APPLICATION PROGRAM INTERFACES

Applications usually do not perform directly certain operations, such as controlling input/output (I/O) devices. On most operating systems, in order to perform I/O, applications act as *clients* that request services from system *servers.* Servers process each request and reply to the respective client, as illustrated in Fig. 1. The format of a server's requests and replies defines that server's application program interface (API).

The goals of an API include:

1. *Portability.* By offering uniform APIs on different platforms (e.g., different hosts and I/O devices), operating systems may eliminate or greatly reduce the effort necessary for porting applications between those platforms.
2. *Modularity / Software Reuse.* By encapsulating into a server services that many applications require, operating systems make it unnecessary for application writers to reimplement those services.
3. *Protection.* By implementing clients and servers in separate protection domains, operating systems can prevent unauthorized users from performing operations that could compromise system protection or integrity. A protection domain defines the memory addresses and objects that code running in it can access. Typical implementations base protection domains on CPU-enforced privilege levels (e.g., kernel or user mode) and virtual memory (VM) mechanisms. By setting up domains so that only a certain trusted server can access network interface hardware directly, for example, operating systems can prevent malicious or buggy applications from gaining access to other users' packets or disrupting the operation of the interface.
4. *Low Overhead.* Ideally, the API should be implementable so that performance is not substantially worse than would be the case if the client itself performed the services (in a nonportable, nonmodular, unprotected manner).

Unfortunately, it can be difficult to achieve simultaneously all these goals. Mainstream operating systems, for example, often incur significant overhead to attain protection. The main sources of overhead are typically:

1. *Data Passing.* Requests and replies often require passing data between client and server because each party holds data in buffers that are not usable by the other

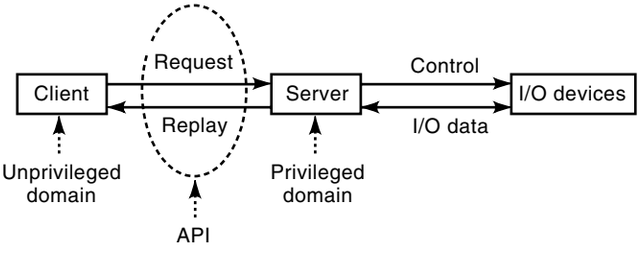

Figure 1. Application program interfaces (APIs) promote portability, modularity, and protection.

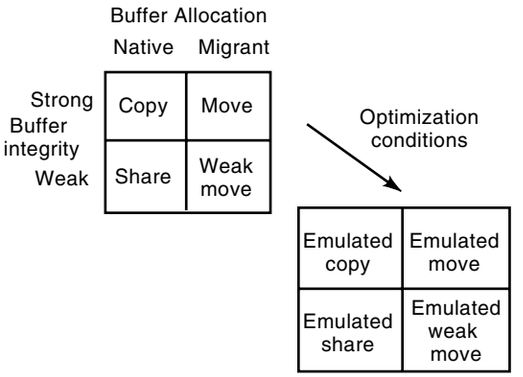

Figure 2. Data-passing taxonomy. Buffer allocation and integrity define the data-passing *semantics.* The *qualified semantics* also takes into account optimization conditions.

party. For example, client buffers may be pageable (i.e., not always present in physical memory), whereas a server requires unpageable buffers, or a server may use buffers that are not accessible by its clients. Many systems, including Unix (1), pass data by copying. The cost of copying, however, has become quite high because memory performance improvements have long been trailing behind performance improvements in processors (2) and I/O devices such as fiber-optic networks (3).

2. *Control Passing.* Monolithic systems, such as Unix (1), integrate most servers in the kernel and run applications as user-level processes. A request is therefore implemented as a system call, the overhead of which is typically much larger than that of a simple function call. On the other hand, microkernel systems, such as Mach (4), implement both applications and most servers as user-level processes. Requests and replies then require interprocess communication (IPC). IPC is even more expensive than a system call because IPC also requires a VM context switch and process scheduling.

Techniques for reducing data and control passing overheads have been proposed in the literature. However, many such techniques change the system's API in an incompatible way (3), sacrificing portability.

This article characterizes API data passing and control passing, presents examples of actual APIs, and discusses data-passing and control-passing optimizations and their performance effects.

DATA-PASSING CHARACTERIZATION

The taxonomy shown in Fig. 2 classifies API data passing according to three orthogonal characteristics: (1) buffer allocation, (2) buffer integrity, and (3) optimization conditions. The following subsections discuss each characteristic in turn.

Buffer Allocation

Data passing may or may not imply allocation and deallocation of the buffers that contain the data, and each option requires a fundamentally different API. In the taxonomy of Fig. 2, buffers allocated or deallocated by virtue of I/O requests are called *migrant,* whereas all other buffers are called *native.* In migrant-mode data passing, the API implicitly allocates input buffers and deallocates output buffers. When making an input request, the application cannot choose the location or layout of input buffers; after making an output request, the application cannot or should not access output buffers.

In native-mode data passing, on the contrary, the API does not allocate or deallocate buffers. When making an input request, the application specifies the location and layout of its input buffers; after making an output request, the application can still access output buffers.

Migrant-mode and native-mode data-passing semantics necessitate different APIs. The main difference regards input buffers: In migrant-mode APIs, the location and layout of input buffers are *output* parameters, returned by the interface; in native-mode APIs, the location and layout of input buffers are *input* parameters, passed to the interface. Migrant-mode APIs also include primitives for explicit migrant buffer allocation and deallocation. Applications with balanced amounts of input and output may be able to avoid explicit buffer allocation and deallocation by reusing input buffers as output buffers.

Migrant-mode APIs should accept as output buffers only migrant buffers. This restriction prevents native regions that must be kept contiguous, such as the stack or the heap, from becoming discontiguous because a migrant-mode interface accepts part of the region as an output buffer and deallocates it, making the region discontiguous. Native-mode APIs can accept as input or output buffers both native and migrant buffers.

