User-Level Protocol Servers with Kernel-Level Performance

José Carlos Brustoloni*
Bell Labs
Lucent Technologies
jcb@research.bell-labs.com

Peter Steenkiste
School of Computer Science
Carnegie Mellon University
prs@cs.cmu.edu

Abstract—**Compared to kernel-level servers, user-level ones can be debugged and maintained more easily and safely, but traditionally have had much worse performance. We describe a novel *I/O-oriented* IPC facility that combines the *emulated copy* data passing scheme for monolithic systems with new copy avoidance techniques for microkernel systems. Unlike previous optimizations, I/O-oriented IPC does not require changes in existing user applications or complex restructuring of servers; it offers an API with copy semantics and allows the same servers to be installed at kernel or user level. In end-to-end experiments on an ATM network at 512 Mbps, I/O-oriented IPC gave user-level protocol servers performance approaching that of kernel-level ones. Performance differences scaled roughly inversely to the processor's SPECint95 rating, projecting fast further improvement.**

I. Introduction

MULTIUSER systems typically do not give user applications direct access to global code or state of communication protocols or to network adapters. Such accesses could allow, for example, unauthorized users to intercept or forge other users' packets or disable communications.

To prevent such tampering, monolithic systems, such as Unix [12], integrate protocol servers and network drivers in the kernel. User applications can access networks only as *clients* that *request*, using system calls, services from appropriate kernel-level *servers*, such as protocol servers and network drivers. Servers process each request and return a *reply* to the respective client.

Microkernel systems, such as Mach [8], achieve similar protection by running servers each as a separate user-level process with its own address space. User-level clients and servers pass requests and replies to each other using an inter-process communication (IPC) facility.

Because the control and data passing overheads of IPC typically have been much higher than those of system calls, kernel-level servers have been far more common than user-level ones in high-speed networking. IPC's control passing normally requires, in addition to a system call, rescheduling and switching context to another process. Data passing across the user/kernel boundary in many systems, including Unix, is by copying, which levies relatively large [15] and increasing [9] overheads. Such overheads may be multiplied in IPC, where data crosses the user/kernel boundary more than once, as illustrated in Figure 1.

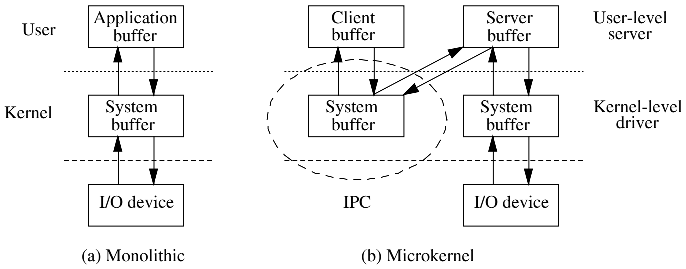

Fig. 1. I/O data crosses the user/kernel boundary multiple times in microkernel systems.

Although kernel-level servers are more common, they do have disadvantages relative to user-level ones. Kernel-level debuggers are usually much more primitive than user-level ones, typically supporting only assembly language and arresting the whole system during a debug session. Additionally, any bugs (or malicious modifications) in a kernel-level server can severely compromise the integrity or protection of the whole system. Therefore, only the system administrator should be allowed to install, debug, or modify kernel-level servers. In contrast, user-level servers can be debugged and modified as easily and safely as any other user application; users may be allowed to install, debug, customize, or modify their own user-level servers.

This paper contributes a novel *I/O-oriented* IPC facility that is the *first* to give user-level servers performance approaching that of kernel-level ones without requiring changes in existing user applications or complex restructuring of servers. I/O-oriented IPC combines the following advantages relative to previous proposals:
1. *Preservation of API with copy semantics* — Many prior optimized IPC facilities, e.g. DASH [22], fbufs [6], and container shipping [16], change the *semantics* of data passing [3], e.g. to *move* or *share* semantics rather than the usual *copy* semantics of the application programming interfaces (APIs) of systems such as Unix [12] and Windows NT [5]. Such modified semantics allow data passing without copying and therefore reduce data passing overheads, but also cause incompatibilities with previously written applications, which expect copy semantics [3]. I/O-oriented

*Work performed while at the School of Computer Science, Carnegie Mellon University.

This research is sponsored in part by the Defense Advanced Research Projects Agency under contract F19628-92-C-0116 and under contract N66001-96-C-8528 monitored by NRaD.

IPC, on the contrary, avoids copying while preserving an API with copy semantics.

2. *Low restrictiveness and criticality* — Data passing optimizations may impose special optimization conditions that cause incompatibility with previously written applications. Some conditions may be, for example, *spatial*, restricting the location of client buffers, or *temporal*, restricting when a client may access its buffers. The *restrictiveness* of an optimization is the likelihood that an application not aware of the optimization will not meet the optimization's 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 non-conformance with the optimization's conditions worsens performance relative to the base case against which the optimization is claimed. *Mandatory* conditions are those that must be met for data passing to occur or that impose heavy penalties if not met. *Advisory* conditions are those that are not mandatory.

