The Spring nucleus: A microkernel for objects

Graham Hamilton Panos Kougiouris

Sun Microsystems Laboratories, Inc.
Mountain View, CA 94043 USA

Abstract

The Spring system is a distributed operating system that supports a distributed, object-oriented application framework. Each individual Spring system is based around a microkernel known as the nucleus, which is structured to support fast cross-address-space object invocations.

This paper discusses the design rationale for the nucleus's IPC facilities and how they fit into the overall Spring programming model. We then describe how the internal structure of the nucleus is organized to support fast cross-address-space calls, including some specific details and performance information on the current implementation.

1. Introduction

Currently there is considerable interest in both industry and academia in structuring operating systems as sets of cooperating services rather than as single monolithic programs. This trend is driven by two main motivations. First, there is a belief that it will be easier (and faster) to develop and modify OS components if they are required to be cleanly isolated from one another. Second, there is a desire to blur the line between operating system components and regular applications so that third party applications can provide functionality (such as naming or paging) that was formerly thought of as part of the monolithic OS.

The Spring operating system is a distributed operating system that is focused on developing strong interfaces between OS components and on treating OS components as replaceable, substitutable parts. To aid in these goals, we have chosen to represent system resources as objects and to define all our system interfaces in an object-oriented interface definition language. Additionally, the OS is structured around a microkernel with most major system functions – such as file systems, pagers, or network software – implemented as application level services on top of this microkernel.

One of the key problems in microkernel systems is providing sufficiently fast inter-process communication (IPC), so that the OS does not suffer a major performance loss by being split into a set of different address spaces. In Spring, we were interested in developing an IPC mechanism that was highly efficient and that also meshed well with our object-oriented application programming model.

This paper describes the reasoning behind our IPC model and describes how the nucleus (our microkernel) is structured so as to provide extremely efficient cross-address-space object invocation. We also provide a detailed performance analysis of our fast-path call mechanism and compare our performance with that of related work.

1.1. Short overview of Spring

Spring currently exists as a fairly complete prototype. Two components of the OS run in kernel mode. One of these is the virtual memory manager, which provides the core facilities for paged virtual memory [Khalidi & Nelson 1993A]. The other is the microkernel proper, known as the nucleus, which provides the basic primitives for domains (the Spring analogue of UNIX® processes) and threads.

Functionality such as file systems, naming, paging, etc., are all provided as user-mode services on top of this basic kernel. These services are provided as dynamically loadable modules and it is only at system boot time that the OS makes the final decisions about which modules are loaded onto which domains. Typically, when we debug core services, we will boot with a debug switch on so that all services are loaded in separate domains. In more normal use, the system start-up code will cluster sets of related services to run in the same domain.

All the inter-service interfaces are defined in our interface definition language, which supports an object-oriented type system with multiple inheritance.The system is inherently distributed and a number of caching techniques are used to boost network performance for key functions.

The system provides enough UNIX emulation to support standard utilities such as make, vi, csh, the X window system, etc. [Khalidi & Nelson 1993B].

1.2. Related work

Many operating systems provide some form of message passing interprocess communication. Recent examples include sockets in Berkeley UNIX [Leffler et al 1989], ports in Mach [Acetta et al 1986], and ports in Chorus [Rozier et al 1992]. It is possible to provide a procedural call and return model based on such message passing facilities, but the fundamental model is of distinct threads reading messages and writing replies. In many of these systems it is possible for processes to pass around access rights to communication endpoints so as to grant controlled access to given resources.

Some operating systems provide direct support for procedural IPC. Multics provided a facilities called *gates* that enabled secure procedural communication between different protection rings within a process [Organick 1972]. More recently, several variants of the Taos system have provided explicit cross-address-space procedure call facilities with high performance [Bershad et al 1990], [Bershad et al 1991].

The Microsoft® NT system [Cutler 92] provides an interesting IPC mechanism known as event pairs that is specialized for cross-address-space calls. If a given pair of threads in different address spaces agree to use a particular event pair for communication, then one of the threads can act as a high performance server for the other thread's cross-address-space calls.

2. The Spring IPC model

2.1. Objects in Spring

Spring attempts to avoid mandating how objects should be implemented. Different applications should be able to implement objects in different ways. This includes not just the implementation of particular object methods, but also the implementation of the object machinery itself, such as object marshalling and even object invocation. For example, not all objects are required to support cross-domain calls. Objects may chose to support cross-domain calls via a variety of different mechanisms (for example using shared memory in addition to IPC). Many objects provide different mechanisms for cross-domain calls and for intra-domain calls. However, despite this ocean of flexibility, it became clear that there are certain commonly desired properties that could benefit from specific OS support.