Native-mode APIs are far more common than are migrant-mode ones. Native-mode data passing is typified by the usual *copy* semantics of Unix's `read` and `write` calls (1). Networking APIs (e.g., BSD Unix sockets (1), Unix SVR4 TLI (5), Windows NT Winsocks) typically have copy semantics. Migrant-mode data passing is typified by the *move* semantics of certain experimental APIs, such as DASH (6), Alloc Streams (7), and Container Shipping (8). Move semantics normally is implemented by unmapping the pages containing the data from one party's address space and mapping those pages to a new region in the other party's address space.

Buffer Integrity

API data passing can be with *strong* or *weak* buffer integrity guarantees. Strong-integrity data passing guarantees that: (1) the owner of an output buffer cannot, by overwriting the buffer after data passing, affect the contents of the other party's input buffer; and (2) the owner of an input buffer can access the buffer only in the states as of before an input request or after successful reply, but not in an intermediate, inconsistent, or erroneous state. Weak-integrity data passing makes no such guarantees.

Copy and move semantics both provide strong integrity because each party cannot access the other party's buffers. On the other hand, weak integrity allows *in-place* data passing, that is, data passing using buffers that can be accessed by both parties. The client can access these buffers while its request is being processed and, consequently, can corrupt output data or observe input data in inconsistent states.

Native-mode weak-integrity data passing defines *share* semantics, whereas migrant-mode weak-integrity data passing

defines *weak move* semantics. Under weak move semantics, an output buffer remains physically accessible to its previous owner after data passing, but this previous owner should not access the buffer because the other party becomes the logical owner of the buffer and may reuse it.

For weak-integrity, in-place input, requests have to be made *before* input physically occurs. If this condition is not met (e.g., when a packet is received unexpectedly from a network), input can be completed according to the strong-integrity semantics with the same buffer allocation scheme (i.e., share reverts to copy semantics, and weak move reverts to move semantics). Additionally, for correctness, clients should not access a weak-integrity buffer during processing of a request that uses that buffer (and, in the case of output with weak move semantics, any time thereafter).

In Unix, share semantics is an exception, used in `read` and `write` system calls for the case of raw (uncached) disk I/O (1). Weak move semantics is used in some experimental APIs, for example, cached and cached volatile fbuf input (9) and exposed buffers (10).

Optimization Conditions

Each data-passing semantics may admit many different optimizations, some of which may depend on special conditions. An API's data-passing *qualified semantics* is defined by the API's data-passing semantics and special *optimization conditions.* Contrary to buffer allocation and integrity, which each admit only two alternatives, optimization conditions admit a spectrum of possibilities, including many not discussed here.

Optimization conditions can be as important as semantics for compatibility between two data-passing schemes. Some optimization conditions may be *spatial,* restricting, for example, buffer location, alignment, or length. Other optimization conditions may be *temporal,* restricting, for example, when requests should occur or when a party may access its buffers. The spatial restrictions of migrant-mode data passing, explained in the "Buffer Allocation" section, and the temporal restrictions of weak-integrity data passing, explained in the section titled "Buffer Allocation", are intrinsic to the respective semantics and not special optimization conditions.

The *restrictiveness* of an optimization is the likelihood that an application not aware of the optimization will not meet the optimization's special conditions. *Hard* conditions are those that are met by practically no application not aware of the optimization. *Soft* conditions are those that are not hard.

The *criticality* of an optimization is the degree to which nonconformance with the optimization's conditions causes performance to worsen relative to the base case against which the optimization is claimed. At one end of the criticality spectrum are *mandatory* conditions, those that must be met for data passing to occur or that impose heavy penalties if not met. At the other end of the spectrum are *advisory* conditions, which, if not met, do not cause substantial penalty.

CONTROL-PASSING CHARACTERIZATION

API control passing can be classified according to how the API handles replies. Replies can be *final* or *pending.* A final reply indicates that the corresponding request succeeded or failed. A pending reply, on the contrary, indicates that the server will process the request and generate the corresponding final reply at some later time. In an *asynchronous* request, the API may return a pending reply to the client and let the client run while the server processes the request. The client later polls for the final reply. In a *synchronous* request, on the contrary, the API never returns a pending reply to the client. Synchronous requests can be blocking or nonblocking. When a server returns a pending reply to a *blocking* request, the API blocks the client (i.e., makes it nonrunnable) until the server generates the corresponding final reply. In contrast, when a server returns a pending reply to a *nonblocking* request, the API aborts the request and returns an error indication to the client.

Usually, a server generates a final reply when the server has actually completed processing the corresponding request. In some cases, however, a server may generate an *anticipated* final reply, instead of an *actual* one. The anticipated reply indicates that the server has checked the request and guarantees that the request will complete successfully at some later time. For example, a TCP/IP server may generate an anticipated reply to an output request after it has checked the specified connection and gained a reference to the output data, but well ahead of actually physically outputting the data. In case of an anticipated reply, client and server may execute in parallel, even if the request is synchronous.

EXAMPLES

In the Unix API (1), the two most common I/O calls are probably `read` and `write`:

```
ssize_t read(int d, void *buf, size_t nbytes)
ssize_t write(int d, const void *buf, size_t
  nbytes)
```

The explicit `read` and `write` calls can be used for very different types of I/O. The descriptor `d` may refer to objects as disparate as, for example, open files and network connections. `read` attempts to read `nbytes` of data from the object `d` into the application buffer pointed to by `buf`. Upon successful completion, `read` returns the number of bytes actually read and placed in the application buffer. In case of error, `read` returns -1. Conversely, `write` attempts to write `nbytes` of data to the object `d` from the application buffer pointed to by `buf`. `write` returns the number of bytes actually written or, in case of error, -1.

`read` and `write` usually pass data with copy semantics. An exception is raw file I/O (1), where data passing has share semantics (i.e., occurs directly between application buffers and the physical disk). Because copy and share semantics both have the same buffer allocation scheme (native), the same Unix calls can be used with either semantics, although with different buffer integrity guarantees.

`read` and `write` are by default blocking; however, control passing in I/O involving an object `d` can be converted to nonblocking by using the call:

```
int fcntl(int d, int cmd, int arg)
```

with `cmd` equal to `F_SETFL` (set status flag) and `arg` equal to `O_NONBLOCK`.

