Efficient Cross-domain Mechanisms for Building Kernel-less Operating Systems *

Dave Probert and John Bruno

Computer Science Department
University of California
Santa Barbara, California 93106

contact: probert@cs.ucsb.edu

Abstract

We describe a set of efficient cross-domain mechanisms that allow operating systems to be implemented as cooperating applications, eliminating the need for a monolithic kernel. Our implementation, called SPACE[1, 2], can achieve higher-performance than kernel-based systems by allowing applications to build customized system services and tailor system interfaces for performance. On the SPARC architecture we have measured minimal application-to-application system service calls that are 5 times faster than Solaris *getpid()*, and customized thread creation that is 50 times faster than minimal Solaris threads.

SPACE unifies exception handling and cross-domain calls into a single mechanism that allows applications to efficiently interface to the underlying hardware. Conventional memory-management hardware is used to provide multiple protection *domains* within each address *space*. Cross-domain calls are implemented by *portals*, which map an exception in one domain to a handler in another. Portals are accessible on a per-domain basis, providing capability-based access to system services. The portals are implemented by synthesizing code according to templates for each *portal type*. The portal types are designed to minimize state saving and eliminate the dynamic binding overhead of generic cross-domain implementations.

1 Introduction

Most contemporary operating systems are structured around a central monolithic kernel (or micro-kernel). An alternative is to construct operating systems without a kernel. In this approach, operating system abstractions (processes, threads, virtual memory, communication) are implemented as application-level code, with the attendant flexibility and extensibility that many user applications require.

We are building such a kernel-less operating system, using optimized cross-domain calls which can reduce cross-domain invocation cost below the current cost of kernel invocations by exploiting properties that kernel-less operating systems have with respect to robustness, extensibility, security, and performance. The first three of these properties are generally accepted as benefits of the kernel-less approach. One of the primary contributions of this work is to achieve these benefits while also achieving gains in performance. Our kernel-less operating system is based on an approach we call SPACE[1] [1, 2].

* This research is supported in part by a UC MICRO grant and the Xerox Corporation.

[1] SPACE is an acronym for *Systems Programming using Address-spaces and Capabilities for Extensibility*.

1.1 Operating Systems Approaches

Operating system design is generally inseparable from hardware architecture. The success of systems like UNIX in the 1970s has brought about a convergence of processor architectures. Most recent machines support multiple paged address spaces, user/supervisor processor modes, and memory mapped I/O. Systems based on pure segmentation, hardware capabilities, or multiple rings of protection are no longer feasible. Current hardware provides domains based on page-level protection within flat address spaces. These domains are used to build process virtual memory. They are also used to structure operating systems, as we explore in the following.

1.2 Kernel-based Operating Systems

Most operating systems are constructed around a central monolithic piece of software which controls the machine at the lowest levels. This software is called the *kernel* or *micro-kernel*, depending on the level of the abstractions it implements.

There are two primary advantages of employing a kernel. First, details of the hardware architecture can be encapsulated and mapped into the appropriate operating system abstractions. Secondly, the set of heavily-used primitive abstractions deemed critical can be insolated in the kernel and be thoroughly tested and tuned for performance.

Evolving application requirements are turning the advantages of kernel-based design into liabilities. Applications require direct access to low-level hardware for I/O, interrupts, and memory management. Even seemingly primitive kernel mechanisms, such as threads and address space management, may be too general and thus too expensive to satisfy applications driven by fine-grain events (e.g. high-speed network interfaces). Novel uses of memory management hardware have been proposed for garbage collection and pointer swizzling, but are only feasible if overheads are extremely low.

The limitations imposed by kernel-based structures are exactly the flip-side of their advantages. Details of the hardware architecture often need to be encapsulated in an *application-specific* way. The primitive operating system abstractions provided by the kernel are not suited for every application. Applications with performance or functional requirements not served by existing abstractions must be able to replace them with new ones, not just build on top of the old ones if they are to achieve high-levels of performance.

1.3 Kernel-less Operating Systems

Using the protection domains in processor architectures that support kernel-based systems, it is possible to build an operating system without relying on a central kernel. What is required is some simple extensions to the hardware trap abstractions. Before discussing these mechanisms, we examine some of the goals of the kernel-less approach:

- **Robustness** The more complex software is, the more likely it is to have bugs. This rule also applies to kernels. The problem is that bugs in the kernel are fatal in most systems. Bugs in services outside the kernel are much less likely to bring down the *entire* system, and can usually be dealt with by restarting the daemon or subsystem.

  If kernel-based services are moved to application code, their robustness will improve. Multiple implementations of a particular type of service can coexist, further insulating the entire system from the failure of one component.

- **Extensibility** Support for multiple implementations of a particular service is critical for extensibility. Systems which only provide extensibilty through refinement can only deliver functionality, not performance.

  Allowing multiple implementation instances to coexist allows applications to replace a service with one better suited, or even specifically tailored to their requirements. A single application may take advantage of multiple implementations (e.g. different virtual memory implementations for different regions of addresses).

  Coexistence is also desirable when developing services, because it is no longer necessary to disable a working service in order to test the next version.

- **Security** Extensibility requires a tailorable interface to system services. Kernel-based systems typically export a single *system call interface* to all applications. Verification of access rights is handled internally to the services. Because only a fixed set of kernel services are available, there has not been the incentive to provide customizable interfaces.