The most important property is secure access to services. We do not want to perform a full scale authentication check whenever a client invokes a protected object. Nor do we want to restrict access to an object to a certain preordained set of clients. If a client has legitimately acquired access to an object, then we would like that client to be able to pass its access on to third parties, who will then be able to operate on the object. This security requirement quickly led us to use a software capability model for providing secure access to specific objects. This approach is sim-

ilar to that used in the Cambridge fileserver [Birrell & Needham 1980], Amoeba [Tanenbaum et al 1986], and Mach [Acetta et al 1986], [Sansom et al 1986].

Following from this desire for security are some secondary requirements. When a cross-address-space call occurs on a capability, neither the client nor the server should be vulnerable to the other's incompetence or malice. Thus, for example, we require that there is well defined behaviour if either the client or the server crashes. Similarly, we must be able to debug either the client or the server, without unduly disrupting the other.

Finally, there are performance considerations. Spring is a system that makes heavy use of cross-address-space calls. Thus we require a mechanism that is highly efficient, particularly for the common cases where the number of arguments and results is small.

2.2. Doors

Doors are a Spring IPC mechanism resembling in many aspects the gates mechanism of Multics [Organick 1972] or the cross-address space-call mechanisms of Taos [Bershad et al 1990], and in other aspects the communication endpoint notions of sockets in BSD UNIX [Leffler et al 1989] or ports in Mach [Acetta et al 1986].

The design for doors progressed through several stages. The earliest designs were for a primitive mechanism for transferring control and data between different address spaces. A domain D1 could create a door designating a particular entry point into its address space. If this domain passed the door on to another domain D2, then a thread in D2 could jump through the door, causing it to arrive at the given entry point in D1. Doors acted as capabilities, in that a domain that obtained access to a door could pass that access on to other domains, and domains without access to a door could not fabricate access. This primitive notion of doors had an appealing simplicity. However, this original design suffered two significant flaws, one major and one minor.

The minor flaw was that a door specified only a PC entry point. Since we were dealing with objects and a given server might support a wide variety of different objects, granting different access to different clients, we would normally want to be able to identify which particular object a given door granted access to. This problem could be solved by dynamically creating short sequences of code as needed to act as entry points for given objects. However, we preferred a solution that avoided the allure of fabricating code at run-time.

The major flaw was that a jump through a door implied no return path. Since we were implementing a system with an object oriented call/return model, we would normally want a server to be able to (securely) return to its callers. The obvious solution was for the caller to pass a return door along as an argument to the door call. Unfortunately, in order to sidestep some security issues caused by malicious clients potentially issuing multiple returns, we needed to be fairly careful about how we managed return doors, possibly to the point of needing to create and delete a distinct return door for every distinct call. This seemed undesirable.

Therefore, we extended the notion of doors to reflect our object-oriented call and return model. A door now represents an entry point for a cross-domain call. Associated with the door are both an entry point PC and an integer datum that can be used to identify a particular object in the target domain.

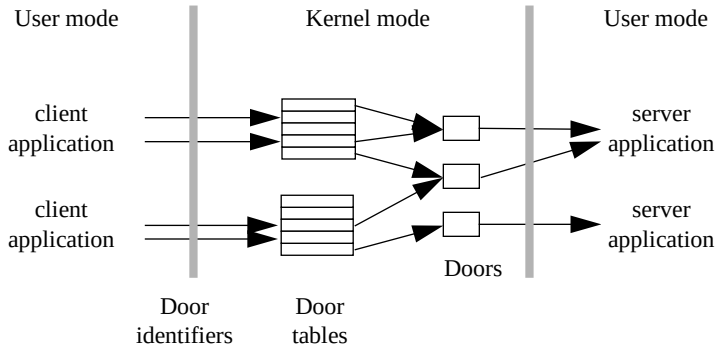
If a domain is granted access to a door, one of its threads may subsequently issue a cross-domain call through the door. This causes a transfer of control into the domain that created the door at the PC specified by the door and with the door datum in a specified register. When the nucleus executes the door call it also records some return information within the nucleus. This return information consists of the caller's domain, the PC to which to return to after the call, and the stack pointer of the caller at the time it issued the call. Then after the call has completed, the called domain will execute a door return. In executing the door return, the nucleus will use the return information that was recorded during the call and return to the callers' address space at the recorded return PC, with the recorded return stack pointer.

When a door call arrives in a server's address space, the server will typically use the datum value to locate the target object and execute an object call. (In practice, there is typically a level of indirection between the datum and actual language level objects in order to permit dynamic interposition on objects for debugging and performance analysis.)