DASH (6) is an experimental system with an API that passes data with move semantics. In DASH, the calls `get_request` and `send_reply` synchronously input or out-

put messages. The corresponding asynchronous calls are `receive` and `send`. Messages are represented by a *header,* which may contain pointers to separate *data pages.* The application passes as an argument a pointer to the message header. `send_reply` and `send` unmap from the application's address space each data page specified in the message header. Conversely, to return a message to the application, `get_request` and `receive` map data pages to the application's address space and fill in corresponding pointers in the message header. The same calls could be used in an API with weak move semantics, except that data pages would then not actually be mapped or unmapped.

DATA-PASSING OPTIMIZATION

Most APIs pass data between client and server buffers by copying. On an output request, the system copies the data from application to system buffer. Output processing thereafter uses only the system buffer, and the application is free to reuse its buffer. Conversely, the system inputs data into system buffers. When returning a successful reply to an input request the system copies data from system to application buffers. Copying is flexible and convenient because it imposes no spatial or temporal conditions for data passing. Both client and server can specify the location and layout of the respective buffers and can access buffers before or after requests or replies without corruption of either party's data.

However, over the years, CPU performance and the bandwidth of certain I/O devices, such as high-speed networks, increased more rapidly than did memory bandwidth. Therefore, copying became relatively expensive, which motivates many proposals for data passing optimization.

The applicability of such proposals depends on whether they assume that server buffers are *ephemeral* or *cached.* An ephemeral buffer is one that is deallocated when the server completes processing the request that uses the buffer. On the contrary, a cached buffer is one that may remain allocated indefinitely after the completion of the first request that uses it. In Unix, for example, with few exceptions, buffers used in character I/O (including networking) (1) are ephemeral, whereas buffers used in block I/O (e.g., file I/O) (1) are cached.

The following two subsections discuss optimizations for ephemeral and cached server buffers, respectively, in the case of system calls. Optimizations for IPC are discussed in the final subsection.

System Calls with Ephemeral Server Buffers

If server buffers are ephemeral, copying can be avoided by using an API with noncopy semantics. More recent optimizations demonstrate, however, that copying can be avoided while preserving the copy semantics of conventional APIs. As explained in the following, optimized APIs with copy and noncopy semantics can provide comparable performance. However, optimized APIs with copy semantics can be less restrictive and less critical.

Noncopy Semantics. APIs with noncopy semantics normally pass data using VM manipulations instead of copying. For example, APIs with move semantics usually pass data by unmapping the pages containing the data and deallocating the corresponding memory region from the address space of party *a* and mapping those pages to a new memory region in the address space of party *b*. The pages carry the data without copying; however, party *a* cannot access the data after data passing, and party *b* cannot choose the location or layout of the data that it receives.

Likewise, APIs with share semantics can pass data inplace, without copying, by mapping client pages to the server's address space and making those pages unpageable during request processing. That is, client buffers are *promoted* to double as server buffers until request processing completion. However, clients must make input requests before input occurs and should not read input buffers or overwrite output buffers during request processing.

APIs with weak move semantics use buffers permanently comapped to client and server, which may only need to be made unpageable during request processing. Only the reference to a buffer is passed from party *a* to party *b*; no data copying or page remapping is necessary. However, party *a* should not access the data after data passing, and party *b* cannot choose the location or layout of the data.

For long data, the cost of data passing using VM manipulations typically is much less than it is using copying. Therefore, APIs with noncopy semantics can offer lower overhead than that of conventional APIs with copy semantics. However, APIs with noncopy semantics may have several problems:

1. *Incompatibility with Existing Applications.* Because existing applications often expect an API with copy semantics, it may be necessary to introduce a "compatibility library" that copies data between application buffers and buffers subject to the noncopy semantics of the new API.
2. *Incompatibility with New Applications.* Some applications, even if new or reimplemented, have requirements that conflict with intrinsic restrictions of noncopy semantics. For example, applications that are sensitive to data location or layout or that need access to output data after output requests would need to copy data between application buffers and buffers used for I/O using an API with move or weak move semantics. Likewise, applications that do not make input requests before input occurs or that access buffers during request processing may not benefit from APIs with share semantics.
3. *Lack of Hardware Support.* APIs with share or weak move semantics require *early demultiplexing* (11), that is, that data be input from a device directly to the corresponding client's buffers. Many devices, especially network adapters, do not have this capability: They allocate input buffers from a *pool* regardless of the data destination.

In all such cases, use of a new API with noncopy semantics may fail to reduce the total amount of data copying.

Outboard Buffering. In I/O where (1) the device controller is implemented outboard and has plenty of memory available, and (2) server(s) and driver do not need to access I/O data or can offload to the outboard controller all processing that requires such access, I/O data can be passed by DMA directly

between application buffers and outboard memory. In network I/O, this solution can be applied if the network adapter computes packet checksums (the only data-touching operation, aside from copying, typically performed by protocol stacks) (12,13).

Outboard buffering removes copying overhead from the host while preserving copy semantics. However, it also makes the controller more complex and costly. It can also increase I/O latency, given its "store-and-forward" architecture.

Emulated Copy. *Emulated copy* (3) is a recent copy-avoidance scheme that preserves copy semantics but does not require outboard buffering. Therefore, emulated copy can use controllers with "cut-through" architecture and achieve correspondingly lower latency.

For client input buffers, emulated copy preserves copy semantics using *input alignment,* that is, by inputting data into distinct server buffers that start at the same page offsets and have the same lengths as the respective client input buffers, as shown in Fig. 3. Emulated copy swaps pages between client and server buffers when returning the reply to the client, after server buffers have been successfully filled with input data. That is, for each pair of pages at the same offset from the start of the respective buffer, emulated copy invalidates all mappings of both pages, removes both pages from the respective memory object, inserts each page in the previous memory object of the other page, and maps each page to the virtual address and address space where the other page was mapped. Partially filled pages in the server buffer are handled as follows: If the data length is less than a configurable threshold t_i, emulated copy simply copies it out; otherwise, emulated copy completes the page with the complementary data of the corresponding page in the client buffer, using *reverse copyout,* that is, copying from client to server page, and then swaps pages. After swapping, the contents of client pages is the same as if data had been copied.