The fbufs facility [6], for example, imposes several hard, mandatory conditions that make it incompatible with previously written applications: Fbufs must be located in a special fbuf region; clients cannot write on input buffers; after cached fbuf output requests, clients cannot write on output buffers. In contrast, I/O-oriented IPC imposes only soft, advisory conditions, which preserve compatibility.

3. *Preservation of server organization and interfaces* — Some previous works [13], [21] have avoided IPC by installing drivers in the kernel and decomposing each remaining user-level server into a *fastpath* component, which is linked as a library with user applications and processes common input and output requests, and a *slowpath* component, which remains a separate user-level server and ideally processes only exceptions.

Unfortunately, depending on the server, it can be quite hard to extract a fastpath component that does not depend on or modify global server state and is self-sufficient for input and output requests. For example, [14] has shown that such extraction was not possible for a file system. Moreover, server organization and interfaces (and, consequently, design, implementation, debugging, and maintenance) become considerably more complex than those of comparable monolithic or microkernel servers. On the contrary, I/O-oriented IPC preserves server organization and interfaces: The same servers can be installed, without modifications, at kernel or user level.

I/O-oriented IPC provides a valuable tool for protocol development and experimentation in high-speed networking: After a protocol server is debugged or customized at user level, it may be placed back in the kernel. However, our experiments on the CreditNet ATM network at 512 Mbps, using Pentium Pro 200 MHz PCs, demonstrate user-level performance that may be deemed already acceptable for production purposes. Moreover, our scaling tests suggest that performance differences between kernel- and user-level servers using I/O-oriented IPC are shrinking inversely to the processor's SPECint95 benchmark, projecting over 50% further improvement per year [9].

The rest of this paper is organized as follows. Section II gives an overview of the design of the I/O-oriented IPC facility, and the following five sections describe individual components of the design: emulated copy, selective transient mapping, user-to-user input alignment and page swapping, fragment subcontracting, and buffer management. Section VIII presents our experimental results, and Section IX discusses related work. Finally, Section X summarizes and concludes.

II. I/O-oriented IPC

The design of I/O-oriented IPC follows the client/server model discussed in the Introduction: To preserve protection, user applications can perform I/O only as clients that request I/O services from authorized servers; servers process each request and return a reply to the respective client, as illustrated in Figure 2. Client and server are also called the *parties* to a request.

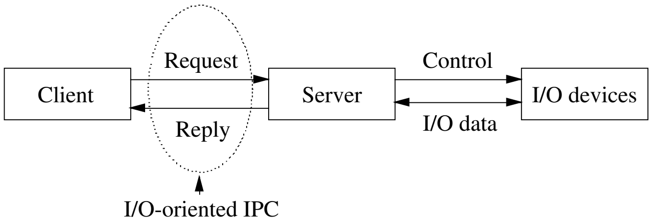

Fig. 2. Client-server decomposition for I/O protection.

If processing of the client does not continue until the server replies, the request is *synchronous*; otherwise, the request is *asynchronous*. When a server can guarantee that it will carry out a request successfully, it may return an *anticipated reply*. For example, a TCP/IP server may return a successful reply to an output request as soon as it has checked request parameters (e.g., connection) and has a reference to the request data, but well ahead of actually physically outputting the data. In cases of anticipated reply, no reply occurs when request processing actually completes.

A server may *subcontract* another server in order to process a given request. For example, a TCP/IP protocol server may subcontract an ATM driver to send or receive data, as shown in Figure 3. Subcontracting allows modular code reuse: Different protocol servers, for example, may subcontract any network drivers. Network drivers access adapters directly and, therefore, normally do not subcontract other servers.

In a subcontract, a *contractor* (server acting as a client) makes a request to a *subcontractor* (another server), which processes the request and returns a reply to its client. The requests made by a server to process a given request are called the latter's *derived requests*. A request that is not derived of any other is an *original request*. A *subcontract*

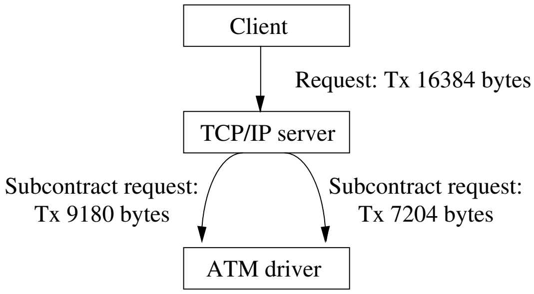

Fig. 3. Subcontract between protocol server and network driver.

graph is a directed acyclic graph that links each request to its pending derived requests.