2.3. Door tables

Doors are pieces of protected nucleus state. Each domain has a table of the doors to which the domain has access. A domain references doors using *door identifiers*, which are mapped through the domain's door table into actual doors. A given door may be referenced by several different door identifiers in several different domains.

FIGURE 1. Doors and door tables



Possession of a valid door identifier gives the possessor the right to send an invocation request to the given door. A valid door identifier can only be obtained with the consent of the target domain or with the consent of someone who already has a door identifier for the same door. As far as the target domain is concerned, all invocations on a given door are equivalent. It is only aware that the invoker has somehow acquired an appropriate door identifier. It does not know who the invoker is or which door identifier it has used.

2.4. Reference counting

There is one last feature of doors that we should mention. Doors are typically used to provide access to application level objects such as files or I/O streams. Many servers are interested in knowing when all the customers for a given object go away, so that they can safely destroy the object and free any resources that the object is using in the server's address space. It is not sufficient for clients to issue explicit deletes on objects, as this does not cope with client crashes and it is difficult for clients to issue explicit deletes when objects are shared between domains.

Therefore we provide a simple reference counting mechanism for doors. Whenever a domain is granted access to a door, we increment the door's reference count. Whenever a domain loses access to a door (either voluntarily, or due to the domain crashing, or due to the door being revoked), we decrement the reference count on the door. When the reference count is decremented from 2 to 1, the nucleus notes that it should deliver an "unreferenced" call on the door. We use 1 rather than 0, as it is normal for an object manager to retain a reference to each door that it implements.

The nucleus keeps a queue of unreferenced doors for each domain, and is prepared to use a single nucleus thread per domain to process this queue and call into the domain to deliver an unreferenced call on each of these unreferenced doors. We originally simply created threads to deliver each unreferenced call, but it is common that large num-

bers of doors may become unreferenced simultaneously (for example, due to a client domain exiting) and having all these calls delivered in parallel caused an unnecessary resource crunch.

To avoid races between incoming calls on a door and the door becoming unreferenced, the nucleus also increments the door reference count when sending a call through a door and decrements it when the call returns. We were extremely reluctant to add this extra cost to the basic nucleus call and return code, but a scrutiny of various race conditions where an unreferenced call might arrive (causing the application to discard its object state) just ahead of a legitimate incoming call (which tries to reference that state) convinced us that we needed to solve the problem, and that we could provide a satisfactory solution more cheaply in the nucleus than in user-land.

A domain that implements a door can at any point revoke access to that door. This will invalidate all access to the door and will cause an unreferenced call to be delivered as soon as there are no longer any active calls on the door.

3. The Spring thread model

3.1. Threads and shuttles

Because Spring is a multi-threaded system we wanted to provide application programs with ways of examining and manipulating threads. Naturally enough, threads are represented as nucleus objects, accessible via cross-domain calls into the nucleus.

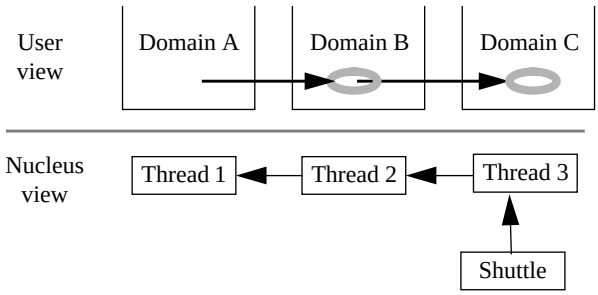
When we issue a cross-domain call from one domain into another domain, we could present this to application software either as a single thread, spanning two domains, or as two threads related by a call chain. The obvious implementation is to represent a chain of cross-domain calls as a single thread. This is, after all, what the nucleus is really doing. However, there are two problems with this.

First, Spring is a distributed system. Network servers, known as proxies, may forward cross-domain calls to remote machines. So, at the application level, several machines' nuclei may be involved in a single cross-domain call chain. Thus, if we wanted to represent a cross-domain call chain as a single thread, some fancy footwork would be necessary to keep the different nuclei in step. This becomes particularly interesting when one of the intermediate nuclei in a call chain crashes. (See the discussion of failure semantics in Section 3.3 below.)

Secondly, security and integrity issues mean that we want to limit the ways in which domains can manipulate one another. Say that a user domain "foo" issues a call into a file server domain. If a debugger should be attached to "foo" and attempt to stop all the threads in "foo" (a perfectly reasonable desire), then we do not want the call that is executing in the file server to be stopped — this call may be in the middle of important updates that will affect many domains in the system. Thus, we wanted to limit the effects of the thread control operations to within a particular domain. If a debugger "stops" a thread in domain D1 that has called into another domain D2, then this should only

affect the thread's behaviour in D1, not in D2. The call in D2 should continue executing as before; it is only when it returns to D1 that it should become stopped.