For client output buffers, emulated copy preserves copy semantics using *transient output copy-on-write* (TCOW). When the client makes a request, for each page in the client output buffer, if the data length is less than a configurable threshold t_o, emulated copy allocates a distinct system page and copies the data into it; otherwise, emulated copy removes write permissions from all mappings of the client page and increases the latter's *output reference count.* At request processing completion time, emulated copy respectively deallocates the system page or decreases the client page's output reference count. Client pages with nonzero output reference count serve as *in-place* system pages during request processing. Any attempt to overwrite such pages causes a page fault. Emulated copy modifies the system's page fault handler to guarantee that results are the same as if data had been copied to *distinct* system pages. The modification affects write faults on regions for which the faulted process has write permissions and the faulted page is found in the top memory object backing the region (14): If the page's output reference count is nonzero, the system recovers the process by invalidating all mappings of the page, copying the contents of the page to a new page, swapping pages in the memory object, and mapping the new page to the same virtual address in the process, with writing enabled. If the faulted page's output reference count is zero, the system recovers the process by simply reenabling writing on the page (no copying).

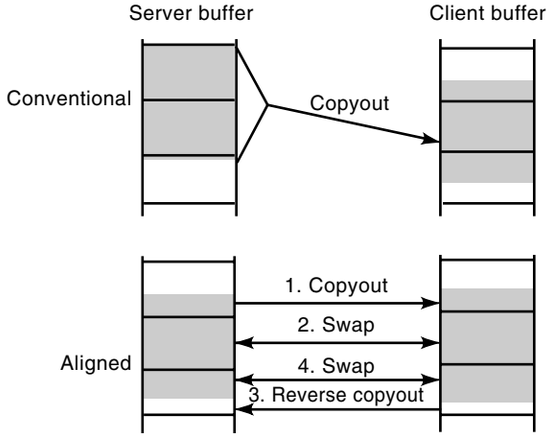

Figure 3. Conventionally, both client and server buffers are allocated without concern for alignment, and all data need to be copied. Input alignment enables page swapping.

Emulated copy uses *I/O-deferred page deallocation* to guarantee correct deallocation of client output pages only after request processing completion. The system's page deallocation routine is modified to refrain from placing pages with nonzero reference count in the list of free pages, where they might be reallocated to other processes. Emulated copy places a client output page in the list of free pages at request processing completion time if the page doesn't have any further references and no longer is allocated to a memory object.

Restrictiveness and Criticality of Emulated Copy. Input alignment can be achieved by *client-aligned* or *server-aligned* buffering, that is, respectively, by the client or server aligning its buffers with respect to the buffers of the other party. Client-aligned buffering imposes a spatial condition: Clients should lay out their buffers according to the *preferred alignment and length* informed by servers. In the case of network servers, for example, the preferred alignment would be the length of unstripped packet headers, while the preferred length would correspond to the network's maximum transmission unit. Server-aligned buffering, on the contrary, imposes a temporal condition: Clients should inform servers about the layout of client buffers *before* input physically occurs; servers then lay out their buffers accordingly. Server-aligned buffering also requires devices to have early demultiplexing (11). Note that the conditions for client-aligned and server-aligned buffering are similar to those implicit in input with migrant-mode and weak-integrity data passing, respectively.

In the terminology of the "Optimization Conditions" subsection, given that many client buffers (especially those allocated via `malloc`) are page-aligned and of length multiple of the page size, the condition for client-aligned buffering may be *soft* in cases of servers that have such preferred alignment and length. Because many existing applications already request input before input physically occurs, the condition for server-aligned buffering is also soft. Both server-aligned and client-aligned buffering impose only *advisory* conditions: With a properly tuned t_i, the cost of data passing is never greater than that of copying.

In cases of asynchronous requests or servers that return anticipated replies, TCOW imposes a temporal, soft condition:

It is more efficient not to overwrite a client output buffer until request processing completion. Note that this condition is similar to, but less restrictive than those implicit in output with migrant-mode or weak-integrity data passing.

The criticality of TCOW's condition may depend both on how buffers are overwritten and on the setting of t_o. Examination of existing applications reveals that often output buffers are overwritten not by the client itself, but by a server processing an input request on behalf of the client. For example, many applications input data, perhaps process the data, and then output the data using the same circular buffer alternately as input and output buffer. An informal analysis shows that TCOW and input alignment interact synergistically to eliminate copying in such cases. For the part of a client buffer that is page-aligned and has length multiple of the page size, it is easy to see that input alignment and page swapping will cause pages with outstanding output to be simply swapped out of the client buffer, with deallocation deferred until completion of the output request. No copying at all occurs for data output or input.

On the other hand, clients themselves (and not input servers on their behalf) may also overwrite buffers with outstanding output. In such case, compared to copying, TCOW with t_o equal to the page size gives output data-passing costs that are the same for pages only partially occupied by client buffers, and that are greater by the cost of swapping pages for fully occupied pages. If the cost of swapping pages is much less than that of copying a page, as is usual, then TCOW has low criticality even in this case. If the relative cost of copying is high, however, it may be desirable to optimize more aggressively, setting t_o less than the page size. Two alternative additional conditions can make TCOW's temporal condition still advisory even with such tuning, but at the cost of making TCOW more restrictive. The first condition is to require that a client, before overwriting an output buffer, make a synchronous *flush* request to the server, so as to ensure that processing of the previous output request is actually completed. The second, alternative condition is to have clients use a circular buffer, overwriting and synchronously outputting, successively at each time, only a fraction of size f of the buffer. The API allows the client to set a limit on the amount of physical memory in the client's pending I/O requests to a value less than the total size of the circular buffers by at least f. In that case, the fraction that is being overwritten at any given time is sure not to have pending output—the client would block on an output before it would have the opportunity to overwrite parts of the buffer with pending output.

