Using Annotated Interface Definitions to Optimize RPC

Bryan Ford Mike Hibler Jay Lepreau

Department of Computer Science, University of Utah

{baford,mike,lepreau}@cs.utah.edu

Technical Report UUCS-95-014
March, 1995

Abstract

In RPC-based communication, it is useful to distinguish the RPC *interface*, which is the "network contract" between the client and the server, from the *presentation*, which is the "programmer's contract" between the RPC stubs and the code that calls or is called by them. Presentation is usually a fixed function of the RPC interface, but some RPC systems, such as DCE and Concert, support the notion of a *flexible presentation* or *endpoint modifier*, allowing controlled modification of the behavior of the stubs on each side without affecting the contract between the client and the server.

Up until now, the primary motivation for flexible presentation has been for programmer convenience and improved interoperability. However, we have found flexible presentation also to be useful for optimization of RPC, and in many cases *necessary* to achieving maximal performance without throwing out the RPC system and resorting to hand-coded stubs. In this paper we provide examples demonstrating this point for a number of different operating systems and IPC transport mechanisms, with RPC performance improvements ranging from 5% to an order of magnitude. In general, we observe that the more efficient the underlying IPC transport mechanism is, the more important it is for the RPC system to support flexible presentation, in order to avoid unnecessary user-space overhead.[1]

1 Introduction

In a typical remote procedure call (RPC) system, interfaces between clients and servers are defined explicitly in an interface definition language (IDL), and the IDL file is processed by a stub generator to produce client and server stubs[4]. The primary purpose of the IDL file is to define the "network contract" between the client and the server: what operations can be invoked and what information must be passed across the network on an invocation. However, in most RPC systems, the IDL file also indirectly defines the "programmer's contract" between the stubs and the programmer: how parameters are passed to the stub, who allocates storage for the parameters, etc. For example, consider the following CORBA[19] IDL fragment:

```
interface SysLog {
      void write_msg(in string msg);
};
```

Given this interface definition, a CORBA-compliant stub compiler for C will *always* produce a stub with the following C function prototype, with the `msg` parameter assumed to be null-terminated:

```
void SysLog_write_msg(SysLog object, CORBA_Exception *ex, char *msg);
```

However, the stub could just as easily conform to the following function prototype instead, taking the length of the string explicitly through the `length` parameter:

```
void SysLog_write_msg(SysLog object, CORBA_Exception *ex, char *msg, int length);
```

[1] This research was supported in part by the Advanced Research Projects Agency under grant number DABT63–94–C–0058 and by the Hewlett-Packard Research Grants Program.

This difference should not affect the protocol between the client and the server: a client stub using the former prototype should still be able to invoke a server stub using the latter, because the C calling convention is merely a local language issue. In our terminology (adopted from the OSI networking model[13]), these function prototypes represent alternate *presentations* of the same interface. The former is the *standard* presentation, but by no means the only possible one.[2]

While the example was drawn from CORBA, this restriction occurs in most existing RPC systems, because they support only a single fixed presentation for any given interface definition. However, a few allow the presentation to be varied independently for a given client or server. In DCE[21], a few presentation attributes can be specified explicitly, separately from the IDL file defining the interface, in a supplemental file known as an *application control file* (ACF). Thus, while all clients and servers using a particular RPC interface generally share the same IDL file, each can have its own ACF and thus specify its own presentation annotations for its stubs.

In existing systems that support a notion of flexible presentation, this feature is primarily used to make RPC more convenient to the programmer and to increase interoperability. However, we have found that flexible presentation is also extremely important in optimizing RPC performance. In this paper we demonstrate this principle with seven examples. To demonstrate the broad applicability of this concept, we implemented the examples in a variety of environments and transport protocols.

Our results show that flexible presentation benefits RPC performance for two reasons:

1. Flexible presentation support is necessary to create optimal stubs: any fixed presentation is the wrong one some of the time, causing unnecessary data copying in either the user code, the stubs, or both. For example, if a client wants to read data through RPC into a particular buffer, but the RPC stub insists on allocating a new buffer for the returned data, the client will have to perform an extra copy—often uselessly, because the stub could just as easily have unmarshaled the data into the client's buffer in the first place.

2. If the transport mechanism can itself be specialized, it can leverage relaxed semantic constraints at the endpoints. Flexible presentation support provides a principled way to specify these constraints. For example, if data sent in an RPC is uninterpreted by the server (e.g., I/O buffers), the client can be allowed to retain access to the data while the server processes it, saving overhead, as in volatile fbufs[6].

In the rest of this paper we describe related work (Section 2), briefly describe the design of our IDL compiler (Section 3), present specific examples in which flexible presentation enhances RPC performance and quantify the improvements (Section 4), propose future work, and conclude.

2 Related Work

The term "presentation" comes from the OSI networking model[13] which defines a *presentation layer* responsible for providing applications with data in the locally-preferred representation. The primary presentation emphasis in the OSI model is on data representation at the "format" level (e.g., ASCII or EBCDIC, big- or little-endian).

We know of two IDLs for RPC in which the notion of separate presentation exists to some degree. DCE[21], with its "Application Configuration File" (ACF), was, to our knowledge, the first RPC system to support separately specified presentation. The ACF is distinct from the IDL file, and allows specification of a few local attributes such as method of binding, error condition treatment, and data type equivalence.

[2] In CORBA terminology, the standard presentation is known as the *language mapping*.

However, presentation is not fully separated out—the DCE IDL still contains many presentation attributes that belong in the ACF. Also, the ACF's primary purpose is for programmer convenience, not optimization.

The Concert/C distributed programming system[2, 1] much more fully develops the concept of flexible presentation. In their system, as in ours, presentation includes storage allocation attributes, such as which entity (application, stub, transport) is responsible for the allocation and deallocation of storage for each parameter. Concert terms the programmer's contract the "endpoint modifier" and the network contract the "contract," striving for a "minimal contract" in order to achieve maximal interoperability between target languages. Since Concert does not have an IDL separate from the target application language, both the endpoint modifier and the contract are automatically generated by the Concert/C pre-processor. In Concert, the primary purpose of this separation is to handle the vagaries of RPC mapping to different languages; for example, whether a returned value maps to an `out` parameter or to a function return value.