FIGURE 2. Domains, threads and shuttles



Thus, we chose to adopt a model where each application visible thread object describes a thread within a given domain. A cross-domain call chain consists of a series of these application visible threads. Access to thread objects is restricted so that an application program running on behalf of user X can normally only get access to and manipulate threads within other domains also belonging to user X.

We use the term "shuttle" to refer to the true nucleus schedulable entity that supports a chain of application visible threads. So when we execute a cross-domain call, we change the apparent application visible thread, but we retain the same nucleus shuttle, which contains the real state of the schedulable entity. (See Figure 2 .)

3.2. Server threads

To service an incoming call, a domain needs to make certain resources available, most notably a stack, but also such minutiae as a region for thread-local storage, a reception buffer for incoming arguments, etc. Originally, we hoped to do this entirely at user-level by having an incoming call allocate itself the necessary resources from a resource pool managed at user-level. However, this approach proved fairly cumbersome. Since several calls might arrive simultaneously, there needed to be some degree of synchronization over resource allocation. If the resource pool becomes empty, then incoming calls might either want to wait until some resources become available or try to allocate more. Doing all of this without having a stack rapidly becomes tedious.

Therefore, we decided that at the very least we wanted an incoming call to arrive in its target domain with a valid stack pointer. It also seemed desirable that all incoming calls should also be immediately associated with application visible "thread" objects so that they can be debugged and controlled.

Therefore, we arranged that Spring applications can explicitly create pools of application level threads to service incoming requests. Such threads are referred to as "server threads". When creating a server thread, the creating domain must specify a stack pointer value and a memory area to be used for incoming arguments. The nucleus will create an application visible thread object to describe this new server thread and will allocate an internal thread descriptor in the nucleus and associate it with the given domain.

So, when delivering a cross-domain call, the nucleus attempts to allocate a server thread for the target domain. If it succeeds, it then delivers the data into the buffer associated with that server thread and leaps into the target domain with the previously registered stack pointer value in the stack pointer register. Unbeknown to the nucleus, the user-level thread libraries have cunningly salted away a few key values (such as a pointer to a thread private storage area)

on this stack, and they use these values to create a full language level environment before leaping off to a higher level object pointed to by the datum value.

Things become a little more interesting if there are no server threads available for a given process when a cross-domain call is delivered to it. We considered a couple of alternatives. We could simply return to the caller with an error code. This seemed fairly undesirable, as normally the caller will not want to have to recover from what may be a temporary overload on the server. Alternatively, we could suspend the calling thread in the nucleus until a server thread becomes available. This seems like a partial solution, and we did indeed provide this facility. However we were concerned that there might be a danger of deadlock if two domains are each waiting for a call into the other domain to complete. So, we also provided a mechanism whereby a domain can find out that it has run out of server threads and, if it wishes, create more. This is done by providing an operation on a domain object where a thread can call and block until the domain runs out of server threads. In practice, what the standard application libraries do is to initially avoid creating any server threads, but merely create a thread to watch for a shortfall and then create threads (up to an application specifiable limit) as needed.

3.3. Failure semantics

When we are executing a cross-domain call, we must be prepared to cope with either the caller or the called domain crashing. In the event of the called domain crashing, the nucleus simply treats this as an involuntary return on all the active incoming calls and forces the calls to return with an error code.

Crashes by caller domains are rather more interesting. We clearly don't want to do anything rash to affect the currently executing call as it may be in the middle of an important activity. Therefore we arrange that the call will return to the nucleus rather than to the true caller so that we can remove the crashed domain from the system.

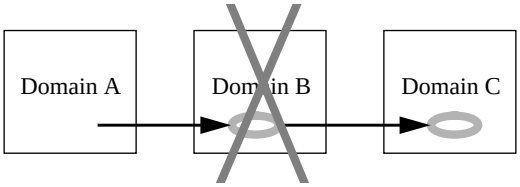
However, the called domain may be executing some lengthy activity on behalf of the caller or it may be sleeping on a resource queue (such as a tty input queue) with the intention of eventually removing some resource and returning it to the caller. So we would like to provide some non-intrusive way for a called domain to optionally detect that its client has gone away and thus perhaps abort whatever action it is performing.

We have chosen to use an alert mechanism similar to that provided by the Taos system [Birrell et al 1987]. Briefly, associated with each thread is a single bit specifying whether that thread is alerted. Whenever a thread sleeps it can specify that it should be awoken from that sleep with an exception if it is alerted. A thread can also poll its alerted status explicitly and set or clear it as it pleases. Or, by default, a thread may simply decide to ignore alerts all together.