Optimization Conditions versus Application Requirements. Applications may have requirements that are incompatible with the conditions for copy avoidance of the data-passing scheme used. Such applications may need to copy data between application data structures and application buffers used for I/O. The total amount of copying may remain the same as with conventional data passing, where the system copies between application data structures and system buffers.

For example, some distributed applications operate on matrices and may need to receive data from other hosts into specific matrix rows or columns. Copy avoidance may not be possible if the API has move or weak move semantics or uses emulated copy with client-aligned buffering, because then the application cannot choose the location and layout of the input data. If the application can post input requests before data are physically received, copy avoidance is possible using emulated copy with server-aligned buffering. Copy avoidance is also possible with outboard buffering.

Certain applications, for example, web proxies, need to cache data. Copy avoidance is not possible if the API has move or weak move semantics, because then output requests deplete the application's cache. Copy avoidance is possible, however, using emulated copy.

Applications that cannot post input requests before input physically occurs may have difficulty using APIs with share semantics. For example, an application may implement an `ftp` server by mapping files into memory regions (see the "System Calls with Cached Server Buffers" section) and inputting or outputting data directly between those regions and the network. Copy avoidance may not be possible on input with share semantics because, *before* making the input request, the application may need to decode an application-layer header that precedes the data and determines the correct file and corresponding memory region (i.e., input must already have occurred physically). Copy avoidance is possible, however, if the API uses emulated copy or outboard buffering. The application can then *peek* at the application-layer header before the actual input request. In the Unix API, for example, the following call could be used:

```
ssize_t recv(int d, void *buf, size_t nbytes,
  int flags)
```

`recv` is similar to `read` (see the "Examples" section), but can only be used on open *sockets* (network connections) and includes the extra argument `flags`. The flag `MSG_PEEK` causes the first `nbytes` to be copied to application buffer `buf` without consuming or deallocating the corresponding system buffers.

Applications that reuse output buffers may also have difficulty using APIs with share semantics. For example, an application may in a loop input video data, compress it, and output it over a network, always using the same buffer. Because the output request may return with an anticipated reply, video input may then corrupt the data being output. Copy avoidance without data corruption is possible using emulated copy: Input then uses a separate server buffer. When input completes, emulated copy swaps the buffer being used for output out of the application's address space.

Performance Comparison. This subsection reports measurements of end-to-end latency for datagram communication between applications running on separate computers connected by an ATM network at 155 Mbps. In the experiments, each computer had an Intel Pentium 166 MHz CPU and 32 Mbyte memory organized in 4 kbyte pages. The operating system used was NetBSD 1.1 augmented with an implementation of Genie (3), an experimental API that allows selection of data-passing schemes. The network adapter supported early demultiplexing. Conditions for copy avoidance according to each data-passing scheme were met.

Figure 4 shows the end-to-end latency for datagrams of length multiple of the page size, using different data-passing schemes. Only copying resulted in distinctly worse performance. Emulated copy provided latency similar to that of noncopy semantics.

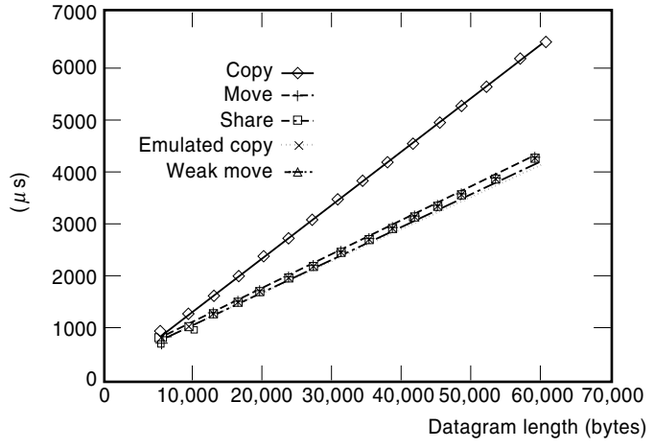

Figure 4. End-to-end latency. Emulated copy provides performance improvements similar to those of noncopy semantics.

Figure 5 shows the corresponding measurements for datagrams shorter than a page, with thresholds t_o = 1666 bytes and t_i = 2178 bytes. Move semantics gave the highest latency for short data because move semantics maps whole pages to the receiving application and, to preserve protection, the part of the page not filled with input data has to be filled with zero before mapping. Copy semantics gave the lowest but also the most rapidly rising latency because of the high incremental cost of copying. Emulated copy had about the same latency as that of copying for data up to a half-page long; above that, reverse copyout and page swapping significantly reduced the latency of emulated copy relative to that of copying.

System Calls with Cached Server Buffers

Servers of storage-related I/O (e.g., file I/O) often cache the buffers of previous requests. Servers can use such buffers to avoid accessing secondary storage (e.g., disks). Because secondary storage devices often are very slow, caching can improve response times by orders of magnitude.

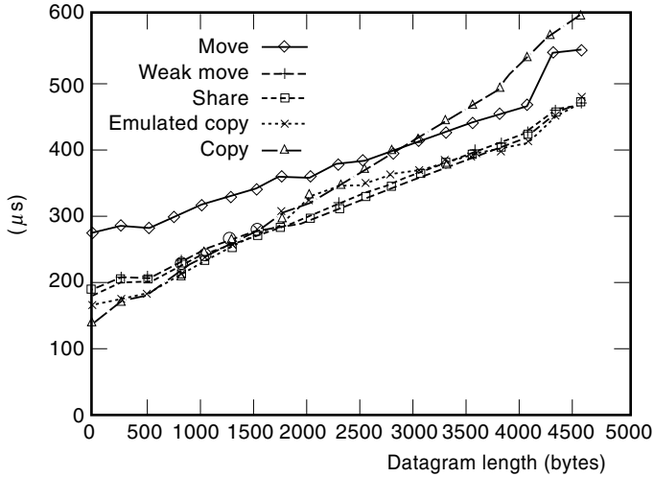

Figure 5. End-to-end latency for short datagrams. Using reverse copyout, emulated copy avoids copying more than about half a page.