Subcontract[11], a general method for extending and specializing object communication mechanisms and semantics[3], is largely complementary to our work, addressing a level higher than the stub level. The Spring system has separate stub, subcontract, and kernel layers, with the language-level stubs and kernel IPC mechanism remaining entirely generic, and all specialization restricted to the subcontract layer. Therefore, in the general case, the exact details of interaction with the application code ("presentation"), performed by the stubs, cannot be specified, varied, controlled, or optimized. However, when an *object* is a parameter to a call, that object's subcontract can specialize the object's marshaling and a few other aspects of presentation. Thus, subcontract supports specialization based on target object type, whereas flexible presentation supports specialization based on local requirements.

In contrast to Subcontract, which addresses a higher level (object communication) than do we, USC—the Universal Stub Compiler[20]—is targeted at a lower level. It is aimed at optimizing the generation of generally fixed-format stubs which marshal network protocol headers, efficiently performing byte swapping, alignment mapping, etc. USC is designed to be used by a higher level stub generator.

In the traditional networking domain, Clark and Tennenhouse[5] emphasize the importance of optimizing the presentation layer, showing that it often dominates processing time. They also emphasize that for performance reasons, the necessity of non-contiguous data location in the recipient is a critical architectural constraint. This can occur not only in RPC, where each parameter is scattered in memory, but in integrated-layer processing of stream-based protocols, due to dropped packets. Their emphasis is on the latter, ours the former, but similar issues and performance effects obtain.

We bring to RPC the focus on presentation optimization that has been elucidated in the traditional networking arena[5, 20]. Our contributions are two fold: (i) demonstrating that, in many situations, flexible presentation is necessary to achieve maximum performance, and (ii) showing how to support flexible presentation in a practical RPC system.

3 Design

We have designed and implemented a new RPC stub compiler that supports flexible presentation while retaining compatibility with existing RPC systems. The stub compiler is cleanly separated into front-ends and back-ends so that it can both read multiple existing IDLs as its input and generate stubs for various operating systems and transport protocols as its output. Currently we have Sun and CORBA IDL front-ends working and a MIG front-end under construction; we have back-ends for Sun RPC/XDR in a Unix environment and several Mach-based protocols (transport mechanisms). Although the stub compiler currently only generates stubs for C, support for other languages could be added easily.

Flexible presentation is supported in our system by adding a third compiler stage between the front-end

[3] CORBA "object adaptors" provide a subset of the functionality of Subcontract.

and back-end, in which the presentation of an RPC interface is modified declaratively through the use of a *presentation definition language* (PDL). Nothing declared in the PDL file can affect the contract between client and server; thus, while all clients and servers using a particular RPC interface will generally share the same IDL file defining that interface, each can have its own PDL file.

The IDL compiler front-end always provides a *default presentation* computed from the interface definition by fixed, standardized rules. A PDL file only has to be written if the user wants a presentation that deviates from that default, and even then the PDL file only needs to contain declarations for stubs and types for which non-default presentations are needed; it is not necessary to "re-declare" everything in the interface.

The syntax of our PDL is similar to that of DCE's Attribute Control File (ACF) format, and also to Concert/C's annotated C language syntax, both of which serve a similar function. For example, given the example CORBA `SysLog` interface presented in the introduction, the following PDL file will cause the second presentation shown (the "alternate" presentation) to be used instead of the default CORBA presentation. The keyword ``` ``length_is'' ``` is reserved only when it occurs in a presentation specification (inside brackets).

```
SysLog_write_msg(,, char *[length_is(length)] msg, int length);
```

4 Experimental Results

In this section we present the results of a number of experiments demonstrating the usefulness of flexible presentation in optimizing RPC. Sections 4.1–4.4 illustrate the first main point made in the introducation, showing how flexible presentation can be used create more optimal stubs. These sections are arranged in order, starting with RPC on heavyweight transport mechanisms on monolithic systems, and progressing to extremely lightweight transports in highly decomposed object-based environments. Section 4.5 demonstrates the second point, that presentation attributes can be leveraged by specializable transport mechanisms, to further optimize RPC.

Except for the Linux example, all tests were performed on an HP730 (66 MHz PA-RISC architecture) running Mach 3.0 and the Lites[12] Unix single server. The times presented below include all RPC costs, including both user-level stub code and kernel processing.

4.1 Linux NFS Client

In several traditional Unix-compatible monolithic operating systems, such as 4.3/4.4BSD[16] and Linux[14][4], the NFS file system support built into the kernel uses manually-written Sun RPC stubs, even though an automatic stub generator (`rpcgen`) is available. In the case of Linux one reason this is done is so that the stubs can marshal read or written data directly to or from the appropriate user address space using the kernel's special "copy-in" and "copy-out" routines. This optimization would not be possible with `rpcgen`-generated stubs, because in its standard presentation, arguments are always marshaled to/from the local address space (i.e., kernel space).

Flexible presentation provides an alternative to hand-coding these stubs. Our stub generator provides a presentation attribute, `[special]`, that can be attached to any data type, and indicates that arguments or variables of that type will be marshaled/unmarshaled by special routines provided by the programmer. In our experiment, we simply provided routines to use the kernel's copy-in/copy-out routines (`memcpy_fromfs()` and `memcpy_tofs()`) instead of the normal `memcpy` routine. The automatically generated stubs call these routines at the appropriate times, but still marshal all other arguments and all handle other aspects of the RPC as before.