For each request and reply, I/O-oriented IPC passes control and data from one party to the other. Data is held in buffers; the buffers used by each party and the IPC facility itself are called *client*, *server*, and *system* buffers, respectively. System buffers are always resident in physical memory and accessible at kernel level. I/O-oriented IPC uses different schemes to pass data between system buffers and user-level client, server, and contractor buffers:

1. Data passing between client and system buffers is in every respect like data passing between application and system buffers in a monolithic system, where kernel-level servers directly access system buffers. I/O-oriented IPC uses the *emulated copy* [3] scheme to avoid copying while preserving copy semantics.
2. I/O-oriented IPC avoids copying between system and server buffers by selectively mapping system buffers to servers' address spaces during request processing.
3. In subcontracts, I/O-oriented IPC avoids data passing between contractor and system buffers if data is already in a original request's system buffers.

Consequently, data passing is asymmetric in I/O-oriented IPC. Data passing for client buffers is with copy semantics, preserving compatibility with existing applications. Data passing for server buffers, on the contrary, mimics schemes traditionally used at kernel level, propitiating uniform interfaces and easy server migration between kernel and user level. Sections III, IV, and VI describe each of the data passing schemes, and Section VII discusses buffer management.

III. Emulated copy

Emulated copy allows I/O-oriented IPC to preserve an API with copy semantics while avoiding copying. In [3], we demonstrated experimentally that emulated copy performs almost as well as or better than other schemes for data passing between client and system buffers, including schemes with move or share semantics. In this section, we briefly review the implementation of emulated copy and then show that emulated copy has low restrictiveness and criticality. Readers familiar with emulated copy may skip to Section III-B.

A. Operating system support

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. 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. Note that, after swapping, the contents of client pages, but not of server pages, is the same as if data had been copied. Note also that the server can be implemented at kernel or user level.

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 [17]: 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).

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, whence 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.

B. Restrictiveness and criticality

Input alignment can be achieved by *server-aligned* or *client-aligned* buffering. We discussed network adapter support for both in [4]. Server-aligned buffering 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. Client-aligned buffering, on the contrary, imposes a spatial condition: Clients should lay out their buffers according to the *preferred alignment and length* informed by servers.

Because many existing applications already request input before input physically occurs, the condition for server-aligned buffering is soft. On the other hand, 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 also be soft in cases of servers that have such preferred alignment and length. 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. The criticality of this condition may depend both on how buffers are overwritten and on the setting of t_o, as discussed in the next subsection.

C. Synergy

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 that 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.

Analysis for pages only partially occupied by the client buffer is more complicated but provides valuable insight. If l is the data length, p is the page size, the cost of copying is cl, and the cost of swapping pages is s, then the cost of output with TCOW is: for $l < t_o$, cl, and for $l \geq t_o$, ct_o. The cost of input with input alignment is: for $l < t_i$, cl, and for $l \geq t_i$, $c(p-l)+s$. Therefore, the combined cost of output and input with TCOW and input alignment is:

1. $l < t_o$ and $l < t_i$: $2cl$
2. $l < t_o$ and $l \geq t_i$: $cp + s$
3. $l \geq t_o$ and $l < t_i$: $c(t_o + p + l) + s$
4. $l \geq t_o$ and $l \geq t_i$: $c(t_o + p - l) + s$

Substituting above $t_i = \frac{\frac{s}{c}+p}{2}$, and considering that the cost of input and output by copying is $2cl$, it is easy to verify that TCOW and input alignment, compared with copying, break even in the first condition, give lower cost in the second and fourth conditions, but give higher cost in the third condition ($t_o \leq l < t_i$: Output makes page read-only and input by copyout causes a write fault). The third condition can easily be avoided by requiring that t_o be tuned with $t_o \geq t_i$.

Alternatively, it is also clear from the equations that when output is with copy avoidance (third and fourth conditions), the total cost is always less when input is effectively by reverse copyout and page swapping (fourth condition). Therefore, input alignment should be modified as follows: If the client buffer page has an outstanding output reference, then input should be by reverse copyout and page swapping and never by copyout, regardless of the data length. With this modified input alignment rule, TCOW and input alignment will give higher cost than that of copying only for $t_o \leq l < \frac{t_o+2t_i}{3}$.

Therefore, in the common cases where clients do not overwrite output buffers during output or do so by reusing them as input buffers, TCOW has low criticality.

In the remaining case, clients themselves (and not input servers on their behalf) overwrite buffers with outstanding output. In such case, compared to copying, TCOW with $t_o = p$ gives output data passing costs that are the same for pages only partially occupied by client buffers, and that are greater by s for fully occupied pages. If $s \ll cp$, 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 $t_o < p$. 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 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 client sets its limit on amount of physical memory in pending I/O requests (as explained in Section VII) 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.