Most contemporary systems offer two APIs for storage-related I/O: (1) *explicit* and (2) *mapped*. The explicit API is typified by Unix's `read` and `write` system calls. The mapped API, on the other hand, is typified by Unix's `mmap` and `munmap` system calls. For network-related and other forms of I/O with ephemeral server buffers, usually only the explicit API is available.

The optimizations discussed in the previous subsection are for explicit APIs. Most of those optimizations assume that server buffers are ephemeral and may be unsuitable for cached server buffers. Explicit APIs with move or weak move semantics, for example, transfer input buffers from servers to clients, and therefore do not allow servers to cache those buffers. Moreover, explicit APIs with weak move semantics may leave client output buffers both client- and server-accessible after request completion, and therefore enable caching by servers but also overwriting by clients, with consequent server cache corruption. Explicit APIs with share semantics may make client buffers server-accessible only during request processing, and therefore may not allow servers to cache them.

Emulated copy is also inappropriate for cached server buffers. First, emulated copy makes client output buffers immune from overwriting only during request processing, that is, while buffer pages have nonzero output reference count. Therefore, those buffers cannot be cached by servers. Second, emulated copy swaps pages on input request completion, corrupting the contents of server buffers with the previous contents of the client buffers. Such corruption is inconsequential if server buffers are ephemeral because ephemeral buffers are deallocated on request completion. Cached server buffers, on the contrary, should be preserved after request completion and therefore do not allow page swapping.

Although copy avoidance with cached server buffers is difficult using explicit APIs, the mapped API lends itself easily to a copy-free implementation. The mapped API allows clients to map a file (or part of it) to a region in the client's address space. In the explicit API, the request to map a file is equivalent to that of allocating a new region and inputting file data into it. Likewise, the request to unmap a file is equivalent to that of outputting the region's data to the file and deallocating the region. (This description corresponds to a file mapped in *shared* mode. If multiple clients map the file in *shared* mode, the region is shared among them, and the output of the region's data to the file occurs when the last such client unmaps the file. It is often also possible to map a file in *private* mode, in which case the region's data are not output back to the file.)

In the Unix API, `mmap` and `munmap` have the following syntax:

```
caddr_t mmap(caddr_t addr, size_t len, int
  prot, int flags, int fd, off_t offset)
  int munmap(caddr_t addr, size_t len)
```

`mmap` causes the pages starting at `addr` and continuing for at most `len` bytes to be mapped from the object described by `fd` (e.g., file descriptor), starting at byte offset `offset` (from the beginning of the file). `addr` is only a hint; `mmap` returns the actual address of the mapped region. Access permissions to the region (read, write, and/or execute) are set by `prot`. `flags` may specify the `MAP_SHARED` or `MAP_PRIVATE` modes. `munmap` deletes the mappings for the specified address range

and causes further references to addresses within the range to be invalid.

The mapped API passes data between client and file server by mapping or unmapping cache pages to or from the client's address space. Using VM techniques, the API may map each page on an exception basis, that is, only when the client actually accesses that page. No copying is necessary if cache and VM pages are allocated from the same pool, as they are in many contemporary systems. (For mapping in *private* mode, cache pages can be mapped copy-on-write to the client's address space.) If cache and VM pools are separate, however, as they originally were in Unix (1), it may be necessary to copy data between pages from each pool.

IPC

The previous subsections assume that clients interact with servers using system calls. This is appropriate for monolithic systems, such as Unix (1), which integrate most servers in the kernel. In contrast, microkernel systems, such as CMU's Mach (4), implement most servers as separate user-level processes. User-level servers are easier to debug and maintain and provide greater fault isolation than do kernel-level ones. However, user-level servers interact with clients using IPC, not system calls. This subsection examines data-passing optimizations for IPC.

In order to support existing applications directly, the IPC facility should offer a client API with copy semantics. Unfortunately, IPC facilities with copy semantics can have high overhead. Such facilities often copy data twice, once between each party's and system buffers, as illustrated in Fig. 6.

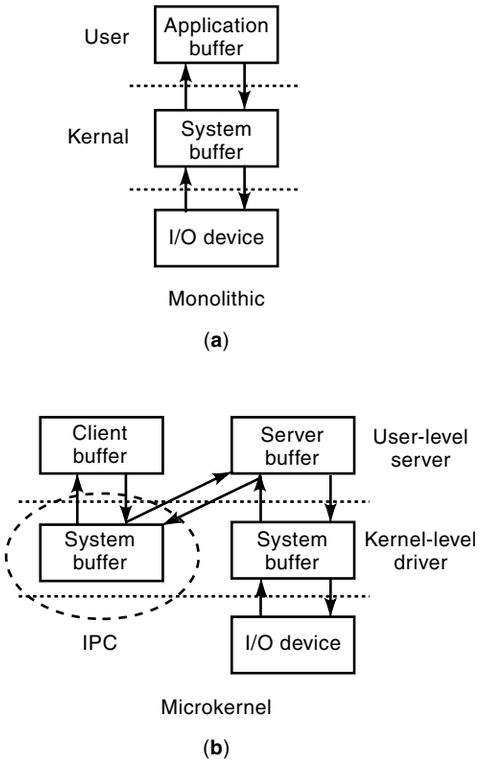

Figure 6. I/O data crosses the user/kernel boundary multiple times in microkernel systems.

L4 (15,16) reduces the number of copies to one (the same as in conventional system calls). To make this possible, L4 maps the buffers of party *a* to the address space of party *b*, copies the data, and unmaps *a*'s buffers from *b*'s address space. This optimization can be used whether server buffers are ephemeral or cached.

All copying can be avoided in cases of outboard buffering (which only applies for certain servers and devices, as explained in the "Outboard Buffering" subsection) and mapped APIs (which only applies for cached server buffers, as explained in the "System Calls with Cached Server Buffers" subsection). Data passing in such cases does not depend on whether servers are implemented at kernel or user level.