Figure 2 shows the performance of reading an 8MB across an ordinary Ethernet from an HP 700 file server running BSD to a 486DX2/66 PC running Linux. The left part of each bar represents network and

[4] Linux is a Unix-compatible operating system with a traditional monolithic design.

```
[comm_status] int nfsproc_read(, nfs_fh *file,
    unsigned offset, unsigned count, unsigned totalcount,
    [special] user_data *data, fattr *attributes, nfsstat *status);
```

Figure 1: PDL Declaration for Linux NFS client stub with special data presentation

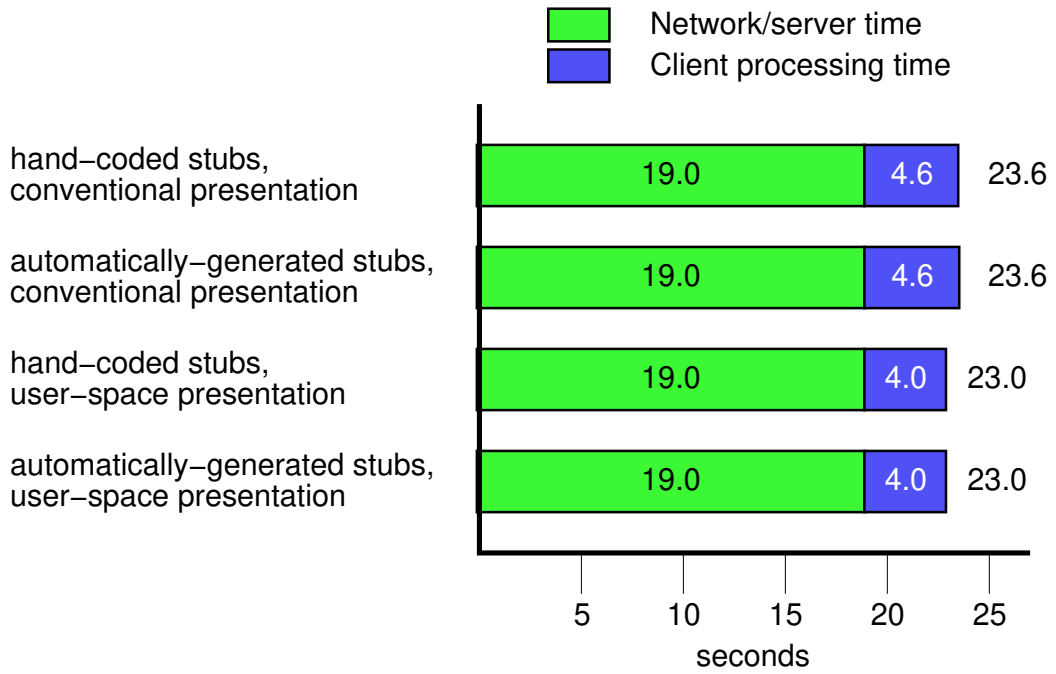

Figure 2: Performance Effect of User-space Buffer Presentation

server processing time; Since we did not change the server or network, this part is the same in each case. The right part represents the client processing time, including all marshaling/unmarshaling and protocol stack processing. The topmost two bars show the performance using stubs that conform to a conventional presentation in which the stubs unmarshal the read data into an intermediate kernel buffer before it is copied to user space. The bottom two bars show the performance of stubs using a special user-space buffer presentation: the third bar for the original hand-coded client stubs, and the fourth for the automatically-generated stubs produced by our flexible-presentation stub compiler. There is essentially no performance difference between hand-coded stubs and automatically-generated stubs supporting the same presentation; however, the user-space presentation optimization, whether implemented manually or automatically, provides about a 13% performance improvement in client-side processing time (about 3% overall). With a faster network and better-optimized protocol stacks in the client and server, we would expect the effect of this optimization to be correspondingly greater.

The PDL declaration used to specify the special handling of the received file data is shown in Figure 1; the parameter of interest is the '`data`' parameter. (The other parameters in the declaration in fact also modify the presentation from the default, but for convenience reasons, not performance: they cause the generated stub to have a function prototype more like that of the original hand-coded stub being replaced.)

4.2 Pipe Server

To demonstrate how presentation can be used to optimize RPC in a decomposed, microkernel-based system, we will examine a simple *pipe server* that provides Unix pipe semantics (buffering, flow control, etc.) on top of ordinary synchronous RPC. This server is representative of a common model of communication: an intermediate entity that performs a data transformation between two parties. Thus, the results presented here should generalize to other components in a decomposed system.

Our implementation runs on the Mach microkernel, in a Lites[12] environment (a Unix server based on BSD 4.4 Lite). The standard Lites server is mostly monolithic in nature, with the same overall design as the original CMU UX single server[9]. However, in our version of Lites we have implemented the pipe server as a separate Mach task outside of the main Unix server. Processes that have open file handles to a particular pipe communicate directly with the pipe server to read and write data.

Communication with the pipe server is done using a new, streamlined low-level Mach IPC mechanism similar to those implemented in other recent microkernels[10, 17, 8]. The new IPC path allows messages to be transferred through processor registers and/or a simple message buffer copied by the kernel directly from the source address space to the target address space. Unlike traditional Mach IPC, the new IPC path does not directly support copy-on-write data transfer; this can still be done either by falling back on the old IPC primitives, or by using the kernel's VM copying primitives directly from user space. However, for the purposes of this paper, we only need the simple optimized IPC path.

4.2.1 Example: Allocation Semantics for Reads from a Pipe

The pipe server provides a typical Unix file interface; clients make `read` and `write` RPCs to the server to read and write data to or from the pipe. The pipe to which reads and writes are directed is not a parameter to these calls, but is implied by the RPC connection or "pipe object reference" through which the RPC is invoked. Part of the relevant interface definition is shown in Figure 3, in CORBA IDL.

Figure 4 shows the standard CORBA server presentation for this interface: basically, the `sequence<octet>` type (a variable-length array of bytes) becomes a C `struct` type; the server work function `FileIO_read` must return an instance of this structure, containing a pointer to the data to return and its length. (The `_maximum` element is irrelevant to our purposes here.) In the default CORBA presentation, the data buffer returned has *move* semantics: after the server stub finishes marshaling the data into the reply message, it deallocates the buffer provided by the server.

Unfortunately, this default presentation requires the pipe server to perform an extra copy of all data it returns. Incoming data written to the pipe gets stored into a permanently-allocated, fixed-length circular buffer in the pipe server's address space. The server work function `FileIO_read` can't simply return a pointer into the pipe buffer because the server stubs will try to deallocate it; this would not work too well because the pointer returned is probably not to the beginning of the buffer, and in any case the buffer is likely to have more data than is requested by the `read` operation, and that data must be retained for future reads.