IV. Selective transient mapping

To pass data between system and server buffers with copy avoidance and the same interface for kernel- and user-level servers, I/O-oriented IPC uses *selective transient mapping*. When delivering a request to a user-level server, I/O-oriented IPC maps the request's system buffers to the server, and, when the server notifies request processing completion, the IPC facility unmaps those buffers from the server. Mapping is *transient*: a user-level server can access a request's system buffers only while processing the request.

Mapping is also *selective*. When a user installs a user-level server, the user specifies *read*, *write*, and *physical* mapping flags for each service provided by the server. I/O-oriented IPC uses these flags as follows:

- Services that do not require access to I/O data and that are implemented by subcontracting may have no mapping flags set. In such cases, the IPC facility passes only the length of each buffer to the server.
- If the physical flag is set, the IPC facility passes the physical addresses of each buffer to the server. This option may be sufficient for user-level drivers of devices that transfer data by DMA, in cases where the driver itself does not need to access I/O data.
- If the read or write flag is set, the IPC facility maps the buffers and passes their virtual addresses to the server. Output buffers are mapped read-only. Input buffers are mapped with access permissions according to the read and write flags.

The read flag is necessary for any service that requires direct access to I/O data. The write flag is normally required only for input services of drivers of programmed-I/O devices. DMA devices usually can write into memory regardless of the access permissions of their drivers, and servers other than drivers typically modify only headers or trailers. As explained in Section VI, headers and trailers can be freshly allocated and prepended or appended to fragments of buffers received in requests without modifying those buffers.

I/O-oriented IPC maps system buffers to *transient mapping regions* which, to minimize virtual address calculations, start at the same virtual address and have the same length in all user-level servers. The length is equal to the global limit on the amount of physical memory in pending I/O requests (explained in Section VII).

I/O-oriented IPC maintains a stack of *transient virtual addresses* (TVAs), each corresponding to an unassigned page-aligned address in the transient mapping region. When first mapping a system buffer page to a user-level server, the IPC facility pops a TVA and assigns it to the page. In subsequent subcontracts, all transient mappings of the page use the same virtual address. The IPC facility pushes the TVA back when unmapping the page from the initial user-level server.

Because system buffers are unpageable, I/O-oriented IPC can reduce mapping and unmapping overheads by updating only the server's page table, as opposed to also updating a machine-independent representation of the server's address space. The system's VM fault handler is modified to treat faults in the transient mapping region as unrecoverable.

To preserve protection, before mapping to a user-level server a system page that is not in-place, I/O-oriented IPC completes with zero those parts of the page not yet filled with zero or with data of the request's client. In case of emulated copy, system pages that are not in-place occur on output (when the data length is less than t_o) or input (any data length). Such system pages may be allocated from the pool of free pages of the virtual memory (VM) subsystem and contain data of clients who do not trust the particular server to which the page needs to be mapped.

Because zero completion can be expensive, I/O-oriented IPC may pass data to and from user-level servers by copying, instead of mapping and unmapping. Let t_m be a configurable threshold for selective transient mapping. The IPC facility maintains separate pools of free pages mapped to each user-level server. The IPC facility may, when delivering a request to a server s, instead of zero-completing and mapping a system page p to s, assign to p a page p' allocated from the pool of s. Such cases occur when the data length in p is less than t_m; I/O-oriented IPC then passes to s the virtual address of p', rather than the mapped address of p. If the requested service has the *read* flag set, the IPC facility copies data from p to p' when delivering the request. If the requested service has the *write* flag set, the IPC facility copies data from p' to p when s notifies request processing completion. The IPC facility also deallocates p' at request processing completion.

V. User-to-user copying and input alignment

To avoid both zero-completion and copying costs, I/O-oriented IPC allows clients to specify the *lender* of each system buffer that is not in-place. If a lender is not specified, the IPC facility assumes the lender to be the server of the client's request. The IPC facility allocates system buffers, if the lender is a user-level server, from the lender's pool; otherwise, from the VM pool of free pages. I/O-oriented IPC does not zero-complete, map, or copy to or from a server a page lent by that server.

In requests to a user-level server that subcontracts a kernel-level driver, the default specification of the user-level server as the lender avoids all zero-complete costs. In requests to a user-level server that subcontracts a user-level driver, zero-complete costs can be minimized by specifying as the lender the driver (input or output) or the server (output only).

If the lender is a user-level server, I/O-oriented IPC copies data or swaps pages directly between the address spaces of client and lender. Before page swapping, reverse copyout guarantees that server pages are filled with client data, and therefore no zero completion is necessary to preserve protection.

Indication of a lender is a soft condition: The default lender is often one that reduces data passing costs. With a properly tuned t_m, relative to copying, lender specification is also advisory only.