All copying can also be avoided if the IPC facility offers a client API with noncopy semantics and server buffers are ephemeral. The IPC facility can then simply move or share pages between the client and a server implemented at user level, instead of kernel level. The DASH (6) and Container Shipping (8) IPC facilities, for example, have APIs with move semantics. On the other hand, the fbuf IPC facility (9) combines several optimizations. Cached fbuf output has semantics similar to that of emulated copy, but leaves buffers read-only until explicit deallocation. Cached volatile fbuf output has share semantics. Cached and cached volatile fbuf input have semantics similar to weak move but use read-only buffers that must be deallocated explicitly.

Unfortunately, such IPC facilities do not directly support many existing applications, which expect copy semantics. The Peregrine (17) IPC facility offers a compromise, with distinct APIs for client and server. The client API has copy semantics, while the server API has move semantics. Data passing between client and system buffers is by copy-on-write (output) and copying (input). Between system and server buffers, data passing is by page mapping and unmapping.

I/O-oriented IPC (18) is a recent proposal that combines a client API with copy semantics *and* bidirectional copy avoidance. Data passing between client and system buffers is by emulated copy (see the "Emulated Copy" subsection). Data passing between system and server buffers is by mapping and unmapping. The greatest novelty with respect to Peregrine is the use of input alignment and page swapping for input. Both Peregrine and I/O-oriented IPC assume ephemeral server buffers.

CONTROL-PASSING OPTIMIZATIONS

Unlike the overheads of data passing, those of control passing typically do not grow with data length. Consequently, control-passing overheads have greatest impact for short data; they are amortized and become less significant for long data. Therefore, applications may be able to optimize control passing and improve throughput by *aggregating* many short-data requests into fewer long-data requests.

Applications may also be able to improve throughput by *overlapping* computations and I/O requests. Such overlap may be implemented using (1) multiple application threads and blocking I/O requests, or (2) a single application thread and multiple asynchronous or nonblocking I/O requests. Blocking requests are easier to program, but their requirement of multiple application threads may increase context switching and, therefore, control passing overheads relative

to asynchronous or nonblocking requests (19). Mainstream APIs usually support both blocking and some form of nonblocking or asynchronous I/O. BSD Unix's API (1), for example, is by default blocking, but also enables nonblocking I/O.

Although request aggregation and overlap can improve throughput, they do not improve latency, that is, they do not reduce the time between a given request and the corresponding request processing completion and reply. For latency improvement, operating system optimizations are necessary. Latency improvements often also result in better throughput.

Researchers have demonstrated that careful design and implementation can reduce the latency of system calls (20) and IPC (15,20) by roughly an order of magnitude. These optimizations can, in principle, be integrated into existing systems. Further latency reductions are possible by more radically changing the *structure* of the operating system, so as to make IPC or system calls unnecessary. Proposed optimizations include:

1. *Moving User-Level Servers into the Kernel.* This modification can be applied to microkernel systems and was used, for example, in Microsoft Windows NT 4.0 (21). It replaces IPC by typically much cheaper system calls. Unfortunately, the benefits of the microkernel structure are also lost: Kernel-level servers are harder to debug and maintain and do not have the fault isolation of servers implemented as separate user-level processes.
2. *Server Decomposition.* This modification can be applied to microkernel systems. Device drivers are moved into the kernel and each remaining user-level server (e.g. TCP/IP server) is decomposed into a *fast-path* and a *slow-path* component (22,23). The fast-path component is linked as a library with applications and ideally processes common-case I/O requests without communicating with the slow-path component. The latter remains a user-level server implemented by a separate process and ideally handles only exceptions. In the ideal case, therefore, IPC is avoided and control-passing overhead is reduced to that of a monolithic system (i.e., system call). However, IPC is avoided only if the fast-path component does not depend on or modify global server state. This was found to be possible for a TCP/IP server (22,23), but not for a file server (24). Additionally, decomposition can result in servers that are more complex and difficult to debug and maintain than the original servers.
3. *Operating System (OS) Bypass.* This alternative adds to the previous item, server decomposition, hardware modifications that allow also device drivers to be decomposed (25–27). The fast-path driver is linked as a library with applications and ideally allows common-case I/O to be performed without any IPC or system calls. Achieving this goal may be hard for reasons similar to those mentioned in the previous item. For example, for quality of service guarantees, it may be necessary to schedule requests globally. That, however, would be unsafe in libraries linked with untrusted applications. Additionally, OS bypass usually requires applications to use specially allocated buffers with special semantics, which may create incompatibilities with existing applications; and the special hardware requirements may generate portability problems.
4. *Extensible Kernel Systems.* Extensible kernel systems allow applications to download *extensions* (application-specific code) into the kernel. Because I/O servers also run in the kernel, extensions can perform I/O without IPC or system calls. Several techniques have been used to make extensions safe. SPIN (28) requires extensions to be written in a type-safe language, and VINO (29) encapsulates extensions for software fault isolation. These techniques have been reported to make code run from 10% to 150% more slowly (30). Proof-carrying code (31) offers the promise of eliminating such overheads, but has not yet been demonstrated to be practical for large extensions. The API for extensions is also unconventional, which may cause portability problems.