Our flexible presentation RPC system provides a fairly simple and obvious fix for this problem: you can specify that the server stubs should *not* deallocate the buffer on return, instead leaving the pipe server to manage its own buffer space.[5] Figure 5 shows a PDL file that modifies the server-side presentation appropriately; basically, it is just a re-declaration of the relevant C types, with the `[dealloc(never)]` presentation attribute added to the data buffer pointer declaration. This modification eliminates the need for the server to make an extra copy of the data before returning it, except when the data to be read wraps around from the end of the circular buffer to the beginning. (This case as well could be optimized with flexible presentation, but we did not implement this.)

Figure 6 shows the throughput of the Unix-compatible pipes provided by our pipe server, for a 4K pipe

[5] There are synchronization issues with this solution which we don't describe here; however, they can be solved fairly easily.

```
interface FileIO {
    sequence<octet> read(in unsigned long count);
    void write(in sequence<octet> data);
};
```

Figure 3: Pipe Server Interface

```
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    char *_buffer;
} CORBA_SEQUENCE_char;

CORBA_SEQUENCE_char FileIO_read(FileIO object, CORBA_Exception *ex,
                                unsigned long count);
```

Figure 4: Pipe Server Default Presentation

```
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    char *[dealloc(never)] _buffer;
} nodealloc_buffer;

nodealloc_buffer FileIO_read(FileIO object, CORBA_Exception *ex,
                             unsigned long count);
```

Figure 5: PDL Example: Modifying Presentation of the `read` Call

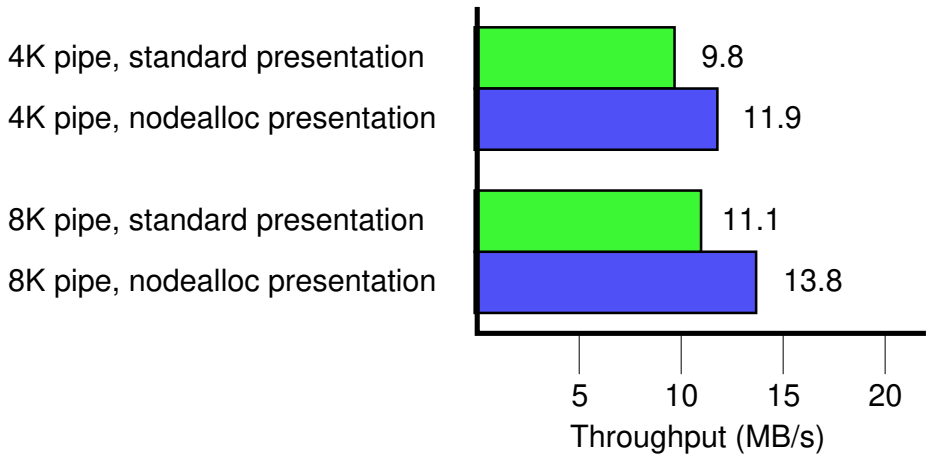

Figure 6: Performance of the Basic Pipe Server

buffer (top) and an 8K buffer (bottom). In each group, the top bar shows the total run time of the program with the pipe server using an unmodified presentation; the bottom bar shows the run time with the modified presentation. The overall difference is 21% and 24%, for 4K and 8K pipes, respectively.

Note that we have only demonstrated one of the possible presentation-based optimizations to this RPC path; others are certainly possible. For example, an additional copy could be eliminated from the `write` path with a similar presentation modification and a slight enhancement to the underlying IPC mechanism.

4.3 Integrating fbufs and RPC

Flexible presentation is useful for more than avoiding unnecessary copying in RPC over conventional IPC mechanisms; it can also be used to be used to integrate highly specialized or constrained transfer mechanisms cleanly into the RPC model, and take advantage of their unique performance features. In this section we examine the *fbufs* data transfer mechanism[6], a shared memory-based protocol that allows large amounts of data to be passed through many protection domains along semi-fixed "data paths" without incurring any expensive copying or virtual memory remapping operations along the way. Complex messages can be composed and split apart along the path by logically splicing together pieces of individual buffers.

The main potential drawback of fbufs is that imposes serious constraints on the way data is produced and consumed along the path: senders must be able to generate data in special buffers managed by the fbuf system (i.e. they can't simply provide a pointer to `malloc`'d memory), and receivers must follow special rules when accessing these buffers. These constraints are not directly compatible with conventional Unix and C programming practices; programs must inevitably be modified in order to take full advantage of fbufs. Further, because conventional RPC systems rely on presentations that match conventional C programming practices, conventional RPC is not well matched to fbufs.

To demonstrate how flexible presentation can be applied to this problem, we first consider an RPC system that uses fbufs as its underlying transport mechanism *completely transparently*, so that users of the RPC system are oblivious to the fact that fbufs are in use as opposed to, say, Unix sockets. In such a system, user code generally does not directly accesses fbufs, because the semantics it expects is incompatible with fbufs; instead, the RPC system marshals RPC parameters into and out of fbufs, passing the resulting fbufs as fully-marshaled messages. Only pairwise shared memory channels are taken advantage of, and the resulting system has the same basic performance characteristics as LRPC[3].

Given this basic mechanism, flexible presentation attributes can be used selectively to allow specific

clients and servers to take advantage the underlying fbufs mechanism, without bypassing the RPC system completely, and without making the modified components incompatible with other, unmodified clients and servers that use only a standard presentation. For example, in a typical decomposed Unix-compatible system, a large number of clients and servers are likely to use one basic file I/O interface such as the one described in the previous section. Any client may potentially talk to any server in arbitrary combinations, but only a small subset of those components and combinations are likely to require extremely high-bandwidth data transfer (e.g., disk and network drivers and file systems, but probably not console I/O, tape, or timers). It is probably feasible to modify a few performance-critical components to use a special fbuf-compatible presentation, but it is very undesirable to be required to modify *all* the components in the system just to get everything working together properly.