VI. Fragment subcontracting

In a request, neither the client nor the server is allowed to specify by virtual address the location of a buffer in the transient mapping region. Such location can be specified only by reference to the request that passed the buffer to the party. Let r be a request that passes a buffer b to a server s. In a subcontract request r', s specifies the location of a fragment b' of b by a triple containing the identifier of r, the index of b within r, and the offset of b' from the start of b. The identifier of r is an integer passed by the IPC facility to s when delivering the request. s uses this same identifier to notify request processing completion. To preserve protection, the interface checks that the caller (s) is indeed the server of r. Indication of location by such triple offers the following advantages:

1. It indicates that b' already has corresponding system buffers and that no further data passing may be required to pass data between b' and system buffers.
2. It allows servers s to subcontract to other servers or drivers s' services whose read and write flags are not set. s cannot specify the system buffers of such services by virtual address because the buffers are not mapped to s.
3. It provides the information necessary for updating the subcontract graph of r, linking r and r'. If r needs to be evicted (Section VII), this link indicates that r' should also be evicted, so as to eliminate the references of r' to the system buffers of r.

s may include in r', in addition to b', other buffer elements, possibly not located in the transient mapping region. If s is a TCP/IP server, for example, it may prepend to b' a buffer element h native to s, to hold the packet header. The data in h can be passed to and from system buffers by emulated copy.

VII. Buffer management

In I/O-oriented IPC, system buffers, and consequently server buffers, are always resident in physical memory. For every original request, system buffers would have to be in physical memory at least while being used by a driver; making system buffers permanently resident simplifies selective transient mapping and eliminates overheads of dynamically making buffers pageable or unpageable.

Resident system buffers also make TCOW safe. Some IPC facilities, notably the *out-of-line* Mach facility [8], pass pageable client buffers copy-on-write to servers. Such scheme is unsafe because client buffers may be backed by untrusted, client-supplied pagers, and therefore servers may take a page fault on a pageable buffer and never recover. The fbuf facility [6] eliminates the unsafety of pageable buffers by imposing a hard condition: Client buffers must be located in the fbuf region, which can be backed by a trusted, facility-supplied pager. It indeed appears that IPC facilities that use pageable buffers either have required copying, like in-line Mach IPC, or have been unsafe, like out-of-line Mach IPC, or have had hard optimization conditions and attendant incompatibility, like fbufs. I/O-oriented IPC uses resident system buffers so that it can be safe, can avoid hard optimization conditions, and can avoid copying while maintaining compatibility with applications that use copy semantics.

Resident system buffers do require, however, certain precautions to prevent buggy or malicious clients or servers from depleting the system's physical memory. I/O-oriented IPC:

1. Enforces per-process and global *limits* on the amount of physical memory occupied in pending I/O requests. If the facility determines that a given request would exceed one of these limits, the facility blocks the client and enqueues the request for later retrial, if the request is synchronous, or returns an indication of buffer exhaustion, if the request is asynchronous. Processes can request adjustment of their limit up to a hard limit imposed by the facility. This mechanism is similar to that of BSD Unix's *socket* interface [12].
2. Enforces a configurable maximum *timeout* interval for every original request. If a given request times out, the facility *evicts* it, as explained below.
3. Maintains per-process lists of pending I/O requests. When a process terminates, the facility evicts all the process's pending requests.

A. Request eviction

To evict a request, I/O-oriented IPC aborts every request in the respective subcontract graph in *postorder* fashion (that is, for each request r in the graph, the facility aborts the requests derived of r *before* aborting r).

The IPC facility aborts a request r by making an *abort* request r' to the server of r. Abort is a service that every server must provide. Abort requests are timed out. If request r' to abort request r times out, and the server of r executes at user level, the facility unilaterally unmaps the system buffers of r from that server's address space.

Eviction of the subcontract graph in postorder has the following advantages:

- It prevents race conditions between a request to abort a request r and completion of the requests previously directly or indirectly derived of r.
- It makes it unnecessary for the server of each aborted request r to make new subcontract requests to abort the requests previously derived of r or to reply to r.
- It ensures that all subcontract requests are aborted, including those whose contractor is an untrusted user-level server. This guarantees that the original request's buffers are actually freed.

VIII. Experimental results

This section reports end-to-end latencies measured for datagram communication between computers running a modified version of the NetBSD 1.1 operating system and connected by the CreditNet ATM network at 512 Mbps. In the experiments, user-level clients made input and output requests to a kernel- or user-level datagram protocol server, which subcontracted a kernel-level network driver. In the user-level server case, client and server communicated using I/O-oriented IPC with or without emulated copy; input and output services had the read flag set.