In Spring, whenever a thread becomes alerted, we propagate that alert forward down its call chain to any thread it has called and to any subsequent threads. (This includes propagating the alert over the network if necessary.) Thus, when the nucleus terminates a domain, it first alerts all the threads in that domain, which will propagate an alert to any called threads. By convention, if a server thread sees an alert, it should interpret it as a mild suggestion from someone earlier in the call chain that perhaps circumstances have changed and it would be wise to return home for clarification and new orders.

Crashes in the middle of call chains are particularly interesting (see Figure 3). When such a call occurs we can immediately transmit an alert to the threads downwind of the crash, but what should we do about the threads before the crash? We could wait until the call chain attempts to return into the crashed domain and then engineer an error return to the preceding domain. However, this will mean that there will be an indeterminate delay until the preceding domain is notified that its call has failed.We decided to avoid this. When a crash occurs in the middle of a call chain, we immediately split the call-chain into two separate chains by creating a new shuttle to handle the call chain before

FIGURE 3. A crash in the middle of a call chain



the crash. This new shuttle does an immediate error return. The old shuttle, handling the call chain after the crash, is redirected to return into the nucleus rather than into the crashed domain.

4. Details of door invocation

We currently support three different flavours of door invocation and door returns.

The first flavour, known as the fast-path, supports the case when all the door arguments are simple data values and total less than 16 bytes. In practice, this is the dominant case for both calls and replies in Spring (see Section 5.1 below) so its performance is extremely important.

The second flavour, known as the vanilla-path, supports the case when less that 5 KBytes of data and some moderate number of doors are being passed as arguments or results. In this case, the data is copied through the nucleus.

The third flavour, known as the bulk-path, supports the case when we are sending entire pages as data or results. In this case, the nucleus will use VM remapping to transmit the data. The bulk-path also supports the transmission of essentially unlimited numbers of doors.

Most application level use of the door invocation mechanisms is, in fact, built on two higher levels of software. The first level is a mechanism known as the “transport” that provides a general purpose marshalling mechanism for storing and retrieving argument data of arbitrary size. The second level consists of C++ stubs that build on top of the transport level. These stubs are automatically generated from our interface definition language. They provide a C++ interface to remote method operations and perform the mechanics of marshalling argument data into transports. Currently, all our marshalling machinery is general purpose and is targeted to support moderate amounts of data, leading to a relatively large overhead for small calls. However, since application level code only sees the interface to the C++ stubs, we plan to start generating special stubs for small calls, which will bypass the general transport machinery and leap off directly to the nucleus’s fast-path mechanism.

4.1. The fast-path

The nucleus fast-path call sequence is currently roughly 60 SPARC® [SPARC 1992A] instructions, and the nucleus return fast-path is roughly 40 nucleus instructions, permitting a complete cross-domain call in roughly 100 nucleus instructions. In practice, there also have to be a certain number of user-level instructions at each end. On the caller, some 13 instructions are unavoidable to save and subsequently restore various pieces of user state (such as a pointer to thread-private memory, the return frame pointer, etc), since the nucleus itself saves only the absolute minimal state. Similarly, at the callee, some 8 instructions are unavoidable to establish an appropriate environment for higher-level language programming and to indirect to an application-level object.

Both the fast-path call and the fast-path return only attempt to deal with the normal case where all resources are available and when neither domain has anything unusual happening to it such as being stopped for debugging or

being in the process of being terminated. This is compressed into a couple of flag tests at the beginning of each fast-path call or return. If there are any problems, then the fast-path code generates a call into the vanilla-path code to handle the full work of the call or return.

4.1.1. The fast-path on SPARC V8

Clearly, the details of the fast-path code will vary from machine architecture to machine architecture. On SPARC V8, one of the main considerations was managing the hardware register window set. To keep cross-domain calls cheap, we did not want to flush all the active register windows to memory when we carried out a call. On the other hand, for security and integrity reasons, we wanted to avoid letting the target of a call access any register windows belonging to the client.

Fortunately, the SPARC V8 architecture provides a Window Invalid Mask (WIM) that allows the kernel to prevent user-mode accessing any given register windows. So during a cross-domain call we merely mask out access to the caller's windows. As we take window overflows and write the windows out to the caller's memory, we make these windows available for the use of the called domain. A little care is required during the return sequence to ensure that the WIM is cleared correctly, and that the caller is in fact returning into the correct window.

Table 1 contains a description of the SPARC nucleus instructions required for a cross-domain call and return. The actual performance of the cross-domain path is very sensitive to cache hits and misses.