To demonstrate this principle, we created a simplified version of fbufs based on Druschel's original implementation; the primary difference is that it uses our new streamlined Mach IPC path for control transfer, and implements all of the fbuf creation and manipulation facilities in user space instead of in the kernel. The details of the implementation aren't relevant here; the basic performance characteristics are essentially the same as those previously reported. On top of that we again use our stub compiler to provide an RPC abstraction, and the pipe server described in the previous section.

Figure 7 shows the performance of the same Unix applications as in Figure 6, this time using fbufs as the transport. The top bar in each pair again represents a system in which all components (reader, writer, and pipe server) use a standard presentation, and thus fbufs is being used merely as a pairwise LRPC-like shared memory transport mechanism. The bottom bar shows the performance after modifying the pipe server to use a `[special]` presentation for incoming read and write calls, as was done in the Linux NFS client examples. This change allows the pipe server to keep all data in fbufs along the entire path through the server instead of copying the data into and out of a circular buffer in the middle. (The reader and writer clients still use standard presentations, so one extra copy is still done in each of them to get the data into and out of the fbufs at the endpoints; these clients could similarly be adapted to use fbufs directly and achieve additional performance improvement, but they don't have to be changed just to get things to work.)

This presentation modification improves the pipe throughput by 92% for 4K pipes and by 160% for 8K pipes. (We have not yet determined why 8K pipes are slower than 4K with the standard presentation). For reference, the graph also shows the throughput of normal Unix pipes in a monolithic 4.3 BSD system on the same hardware. In that implementation pipe buffers are always 4K in size.

4.4 Same-Domain Communication

One well-known way to reduce the overall performance overhead in decomposed systems is to "decouple modularity from protection"[7]: make all "inter-module" communication go through well-defined RPC-like interfaces that can cross process and machine boundaries, but allow the resulting system to be configured so that several modules share the same protection domain and RPC between them is optimized to simple procedure calls. Thus, in theory, the system remains highly decomposed and modular in structure, but can retain good performance in practical use.

However, short-circuiting same-domain RPCs into procedure calls does not automatically achieve the *overall efficiency* of normal intra-module procedure calls. In the following two sub-sections we will show through two examples how RPC systems typically still impose constraints on invocation semantics which the client and server code must work around. These constraints incur extra conversion and copying overhead in the client or server code. Similarly, the semantics defined by the RPC system may require the RPC stubs themselves to perform data conversion and copying even when the client and server are in the same domain.

Flexible presentation support in the RPC system can provide a convenient, powerful, and principled solution to this problem. Although all aspects invocation semantics can't be considered "presentation attributes," because they often involve a contract between the caller and callee, they can often be *derived from*

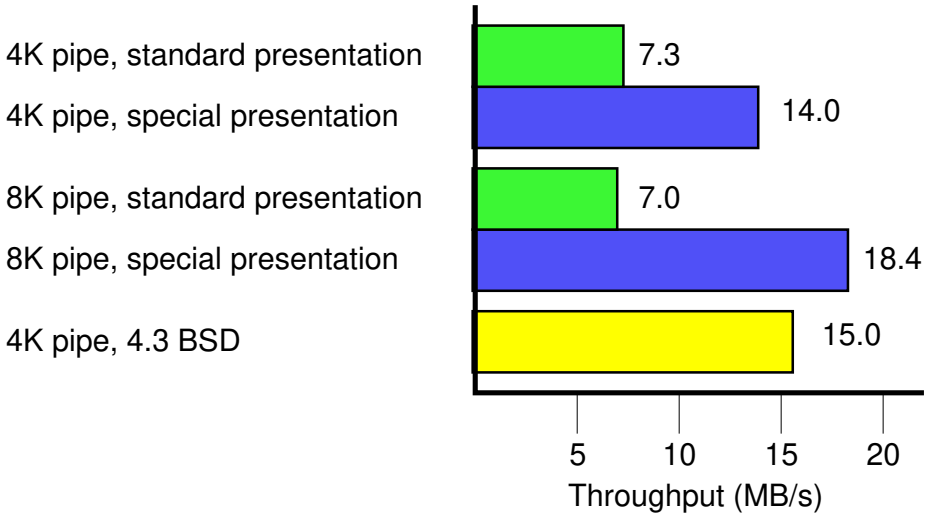

Figure 7: Performance of the Basic Pipe Server

presentation attributes. The RPC system can make use of the presentation attributes specified independently on each side in order to determine the optimal method of transferring the data while retaining semantic correctness.

In our current implementation of same-domain invocation, the RPC stubs perform all computation of invocation semantics at run time, once for each RPC invocation. More optimal implementations are certainly possible; however, even with the current "dumb" implementation, we found the additional overhead of this computation to be negligible.

We will now walk through two specific examples exhibiting the same-domain invocation semantics problem, and show how flexible presentation can be used to optimize it; these examples can easily be generalized to other aspects of invocation semantics.

4.4.1 Example: copy vs. borrow semantics for `in` parameters

When a server receives a data buffer or other complex argument as an `in` parameter during a normal RPC, it can usually assume that it is allowed to trash the contents of that buffer during processing, because the buffer generally has been copied into the server's private address space from another domain, and therefore modifications to that buffer won't affect the client's original data. In other words, RPC servers can generally assume pass-by-value (copy) semantics because of the "remoteness" of RPC. However, under these rules, if the client and server happen to be located in the same domain, the RPC stubs must *still* make a new copy of the buffer to pass to the server, even though in the common case the server probably won't modify the buffer.

The rules of the RPC system might instead forbid servers from ever modifying the contents of `in` parameters (i.e. "borrow" semantics); that would eliminate the additional copy in the RPC stubs for same-domain communication. However, this solution is akin to swatting flies with a sledgehammer: it *always* forces the server to make its own copy of incoming parameters if it wants to modify them, even though in many cases this isn't necessary, either because the invocation came from another process, or because the client doesn't care if its buffer is trashed.