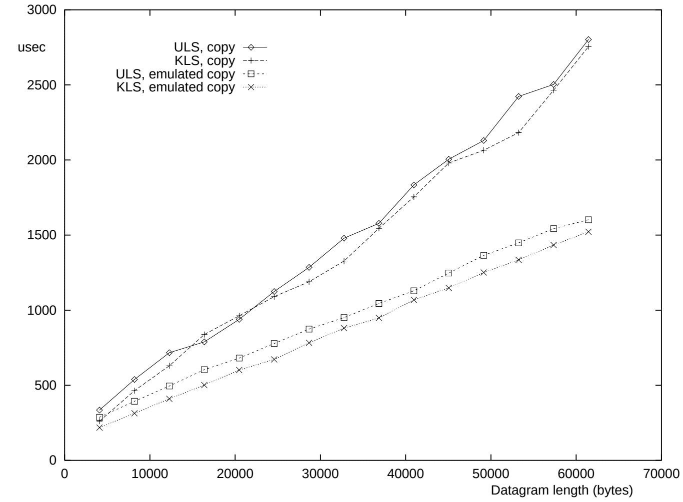

Fig. 4. With I/O-oriented IPC, the performance of user-level servers (ULS) approaches that of kernel-level ones (KLS). User-level performance with emulated copy is better than kernel-level performance with copying.

Figure 4 shows the end-to-end latency for communication between user-level clients making requests to kernel- or user-level protocol servers (KLS and ULS, respectively) and running on Pentium Pro 200 MHz PCs. Data passing between client and system buffers was by copying or emulated copy. The equivalent throughputs for single 60 KB datagrams were, using respectively kernel- or user-level servers, 178 or 175 Mbps with copying, and 323 or 307 Mbps with emulated copy. The curves indicate that for data longer than about a page, end-to-end performance was dominated by the network latency and presence or absence of data copying in the end systems. The placement of servers (KLS vs. ULS) had relatively low impact. For all data lengths, performance was *better for a user-level server* with I/O-oriented IPC and emulated copy than for a conventional kernel-level server with data passing by copying.

Figure 5 shows the corresponding latencies for short data with thresholds $t_o = 1666$ and $t_i = 2178$ bytes. Both for kernel- and user-level servers, latency was about the same with copying and emulated copy for datagrams of length up to about half a page, above which reverse copyout significantly reduced latencies with emulated copy. Kernel-level servers decreased end-to-end latency for 4-byte datagrams from about 200 μs to about 140 μs.

Figures 4 and 5 indicate that, with I/O-oriented IPC, the *ULS overhead* (i.e., the difference in latencies with user- or kernel-level servers) is relatively insensitive to data length

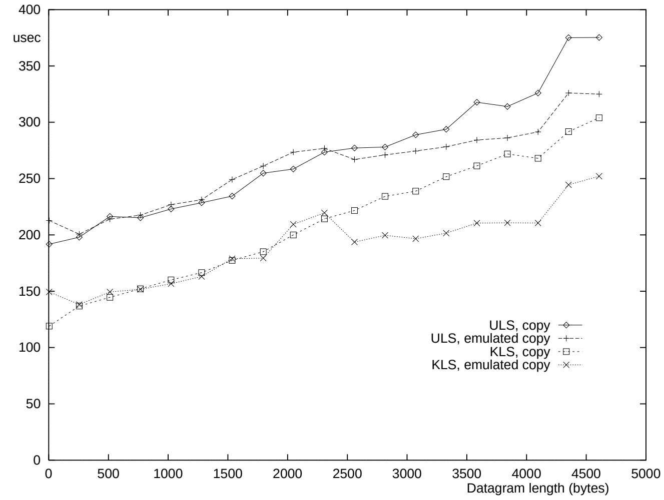

Fig. 5. I/O-oriented IPC imposes essentially only control passing overheads: rescheduling and context switching between user-level clients and servers.

and data passing scheme. That is, ULS overhead is more of a control passing overhead than a data passing overhead. This clearly would not the case in conventional IPC mechanisms, which pass data by copying. In I/O-oriented IPC, much of the ULS overhead is due to a full context switch between user-level client and user-level protocol server at each end host. Therefore, I/O-oriented IPC may exhibit considerably lower ULS overhead on a true microkernel system (e.g., L3 [11]), which typically has much lower context switch times than those of a monolithic system such as the one used in this paper's experiments (NetBSD).

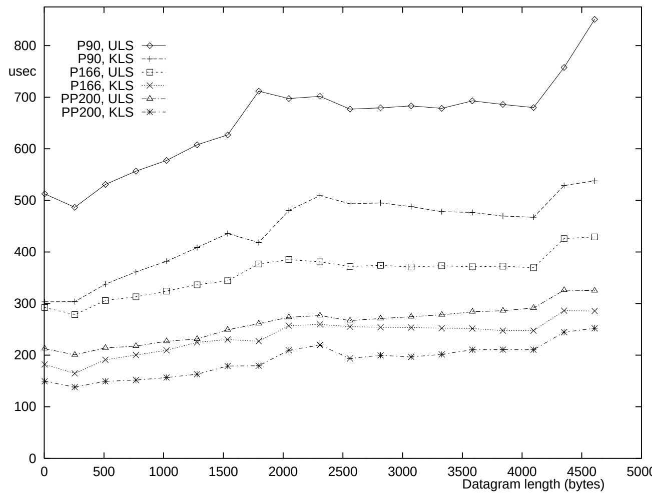

Fig. 6. Processor improvements are quickly reducing the overhead of user-level servers (ULS vs. KLS).