TABLE 1. Instruction breakdown of SPARC fast-path

	call	**return**	**total**
Fiddling with register windows	10	9	19
Switching thread/domain	12	11	23
Getting target info	7	0	7
Reference counting target door	3	4	7
Saving/restoring return info	6	4	10
Switching VM context	7	7	14
Miscellany	13	9	22
Total	58	44	102

To verify our understanding of this code, we obtained a full cycle by cycle trace with a hardware monitor on a SPARCstation™ 2. This showed no real surprises. With some minor exceptions, the CPU time is fairly evenly smeared across the different activities. On SPARCstation 2, both the trap entry and trap return sequences are fairly cheap, at about 7 cycles each, which is roughly the same as a cache miss. The actual instruction for switching the hardware VM context is slightly more expensive at about 14 cycles, but this still only constitutes a very small percentage of the overall cycle count. The main non-obvious cost is that saving the return information and writing the new thread/domain information both saturate the CPU's write buffer, causing stalls on store instructions until earlier writes are flushed.

We expect it should be possible to obtain similar instruction counts on most modern RISC CPUs. The main imponderable is the cost of switching the MMU context, which happens to be fairly low on SPARC.

4.1.2. SPARC V9

The SPARC V9 architecture [SPARC 1992B] is a 64-bit extension of the 32-bit SPARC V8 architecture. It is fully compatible with SPARC V8 for user-mode instructions, but has a number of significant changes to the privi-

leged mode architecture. As part of the SPARC V9 design, consideration was given to what (if anything) could be done to accelerate cross-domain calls. Fortunately, or unfortunately, the answer was comparatively little.

The cross-domain call code will benefit from several general purpose improvements to the privileged architecture that appear in V9 such as support for nested traps, and the addition of an alternate set of global registers for use by trap handlers. The only changes that were specifically motivated by cross-domain calls were some changes to the register window management model to make it easier to have windows belonging to two different address spaces in the register window file simultaneously. (Basically, this involved replacing a mask register with a small set of counter registers which are more convenient to manipulate, and adding additional trap vectors for different flavors of window overflows and underflows.)

Short of adding specific CISC like cross-domain call instructions (an option that was quickly rejected) we could identify no other changes that seemed likely to particularly benefit cross-domain calls. The cross-domain call code is, after all, mostly simple, regular RISC instructions.

4.2. The vanilla-path

In the vanilla-path cases, we take a full trap into the nucleus to copy the argument data into the target domain and to move any argument doors across. This code is fairly straightforward and has received no special tuning.

4.3. The bulk-path

The bulk-path is used to transfer large quantities of page-aligned data from a source domain to a destination domain. A thread in the source domain traps into the kernel using the bulk_call (or bulk_return) trap. The thread passes two arguments in registers: a pointer to a descriptors area and the number of descriptors in that area.

Each descriptor specifies an “indirect data block” that should be passed to the destination domain. Each indirect data block is a page-aligned region of the address space of the source domain. There are two ways to pass an indirect data block through the bulk path: either unmapping the pages from the address space of the source domain and mapping them into the address space of the destination domain; or by mapping the pages in both the domains using copy-on-write semantics. The nucleus uses virtual memory services [Khalidi & Nelson 1993A] to satisfy calls that go through the bulk path.

The bulk path was added recently, as an extension to the basic cross-domain call mechanism. It is currently used in the libraries to provide the illusion that a practically unlimited number of bytes and door identifiers can be transferred through the kernel.

5. Performance

5.1. Marshalled arguments size

As we mentioned earlier, our belief was that in most cases a cross-domain transfer would copy a few bytes and no door identifiers. Such transfers follow the optimized fast call and return paths. In addition we expected that in the most common cases, an object invocation would follow both the fast call and return path. We based these assumptions on the structure of our interfaces and the work of other researchers [Bershad et al. 1990]. Early results show that these assumptions were correct.

In the current implementation of Spring, we measured and classified the number of cross-address-space control transfers (both calls and returns) in three cases: when we boot the basic Spring services, when we "make" a medium sized C++ program, and when we start an X11/OpenLook based desktop. We present the results in Tables 2 and 3.

TABLE 2. The number of calls observed in a Spring system under three different loads. (In thousands)

Cases	make ssh	start xnews	boot spring	total
fast calls	393 (84%)	91 (83%)	22 (92%)	507 (84%)
vanilla calls	75 (16%)	19 (17%)	2 (8%)	96 (16%)
total	468 (100%)	110 (100%)	24 (100%)	603 (100%)

TABLE 3. The number of returns observed in a Spring system under three different loads. (In thousands)

Cases	make ssh	start xnews	boot spring	total
fast returns	338 (72%)	77 (70%)	18 (72%)	433 (72%)
vanilla returns	130 (28%)	33 (30%)	7 (28%)	170 (28%)
total	468 (100%)	110 (100%)	25 (100%)	603 (100%)

Table 4 shows the percentage of calls and returns that passed various quantities of data. The majority of both calls and returns involved less than 16 bytes of data.