Although the overall issue of copy versus borrow semantics cannot be considered a presentation attribute, it can be derived from two presentation attributes, one on each side. The client specifies, as part of its local presentation of the `in` parameter, whether or not it is OK for its buffer to be trashed during the

```
void FileIO_write(char *[trashable] _buffer, unsigned long _length);
```

Figure 8: Example client PDL file: indicates that the data buffer may be trashed

```
void FileIO_write(char *[preserved] _buffer, unsigned long _length);
```

Figure 9: Example server PDL file: server promises not to modify the buffer

call: we call this the `trashable` attribute. Similarly, the server specifies whether or not it wants to trash the contents of the buffer it receives, using the `preserved` attribute. During remote invocation, these attributes can generally be ignored by the RPC system; for same-domain invocation, the RPC stubs only need to make a separate copy of the parameter if *neither* the `trashable` *nor* the `preserved` attribute was specified (by the client and server, respectively). Figures 8 and 9 show example PDL declarations indicating these attributes for the client and server, respectively.

Performance: Figure 10 shows the performance of a same-domain RPC that takes a single 1KB `in` parameter, but otherwise does nothing. The leftmost bar in each group is for an RPC system that always provides copy semantics: the stubs must always make a new copy of the parameter for the server, so performance is always the same (bad). The middle bars are for an RPC system that only provides borrow semantics (i.e. requires that the server never modify incoming parameters): in this case, the stubs are always optimal, but if the server wants to modify its parameters in-place (rightmost two bar groups), it must always make its own copy even if the client doesn't care if its buffer is trashed or if the invocation is from another protection domain. The rightmost bar in each group is for flexible presentation, which copies only when necessary, and never requires the programmer to make a copy manually.

4.4.2 Example: allocation/deallocation of `out` parameters

Many systems, including COM[18] and CORBA, specify "move" semantics for `out` parameters, meaning the server allocates storage for the parameter and "donates" that storage to the client upon returning; the client must then free the buffer later. This allows arbitrary-size parameters to be returned without requiring the client to know in advance how big the `out` parameter will be. However, in normal intra-domain C or C++ programming practice, the client often *does* know how big the result will be, e.g., because it requested a certain number of bytes/elements. In this case, it is usually more convenient for the caller to provide a buffer which the callee simply fills in during the procedure invocation. If a communicating client and server would prefer the latter semantics due to their internal implementations, but are forced by the RPC system to use the former semantics, then extra overhead will be introduced on both sides even if the client and server are in the same domain: a buffer must be unnecessarily allocated by the server and deallocated by the client, and the client must copy its contents to wherever it wanted them in the first place.

Again, flexible presentation can be used to resolve this problem, achieving maximum performance when the client and the server agree on what they want, while retaining automatic interoperability between *all* clients and servers. The client and the server each specify, independently, whether they expect to allocate the buffer for the `out` parameter, or they expect the "other side" to do so. The RPC stubs automatically adapt to the requirements of each side, performing a copy only if *both* sides insist on allocating their own buffer for the parameter.

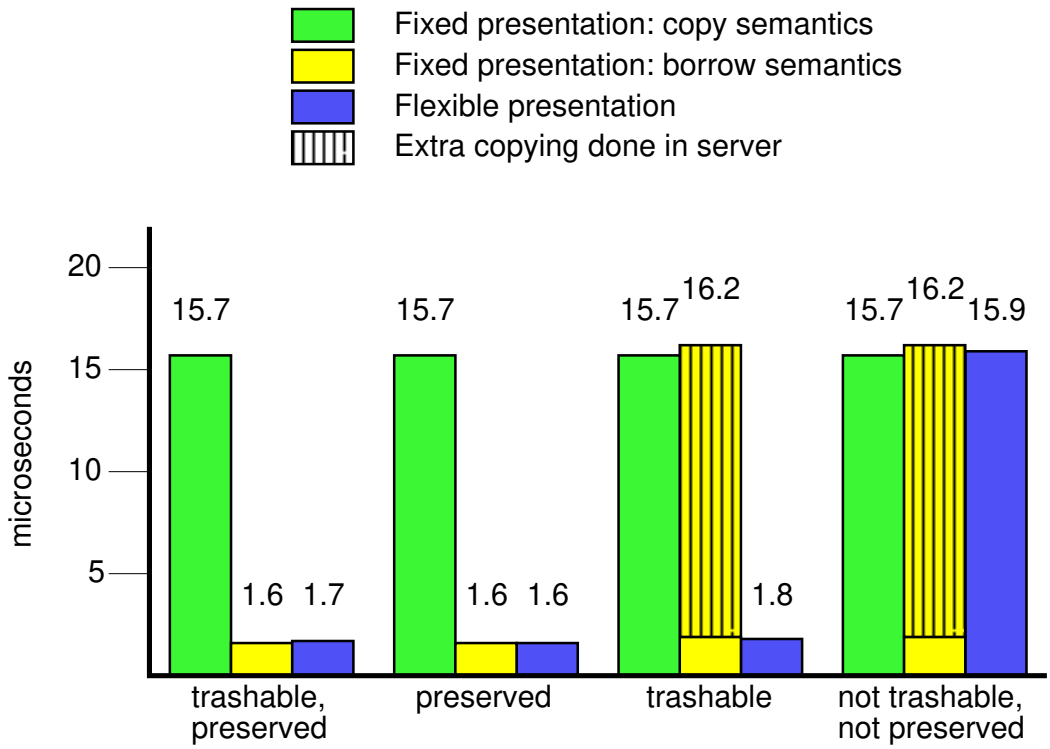

Figure 10: Performance of Varying Mutability Semantics

Performance: Figure 11 shows the performance of a same-domain RPC with a single 1KB `out` parameter, in various situations. Each bar group represents a particular combination of semantic *requirements* as seen by the endpoints: i.e. which semantics are most appropriate or convenient for the user code on each side. For example, the first group represents the situation where both sides prefer to let the "other side" (or the RPC system) allocate the buffer for the `out` parameter (i.e. neither side has any constraints); whereas in the second group, the server wants to provide the data buffer for the parameter (e.g. it is sending back data that had already been allocated before the call). The leftmost bar in each group is for a fixed-presentation RPC system that only supports "server allocates, client consumes" semantics (e.g. COM or CORBA). The middle bar is for a fixed-presentation system that only supports "client allocates, client consumes" semantics (e.g. MIG, for non-copy-on-write parameters). The rightmost bar in each group shows the performance with flexible presentation, where the RPC system adapts to the requirements of each side. In each case, the base (solid) bar shows the time spent in the RPC stubs, whereas the lined bars show time spent in the extra manually-written client or server "glue" code the programmer must write to adapt the requirements of the endpoints to the constraints imposed by the RPC system.