Figure 6 shows end-to-end short-datagram latencies using emulated copy on the same network but with different end hosts: pairs of Pentium 90 MHz (P90), Pentium 166 MHz (P166), or Pentium Pro 200 MHz (PP200) PCs. The curves demonstrate that faster processors reduce not only end-to-end latency with kernel-level servers (with a lower

TABLE I

SCALING OF ULS OVERHEAD OF EACH PLATFORM RELATIVE TO THAT OF THE PENTIUM PRO 200 MHZ PCS.

Processor	SPECint95	ULS overhead (μs)	Predicted scaling	Actual scaling
Pentium 90 MHz	< 2.88	209	> 2.85	3.32
Pentium 166 MHz	4.56	110	1.80	1.75
Pentium Pro 200 MHz	8.20	63	1.00	1.00

bound at the device latency), but also the ULS overhead. The 4-byte latency with kernel-level servers on the P90 was close to that with user-level servers on the P166, a processor introduced about one year later than the P90. Similarly, the 4-byte latency with kernel-level servers on the P166 was close to that with user-level servers on the PP200, a processor introduced about one year later.

In the measurements with user-level servers, the latter were the lenders of system buffers (by default), and neither zero completion nor copying between system and server buffers was necessary. In such conditions, the number of data-touching operations in I/O-oriented IPC is the same as that in a system call, and therefore it is reasonable to expect ULS overhead to scale inversely to processor performance, rather than memory performance. Table I validates this model with data from Figure 6 and the listed SPECint95 ratings of the respective processors.

This scaling model projects that processor performance improvements (more than 50% per year in the recent past [9]) tend to reduce ULS overhead and the performance advantage of kernel-level servers.

IX. Related work

Container shipping [16] allows something similar to a service with no mapping flag set, that is, a subcontracted service whose contractor passes to a subcontractor a buffer that the contractor itself cannot access. Like I/O-oriented IPC, the fbufs facility [6] maps most buffers read-only (in fbufs, except in the case of *producers* – driver on input, client on output). DASH [22] and fbufs also use a region that starts at the same virtual address in every address space, so as to reduce virtual memory calculations.

IPC facilities that pass data by copying usually copy the data twice, once between each party's and system buffers. L3 [11] reduces the number of copies to one using a temporarily mapped *communication window*, but does not have anything analogous to user-to-user input alignment, reverse copyout, and page swapping, which may reduce or eliminate copying altogether.

Local Peregrine RPC [10] comes close to the I/O-oriented IPC solution, passing data out of client buffers by copy-on-write and into and out of servers by mapping and unmapping. However, Peregrine passes data into client buffers by copying instead of input alignment and page swapping. I/O-oriented IPC's novel use of the latter technics allows it to be the first IPC facility to preserve an API with copy semantics *and* offer bi-directional copy avoidance.

Extensible kernel systems can, in principle, make it safe for users to install, modify, or customize kernel-level servers. The usual price of safety, however, is run-time overheads: For example, SPIN [1] requires extensions to be written in a type-safe language, and VINO [18] encapsulates extensions for software fault isolation. These safety techniques have been reported to make code run from 10% to 150% more slowly [19]. Additionally, kernel-level debuggers of extensible kernel systems have been as primitive and unsafe as those of other systems.

Several previous works have proposed direct application access to network adapters [7], [20], [2], bypassing the operating system and eliminating control and data passing overheads. The system retains the global state of the adapter, but allows applications to open a *channel* to the adapter. A channel consists of dedicated control/status registers in the adapter and a buffer for data passing between application and adapter. While the channel is open, the application can write directly into channel registers, without system call overheads, requests for receiving data into or sending data from the channel buffer. Several difficulties make it hard to adopt this scheme in production systems:

1. Data passing in channel buffers without copying does not preserve copy semantics and therefore causes incompatibilities with many existing applications.
2. The input and output functionality of any required servers and driver must be linked as a library with the user application. As discussed in the Introduction, library-based, decomposed servers [13], [21] can be considerably more complex than equivalent monolithic or microkernel servers and may not always be feasible. For example, for quality of service guarantees, network drivers must schedule packets globally, which may be unsafe in code linked with user applications.
3. Few adapters have channel registers, which can be dedicated to specific applications. In adapters that do have such registers, their number is necessarily limited. Also, channel buffers must be unpageable, given that they are directly used by the adapter; unpageable memory is also a limited resource. It is not clear how the system would prevent the hoarding of channel registers and unpageable memory by applications.