TABLE 4. Distribution of the number of bytes passed per call or return for the combined load.

number of bytes passed	percentage of calls	percentage of returns
0-16 bytes	85.0 %	81.8 %
16-100 bytes	11.5 %	13.7 %
100-1000 bytes	2.3 %	2.8 %
1000-4000 bytes	0.0 %	0.4 %
4000-5000 bytes	1.2 %	1.3 %

In addition, approximately 2.5 % of calls and 6 % of returns passed one or more kernel door identifiers.

In conclusion, our early results show that the bulk of the control transfers involve very few bytes and no doors, and consequently it was worthwhile to optimize these cases in our nucleus IPC mechanism.

5.2. Performance for small calls

Table 5 shows the costs for minimal cross-address-space calls in a number of systems including Taos LRPC [Bershad et al 1990], Mach [Draves et al 1991], and NT [Cutler 1992]. All times are for bare calls on uniprocessors and exclude stubs or other higher level software costs.

TABLE 5. Minimal cross-address-space call times.

	raw time	Spec 89	scaled time
Berkeley sockets (Sparcstation 2)	850 µs	25	21000
Taos LRPC (CVAX Firefly)	129 µs	3.5	450
Mach 3.0/MK40 (DECstation 3100)	95 µs	12	1100
NT event pairs (50MHz R4000)	18 µs	70	1200
Spring cross-domain call (SPARCstation 2)	11 µs	25	275

Since the benchmark machines vary considerably in raw CPU performance, we provide both raw times and times scaled by the CPU's Specmark89 speed. (Since we lack Specmark numbers for the exact machines used for the Taos and NT benchmarks, we use Specmark numbers from product systems using the same CPU chips.) However, OS performance does not necessarily scale linearly with raw CPU speed [Ousterhout 1990] [Anderson et al 1991] and the scaled numbers should be regarded as only an extremely approximate guide to relative performance.

5.3. Performance discussion

In traditional IPC systems, such as Berkeley sockets, there are three major costs:

First, there is the cost of making scheduling and dispatching decisions. Typically, the thread that issued the request will go to sleep, and the OS kernel will make a careful and objective decision about which thread to run next. With a little luck this will actually be the thread that will execute the request. This thread will then run and upon completion of the call, it will wake up the caller thread and put itself to sleep. Once again, the OS kernel will make a careful and scholarly scheduling decision and will, hopefully, run the caller thread.

Second, there is the cost of performing the context switch between the calling address space and the called address space. At its worst, this will involve a full save of the CPU registers (including floating point registers), a stack switch, and then a restoration of CPU state.

Third, there is the cost of moving the argument and result data between the caller and the callee. Traditional kernels tend to use buffer management schemes that are optimized for passing moderately large amounts of data.

Recent microkernel IPC systems have pushed back on all three of these costs. By assuming an RPC call and return model, it is possible to perform a direct transfer of control between the caller and the callee threads, bypassing the scheduler. Given such a direct transfer, it is also possible to avoid the full costs of a context switch and only save the minimal information that is necessary for an RPC return. By exploiting the fact that most argument sets are small (or that if they are large then they are passed through shared memory), it is possible to avoid buffer management costs.

Different systems vary in the degree to which they have succeeded in minimizing these costs. For example, the Taos LRPC system modelled a cross-domain call as a single thread of execution that crossed between address spaces, thereby avoiding and dispatching or scheduling costs. However, both Mach and NT model the callee threads as

autonomous threads which simply happen to be waiting for a cross-process call. This leads to a certain amount of dispatcher activity when a cross-process call or return occurs.

The Spring nucleus has attempted to minimize all three costs. The nucleus' dispatcher works in terms of shuttles, which do not change during cross-domain calls. There are, therefore, no scheduling or dispatching costs during a cross-domain call. Only an absolutely minimal amount of CPU state is saved (basically a return program counter and stack pointer). We do not attempt to save the current register window set, or attempt to switch to a different kernel stack. The fast-path is optimized for the case where all the arguments are passed in registers or shared memory, so the nucleus need not concern itself with buffering arguments or results. In addition, the fast-path code is executed directly in low-level trap handlers and avoids the normal system call costs.