As one can see, flexible presentation always minimizes the amount of copying and memory management done, in addition to making life more convenient to the programmer by eliminating the need to write additional glue. The two middle groups represent the common case in which the client and server agree on allocation semantics. With flexible presentation, the minimum amount of work is done in either case; with fixed presentation, the performance is optimal if the RPC system's requirements match those of the endpoints, and terrible if they don't. The first and last bar groups represent "mismatches," in which the client and server disagree on who should allocate the buffer. Flexible presentation makes essentially no difference in these cases: *someone* must do the necessary matching and (in the case of the last group) copying; it

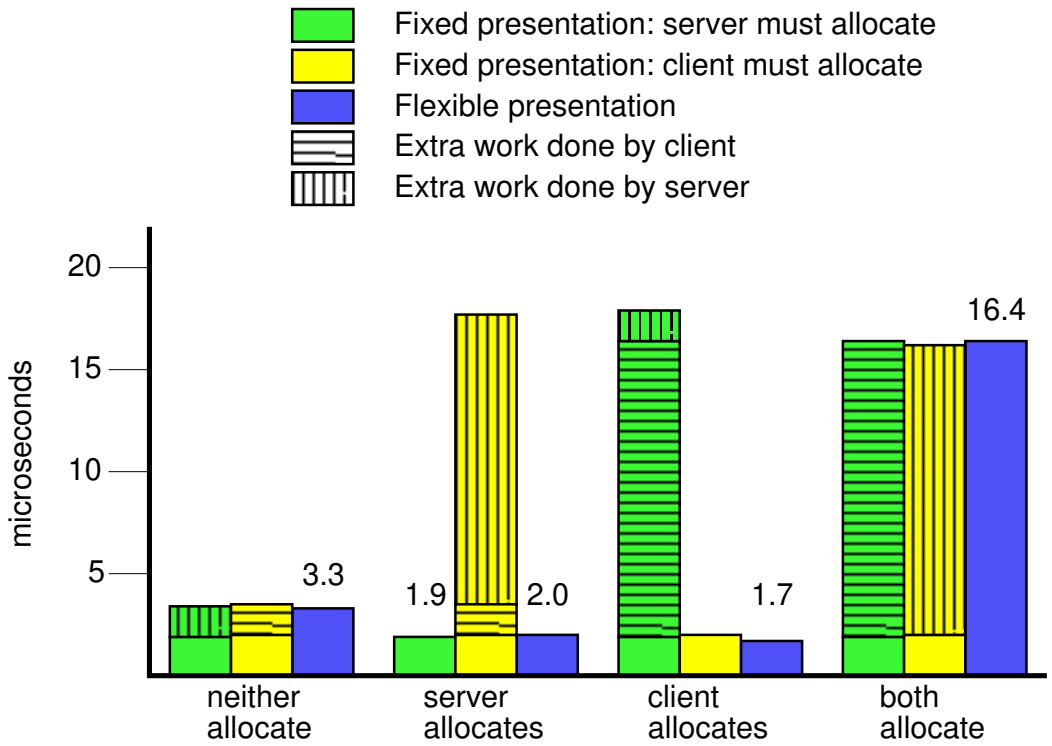

Figure 11: Performance Effects of Allocation Semantics

makes no performance difference whether the client, the server, or the stubs do it. (Of course, the programmer would probably prefer that the stubs do it.)

4.5 Leveraging Flexible Presentation with Specialized Transport Mechanisms

Our final two examples illustrate the second reason that flexible presentation can improve RPC performance: if the transport mechanism can itself be specialized on a per-connection basis, it can leverage relaxed semantic constraints at the client and server endpoints, optimally matching their requirements. Flexible presentation support provides a principled way to specify these semantic constraints, as well as an implementation framework to incorporate them into the RPC system.

We have implemented a prototype transport mechanism, that at RPC bind time specializes the code path between each client-server pair. This mechanism can leverage a variety of endpoint presentation attributes. We implemented two classes of presentation-specific transport optimizations, described below, and achieved significant performance gains over the default presentation. Both of these presentations are realistic ones, that can occur in common practice.

We give a brief summary of the transport mechanism here, but its details are not important for this paper: before a Mach port can be used for communication, as part of binding the client and server each *registers* with the kernel their respective "type signatures," generated by the IDL compiler. Once both the client and server type signatures are available, the kernel checks them against each other, verifies that the interfaces are compatible and searches for optimizations made possible by the specified presentations. It then builds a *combination signature* based on both input signatures. The combination signature includes a block which threads[15] together small blocks of code which perform key parts of the RPC, such as register saving and restoring. The binding time work needn't be done by the kernel; a privileged user process could equally

well do it. In this implementation, the Mach kernel does most of the work of marshaling and unmarshaling data as well as transferring it across protection boundaries.

Relaxing Mach's unique-name requirement: In our first example of relaxed semantic constraints, we addressed constraints on *object names* in Mach. The standard Mach IPC system always enforces the semantic requirement that all references to a particular port from a particular task have only a single *name* within that task. This requirement is required in some situations, such as authentication, but is unnecessary for simple object invocation. It substantially slows down transfer of object references (Mach port rights—capabilities) from one task to another, since it commonly requires a number of operations, including looking the name up in a hash table, adding the name if not present, or incrementing the reference count. These operations invoke many layers of function calls and are surprisingly expensive. The single-name requirement is clearly a presentation feature, since it only affects the *appearance* of a port locally within a task. Simply leveraging the relaxed presentation semantics of not needing unique port names `[nonunique]` resulted in a significant performance improvement when passing a single port between two tasks: a reduction from 32.4 μsecs to 24.7 μsecs, or 24%.