X. Conclusions

In previous papers [3], [4], we demonstrated that emulated copy preserves copy semantics but performs almost as well as or better than other data passing schemes for monolithic systems. In this paper, we presented a novel analysis of aspects that can be, in practice, as important as semantics for compatibility with legacy applications: namely, the restrictiveness and criticality of optimization conditions. We demonstrated that emulated copy's conditions may be satisfied by unmodified legacy applications and, when not satisfied, do not penalize the application's performance by much if thresholds are properly tuned.

Additionally, we expanded emulated copy with novel techniques for copy avoidance in microkernel systems: selective transient mapping, user-to-user input alignment, fragment subcontracting, and request eviction. In experiments on a high-speed network, the resulting I/O-oriented IPC facility gave user-level protocol servers performance approaching that of kernel-level ones. Performance differences were due mostly to control passing overheads. Tests on different processors showed that such differences are shrinking very fast and scaling roughly inversely to processor performance.

Our results suggest that common operating systems, such as Unix and Windows NT, can adopt I/O-oriented IPC and move protocol servers to user level, greatly and safely improving server debuggability, maintainability, and fault isolation without disruption of legacy applications, complex server restructuring, or significant performance degradation. In fact, for data longer than about a page, I/O-oriented IPC gives user-level servers better performance than that of kernel-level servers of conventional systems, which pass data by copying. We are not aware of any previous solution with the same combined advantages.

References

[1] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System", *Proc. 15th SOSP*, ACM, Dec. 1995, pp. 267-284.

[2] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten and J. Sandberg. "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer", in *Proc. 21st Annual Int. Symp. Comp. Arch.*, IEEE/ACM, Apr. 1994, pp. 142-153.

[3] J. Brustoloni and P. Steenkiste. "Effects of Buffering Semantics on I/O Performance", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 277-291.

[4] J. Brustoloni and P. Steenkiste. "Copy Emulation in Checksummed, Multiple-Packet Communication", in *Proc. INFOCOM'97*, IEEE, Kobe, Japan, April 1997, pp. 1124-1132.

[5] H. Custer. "Inside Windows NT". Microsoft Press, 1993.

[6] P. Druschel and L. Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", in *Proc. 14th SOSP*, ACM, Dec. 1993, pp. 189-202.

[7] P. Druschel, L. Peterson and B. Davie. "Experience with a High-Speed Network Adaptor: A Software Perspective", in *Proc. SIGCOMM'94*, ACM, Aug. 1994, pp. 2-13.

[8] D. Golub, R. Dean, A. Forin, R. Rashid. "Unix as an Application Program", in *Proc. 1990 Summer Conf.*, USENIX, June 1990.

[9] J. Hennessy and D. Patterson. "Computer Architecture: A Quantitative Approach", 2nd. ed., Morgan Kaufmann Pub., San Francisco, CA, 1996.

[10] D. B. Johnson and W. Zwaenepoel. "The Peregrine High-Performance RPC System", in *Software — Practice and Experience*, 23(2):201-221, Feb. 1993.

[11] J. Liedtke. "Improving IPC by Kernel Design", in *Proc. 14th SOSP*, ACM, Dec. 1993, pp. 175-188.

[12] S. Leffler, M. McKusick, M. Karels and J. Quaterman. "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley Pub. Co., Reading, MA, 1989.

[13] C. Maeda and B. Bershad. "Protocol Service Decomposition for High-Performance Networking", in *Proc. 14th SOSP*, ACM, Dec. 1993, pp. 244-255.

[14] C. Maeda. "Service Decomposition: A Structuring Principle for Flexible, High-Performance Operating Systems", Ph. D. Thesis, CMU-CS-97-128, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Apr. 1997.

[15] J. Ousterhout. "Why aren't operating systems getting faster as fast as hardware?", in *Proc. 1990 Summer Conf.*, USENIX, June 1990, pp. 247-256.

[16] J. Pasquale, E. Anderson and P. Muller, "Container Shipping: Operating System Support for I/O-Intensive Applications", *Computer*, 27(3):84-93, IEEE, Mar. 1994.

[17] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black,W. Bolosky and J. Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", in *Proc. 2nd ASPLOS*, ACM, Oct. 1987, pp. 31-39.

[18] M. Seltzer, Y. Endo, C. Small, K. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions", *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 213-227.

[19] C. Small and M. Seltzer, "A Comparison of OS Extension Technologies", *Proc. 1996 Annual Tech. Conf.*, USENIX, Jan. 1996, pp. 41-54.

[20] T. von Eicken, A. Basu, V. Buch and W. Vogels. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", in *Proc. 15th SOSP*, ACM, Dec. 1995, pp. 40-53.

[21] C. Thekkath, T. Nguyen, E. Moy and E. Lazowska. "Implementing Network Protocols at User Level", in *Proc. SIGCOMM'93*, ACM, Sept. 1993, pp. 64-73.

[22] S.-Y. Tzou and D. Anderson. "The Performance of Message-passing using Restricted Virtual Memory Remapping", in *Software – Practice and Experience*, vol. 21(3), March 1991, pp. 251-267.