BIBLIOGRAPHY

1. S. Leffler et al., *The Design and Implementation of the 4.3BSD UNIX Operating System,* Reading, MA: Addison-Wesley, 1989.
2. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* 2nd ed., San Francisco: Morgan Kaufmann, 1996.
3. J. C. Brustoloni and P. Steenkiste, Effects of buffering semantics on I/O performance, *Proc. OSDI'96,* USENIX, 1996, pp. 277–291.
4. D. Golub et al., Unix as an application program, *Proc. USENIX Summer Conf.,* USENIX, 1990.
5. Anonymous, *UNIX System V Release 4 Network Programmer's Guide,* AT&T, 1991.
6. S.-Y. Tzou and D. Anderson, The performance of message-passing using restricted virtual memory remapping. *Softw. Pract. Exp.,* **21** (3): 251–267, 1991.
7. O. Krieger and M. Stumm, The alloc stream facility, *Computer,* **27** (3): 75–82, 1994.
8. J. Pasquale, E. Anderson, and P. Muller, Container Shipping: Operating system support for I/O-intensive applications, *Computer,* **27** (3): 84–93, 1994.
9. P. Druschel and L. Peterson, Fbufs: A high-bandwidth cross-domain transfer facility, *Proc. 14th SOSP,* ACM, 1993, pp. 189–202.
10. J. C. Brustoloni, Exposed buffering and sub-datagram flow control for ATM LANs, *Proc. 19th Conf. Local Comput. Netw.,* IEEE, 1994, pp. 324–334.
11. J. C. Brustoloni and P. Steenkiste, Copy emulation in checksummed, multiple-packet communication, *Proc. INFOCOM'97,* IEEE, 1997, pp. 1124–1132.
12. C. Dalton et al., Afterburner, *Network,* 36–43, 1993.
13. K. Kleinpaste, P. Steenkiste, and B. Zill, Software support for outboard buffering and checksumming, *Proc. SIGCOMM'95,* ACM, 1995, pp. 87–98.
14. R. Rashid et al., Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures, *Proc. 2nd ASPLOS,* ACM, 1987, pp. 31–39.
15. J. Liedtke, Improving IPC by kernel design, *Proc. 14th SOSP,* ACM, 1993, pp. 175–188.
16. H. Härtig et al., The performance of μkernel-based systems, *Proc. 16th SOSP,* ACM, 1997, pp. 66–77.
17. D. B. Johnson and W. Zwaenepoel, The Peregrine high-performance RPC system, *Softw. Pract. Exp.,* **23** (2): 201–221, 1993.
18. J. C. Brustoloni and P. Steenkiste, User-level protocol processing with kernel-level performance, *Proc. INFOCOM'98,* IEEE, 1998, pp. 463–471.

19. P. Druschel, V. Pai, and W. Zwaenepoel, Extensible kernels are leading OS research astray, *Proc. HotOS-VI,* IEEE, 1997, pp. 38–42.
20. D. Engler, M. F. Kaashoek, and J. O'Toole, Jr., Exokernel: An operating system architecture for application-level resource management, *Proc. 15th SOSP,* ACM, 1995, pp. 251–266.
21. Anonymous, Getting to the kernel: Is NT still safe?, *Byte,* 130, 1996.
22. C. Thekkath et al., Implementing network protocols at user level, *Proc. SIGCOMM'93,* ACM, 1993.
23. C. Maeda and B. Bershad, Protocol service decomposition for high-performance networking, *Proc. 14th SOSP,* ACM, 1993, pp. 244–255.
24. C. Maeda, Service Decomposition: A Structuring Principle for Flexible, High-Performance Operating Systems, Ph.D. Thesis, CMU-CS-97-128, School of Computer Science, Pittsburgh, PA: Carnegie Mellon University, April 1997.
25. M. Blumrich et al., Virtual memory mapped network interface for the SHRIMP multicomputer, *Proc. 21st Annual Int. Symp. Comp. Arch.,* IEEE/ACM, 1994, pp. 142–153.
26. P. Druschel, L. Peterson, and B. Davie, Experience with a high-speed network adaptor: A software perspective, *Proc. SIGCOMM'94,* ACM, 1994, pp. 2–13.
27. T. von Eicken et al., U-Net: A user-level network interface for parallel and distributed computing, *Proc. 15th SOSP,* ACM, 1995, pp. 40–53.
28. B. Bershad et al., Extensibility, safety and performance in the SPIN operating system, *Proc. 15th SOSP,* ACM, 1995, pp. 267–284.
29. M. Seltzer et al., Dealing with disaster: surviving misbehaved kernel extensions. *Proc. OSDI'96,* pp. 213–227, USENIX, Oct. 1996.
30. C. Small and M. Seltzer, A comparison of OS extension technologies. *Proc. USENIX Annual Conf.,* pp. 41–54, USENIX, Jan. 1996.
31. G. Necula and P. Lee, Safe kernel extensions without run-time checking. *Proc. OSDI'96,* pp. 229–243, USENIX, Oct. 1996.

Reading List

T. Anderson et al., The interaction of architecture and operating system design, *Proc. 4th ASPLOS,* ACM, 1991, pp. 108–120.

J. C. Brustoloni and P. Steenkiste, Evaluation of data passing and scheduling avoidance, *Proc. NOSSDAV'97,* IEEE, 1997, pp. 101–111.

H. J. Chu, Zero-copy TCP in Solaris, *Proc. USENIX Annu. Conf.,* USENIX, 1996.

R. Dean and F. Armand, Data movement in kernelized systems, *Proc. Workshop Microkernels Other Kernel Architectures,* USENIX, 1992.

K. Fall and J. Pasquale, Exploiting in-kernel data paths to improve I/O throughput and CPU availability, *Proc. USENIX Winter Conf.,* USENIX, 1993, pp. 327–333.

K. Fall and J. Pasquale, Improving continuous-media playback performance with in-kernel data paths, *Proc. 1st Int. Conf. Multimedia Comput. Syst.,* IEEE, 1994, pp. 100–109.

M. F. Kaashoek et al., Application performance and flexibility on exokernel systems, *Proc. 16th SOSP,* ACM, 1997, pp. 52–65.

J. Ousterhout, Why aren't operating systems getting faster as fast as hardware?, *Proc. USENIX Summer Conf.,* USENIX, 1990, pp. 247–256.

JOSÉ CARLOS BRUSTOLONI
Bell Laboratories

PETER STEENKISTE
Carnegie Mellon University

APPLICATIONS OF RADAR. See RADAR APPLICATIONS.

APPLICATION-SPECIFIC INTEGRATED CIRCUITS. See LOGIC ARRAYS.

APPROXIMATION METHODS FOR FILTERS. See FILTER APPROXIMATION METHODS.

APPROXIMATION OF LINEAR SYSTEMS. See LINEAR DYNAMICAL SYSTEMS, APPROXIMATION.

APPROXIMATIONS. See FILTERING THEORY; LOW-PASS FILTERS.

ARBITRATION. See CONTRACTS.