A more mundane factor in our IPC performance is that our nucleus data structures have been tailored to optimize their performance for cross-domain calls. For example, during a cross-domain call there is no need to check that the target door identifier is valid. Instead, a simple mask and indirect load is performed. If the target door identifier was invalid we will get a pointer to a special nucleus door entry point which always returns an error code. Similarly there was an effort to concentrate the number of flags that the fast-path call and return code would need to test into a single per-thread "anomalous" flag.

If we were prepared to ignore security or debugging issues, we could probably shave several more microseconds off our fast-path time. For example, we have to pay several instructions to prevent the callee thread tampering with register windows belonging to the caller thread. Similarly, during both call and return we are prepared to cope with threads that have been stopped for debugging. However, for our desired semantics, we believe we are fairly close to the minimum time required for a cross-domain call.

6. Conclusions

We have provided a high performance cross-address-space communication facility, doors, which efficiently supports the object-oriented communication required for the Spring operating system.

7. Acknowledgments

We would like to thank Marcel Janssens for obtaining the low level timing measurements described in Section 5.1.1, and Yousef Khalidi for implementing the Spring VM support for the bulk-path data transfers.

8. References

[Acetta et al 1986] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young. "Mach: A New Kernel Foundation For UNIX Development." Summer USENIX Conference, Atlanta, 1986.

[Anderson et al 1991] T. E. Anderson, H. M. Levy, B. N. Bershad and E. D. Lazowska. "The Interaction of Architecture and Operating System Design." Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, April 1991.

[Bershad et al 1990] B. N. Bershad, T. E Anderson, E. D. Lazowska and H. M. Levy. "Lightweight Remote Procedure Call." ACM Transactions on Computer Systems 8(1), February 1990.

[Bershad et al 1991] B. N. Bershad, T.E Anderson, E. D. Lazowska and H. M. Levy. "User-Level Interprocess Communication for Shared Memory Multiprocessors." ACM Transactions on Computer Systems 9(2), May 1991.

[Birrell & Needham 1980] A. D. Birrell and R. M. Needham. "A Universal File Server." IEEE Transactions on Software Engineering, 6(5), September 1980.

[Birrell et al 1987] A. D. Birrell, J. V. Guttag, J. J. Horning and R. Levin. “Synchronization Primitives for a Multiprocessor: A Formal Specification.” Proceedings of the 11th ACM Symposium on Operating Systems Principles, Austin, Texas, November 1987.

[Cutler 1992] D. N. Cutler. “NT.” Presentation at the USENIX Workshop on Micro-kernels and Other Kernel Architectures, Seattle, April 1992.

[Draves et al 1991] R. P. Draves, B. N. Bershad, R.F. Rashid and R. W. Dean. “Using Continuations to Implement Thread Management and Communication in Operating Systems.” Proceedings of the 13th ACM Symposium on Operating Systems Principles, Pacific Grove, California, October 1991.

[Khalidi & Nelson 1993A] Y. A. Khalidi, M. N. Nelson. “The Spring Virtual Memory System.” Sun Microsystems SMLI TR93-9, March 1993.

[Khalidi & Nelson 1993B] Y. A. Khalidi, M. N. Nelson. “An implementation of UNIX on an object-oriented operating system.” Proceedings of the Winter USENIX Conference, San Diego, January 1993.

[Leffler et al 1989] S. Leffler, M. McKusick, M. Karels and J. Quaterman. “The Design and Implementation of the 4.3BSD UNIX Operating System.” Addison-Wesley, 1989.

[Organick 1972] E. I. Organick. “The Multics System: An Examination of Its Structure.” The MIT Press, Cambridge, Massachusetts, 1972.

[Ousterhout 1990] J. Ousterhout. “Why Aren’t Operating Systems Getting Faster as Fast as Hardware?” Proceedings of the Summer USENIX Conference, Anaheim, June 1990.

[Rozier et al 1992] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillermont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard and W. Neuhauser. “Overview of the Chorus Distributed Operating System.” Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures, Seattle, April 1992.

[Sansom et al 1986] R. D. Sansom, D. P. Julin and R. F. Rashid. “Extending a Capability Based System into a Network Environment.” SIGCOMM ‘86 Symposium On Communications Architectures & Protocols, Stowe, Vermont, August 1986.

[SPARC 1992A] SPARC International. “The SPARC Architecture Manual Version 8.” Prentice-Hall, 1992.

[SPARC 1992B] SPARC International. “The SPARC Architecture Version 9.” 1992.

[Tanenbaum et al 1986] A. S. Tanenbaum, S. J. Mullender and R. van Renesse. “Using sparse capabilities in a distributed operating system.” Proceedings of the 6th IEEE International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986.