Varying Trust Parameters: In our second example of relaxed semantic constraints, we addressed the varying degress to which programs may *trust* each other. The trust relationships between clients and servers count as "presentation" for our purposes, because they are useful in optimizing RPC but do not affect the "network contract." Therefore, in our RPC system, we allow each "side" of a connection to specify the degree to which it trusts the other side. Three levels of trust are provided: (i) no trust (the default), (ii) trust of confidentiality (allows leakage of information *to* the other side) but not integrity (protects against corruption *by* the other side) `[leaky]` and (iii) full trust of both integrity and confidentiality (allows both information leakage and corruption) `[leaky, unprotected]` The middle level would typically be used between processes owned by the same user, which want to maintain protection boundaries for robustness, but have no information to hide from each other. The full trust level could be used by clients communicating with privileged servers, such as a Unix personality server.

We implemented relaxed trust levels by requiring less register saving, restoring, and clearing on the RPC path. Figure 12 shows the performance of null RPC for all trust combinations; the overall improvement from slowest (no trust, back corner) to fastest (full trust, front corner) is about 30%.[6]

5 Future Work

Having demonstrated the value of flexible presentation, we plan to use it aggressively to improve performance in a real system: the Mach kernel, the Lites Unix server, and applications. We will exploit its ability to hide the difference between different transport mechanisms as we experiment with new IPC mechanisms.

An interesting area for research is to extend the varieties of semantic attributes on interfaces to cover issues like resource consumption and resource locking behavior and requirements. Our framework could be used to explore the flexible composition of interfaces and modules, while maintaining correctness.

Flexible presentation offers the opportunity to narrow interfaces, by combining presentation variants of logically identical procedures. At the same time, it offers wider functionality, by allowing a wide variety of presentation variants to be applied to most RPC parameters, by any client of that interface. Evolving existing interfaces in this way should be a useful direction.

[6] The right-most columns (leaky, unprotected) are the same because our mechanism does nothing additional when a server claims to trust a client's correctness (unprotected).

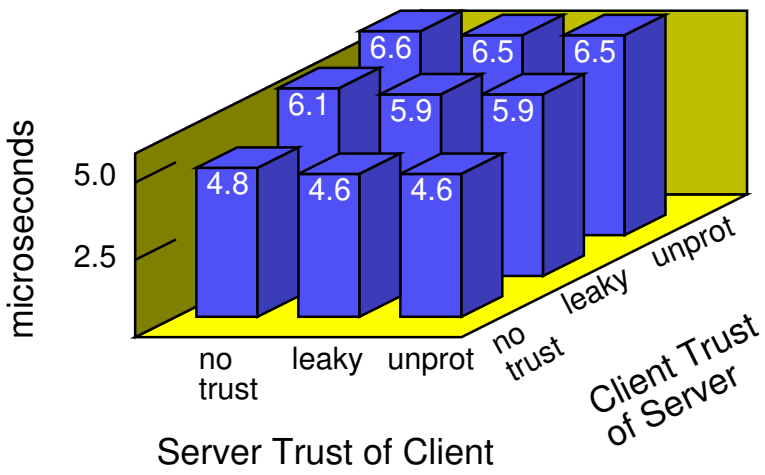

Figure 12: Performance effect of varying trust parameters

6 Conclusion

In this paper we presented a variety of ways in which flexible presentation can be used to improve RPC performance; these are only examples, and many others are possible. Understandably, the performance improvements are more noticeable for RPC over faster, lighter-weight underlying transport protocols: ranging from only 3% for Sun RPC over Unix sockets to an order of magnitude for lightweight same-domain communication. This leads us to believe that flexible presentation support will be most important in two domains: highly decomposed and microkernel-based operating systems that support extremely fast IPC mechanisms, and in very high speed networking.

Acknowledgements

Thanks to John Carter for reviewing this paper, to Jeffrey Law for his careful timings, to Nathan Dykman for his careful survey of interfaces, and to Gary Lindstrom for exploration of the language issues.

References

[1] J. S. Auerbach et al. Concert/C: A language for distributed programming. In *Proc. of the Winter 1994 USENIX Conference*, pages 79–96, Jan. 1994.

[2] J. S. Auerbach and J. R. Russell. The Concert signature representation: IDL as an intermediate language. In *Proc. of the Workshop on Interface Definition Languages*, pages 1–12, Jan. 1994.

[3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, Feb. 1990.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.

[5] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, 1990.

[6] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, 1993.

[7] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan, June 1992.

[8] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. of the Winter 1994 USENIX Conference*, pages 97–114, Jan. 1994.

[9] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proc. of the Summer 1990 USENIX Conference*, pages 87–96, Anaheim, CA, June 1990.

[10] G. Hamilton and P. Kougiouris. The Spring nucleus: a microkernel for objects. In *Proc. of the Summer 1993 USENIX Conference*, pages 147–159, Cincinnati, OH, June 1993.

[11] G. Hamilton, M. L. Powell, and J. J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, 1993.

[12] J. Helander. Unix under Mach: The Lites server. Master's thesis, Helsinki University of Technology, 1994. 64 pp. Also http://www.cs.hut.fi/~jvh/lites.MASTERS.ps.

[13] ISO. *Information Procesing Systems — Open Systems Interconnetion — Basic Reference Model*. ISO-7498, 194.

[14] M. K. Johnson. *LINUX Kernel Hackers' Guide*. ftp://tsx-11.mit.edu:/pub/linux/docs/linux-doc/-project/khg-0.6.ps.gz, 1993. 119 pp.

[15] P. Klint. Interpretation techniques. *Software — Practice and Experience*, 11(9):963–973, Sept. 1981.

[16] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley, Reading, MA, 1989.

[17] J. Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[18] Microsoft Corporation and Digital Equipment Corporation. *Common Object Model Specification*, July 1994. 62 pp.

[19] Object Management Group. The Common Object Request Broker: Architecture and specification. Draft 10 Rev 1.1 Doc # 91.12.1, OMG, Dec. 1991.

[20] S. O'Malley, T. Proebsting, and A. B. Montz. USC: A universal stub compiler. In *Proc. of the SIGCOMM '94 Symposium*, Aug. 1994.

[21] Open Software Foundation. *OSF DCE Application Development Guide*. Prentice Hall, 1993.