure high performance assumes a greater burden, but we have not found the burden unreasonable. Higher level and more conventional interdomain transfer facilities, such as Unix read and write, can be built on top of container shipping, but they may not perform as well because they require physical copying. As we gain more experience with applications that use container shipping, we hope to analyze the design more critically and report more extensively on performance.

7.0 References

[1] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a paged time sharing system for the PDP-10," *Comm. ACM*, Vol. 15 (3), (March 1972), pp. 135-143.

[2] P. Druschel and L. Peterson, "Fbufs: a high-bandwidth cross-domain transfer facility," *Proc. 14th ACM Symp. Operating System Principles (SOSP)*, Asheville, NC, (December 1993), pp. 189-202.

[3] K. Fall and J. Pasquale, "Improving continuous-media receiver performance with in-kernel datapaths," *Proc. IEEE Intl. Conf. on Multimedia Computing and Systems (ICMCS)*, Boston, MA, (May 1994), to appear.

[4] J. Kay and J. Pasquale, "The importance of non-data touching processing overheads in TCP/IP," *Proc. ACM Communications Architectures and Protocols Conf. (SIGCOMM)*, San Francisco, CA, (September 1993), pp. 259-269.

[5] J. K. Ousterhout, "Why aren't operating systems getting faster as fast as hardware," *Proc. USENIX Summer Conference*, (June 1990), pp. 247-256.

[6] R. Rashid and G. Robertson, "Accent: a communication-oriented network operating system kernel," *Proc. 8th ACM Symp. Operating System Principles (SOSP)*, Pacific Grove, CA, (December 1981), pp. 64-85.

[7] S-Y. Tzou and D. P. Anderson, "The performance of message-passing using restricted virtual memory remapping," *Software - Practice and Experience*, Vol. 21 (3), (March 1991), pp. 251-267.