  System call interfaces are implemented by mapping exceptions (i.e. trap instructions) into protected calls on system services. Such an interface *can* be customized on a per-application basis. Access to a particular cross-domain call represents a capability, analogous to the capabilities to access memory pages granted by the hardware memory management unit (MMU).

  These types of capabilities provide a run-time alternative to schemes based on *user-id* or other authentication checks. There are also performance advantages to checking access when the capability is granted rather than every time the service is accessed.

- **Performance** Performance is *the* critical goal. Application-based extensibility and capability-based security contribute to this goal. But performance also benefits from the role extensibility plays in furthering operating systems research. New ideas can be tested side-by-side with old ideas. The kernel-less approach diminishes the need to build a new system from scratch or heavily cannibalize an existing system when experimenting with new systems ideas.

  To achieve performance requires that cross-domain invocations of services can be performed cheaply. Performance is a primary goal of many other operating system research projects which we will examine in the next section.

  In sections 4 and 5 we will present our approach to domain crossing which allows cross-domain invocations to be performed even more efficiently than kernel invocations are today.

2 Related Work

Interest in operating systems research has accelerated in recent years, with large numbers of projects underway[3]. Increasing application requirements for efficient system services are in conflict with increasing demands for more features and backwards compatibility.

One approach to resolving this conflict between functionality and performance. is to allow applications to build custom operating system services. A common strategy is to allow applications to dynamically load systems code into a central kernel. The Cache Kernel[4] defines low-level system objects: address spaces, threads, and kernels. These objects can be loaded into the kernel which will execute the performance critical parts. The SPIN Operating System[5] allows application code written in Modula 3 to be linked into the kernel, using language and loader mechanisms to build software protection domains. A similar approach is taken by Bridge[6] using Software Fault Isolation[7] to allow dynamic linking of application code written in standard languages, like C.

The Flexible Mach[8] effort and Mach development at OSF[9] are co-locating operating system services into applications, at user-level and in the kernel, depending on protection requirements. The Utah effort has also experimented with dynamically attaching trap vectors. Bryce & Muller have reported experiments using the hardware MMU to implement protection domains[10] that are similar to those in SPACE. They have gained performance improvements over Mach 3.0 using an optimization of LRPC[11, 12]. A more limited strategy is adopted by Protected Shared Libraries[13], which uses protected memory regions to build co-located domains for implementing operating system services using segmentation features of the RS/6000 architecture.

An alternative to implementing new system services in the kernel is to streamline the services that are already there through dynamic code synthesis. Synthesis[14] and Synthetix[15] takes this approach, as does Scout[16].

All these approaches still maintain a central kernel (or microkernel). One of the primary considerations in their design is minimizing the crossing of hardware protection boundaries. Cross-domain calls have been a critical issue in the performance of previous systems. However evolving hardware architectures have reduced the cost of cross-domain calls to the same level as system calls to monolithic kernels. Dynamic kernel approaches gain their performance by reducing the number of cross-domain calls at the expense of somewhat lower performance within the kernel, due to the increased abstraction overhead required to support software protection. As cross-domain calls become less expensive, the increased abstraction expense reduces the performance that can be gained. When these techniques are applied to existing applications written to a specific operating system API, the number of cross-domain calls becomes difficult to reduce and the kernel abstraction costs reduce performance. Centralized kernels represent centralized abstractions, which become imposed on all system services.

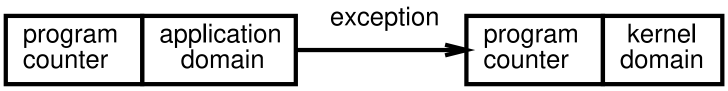
The Exokernel approach is being developed to specifically address the problem of burdening applications with a rigid set of abstractions maintained by the kernel. Originally the Exokernel[17] took a very similar approach to the Cache Kernel, but more recent work indicates that its developers are moving toward the kernel-less model that we have been advocating[1], though with less emphasis on mechanisms realizable in hardware.

One of the key differences between the SPACE approach and that taken by Exokernel and Cache Kernel is the viewpoint we have taken about the necessary abstractions. The Cache Kernel provides abstractions that seem beneficial to operating systems, namely the caching of operating system objects. SPACE abstractions are generalizations of the available hardware trap mechanisms. With continued evolution of processor architectures, our generalized framework is realizable in the hardware itself. The abstractions in the Exokernel and Cache Kernel will likely continue to require software or microcode implementation.

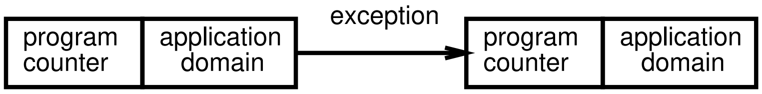
3 Building Kernel-less O/S Abstractions

In this section we describe how some of the common operating system abstractions, typically found in kernels, can be implemented as applications using generalized hardware trap handling.

Invocation of kernel services requires that a program generate an exception, e.g. executing a *trap instruction*. The hardware trap architecture changes the processor mode to *supervisor*, saves minimal state and performs an indirect branch through the trap vector table.



To implement a cross-domain mechanism requires a slight generalization of this hardware mechanism:



Instead of handling the exception in the kernel domain, a handler is invoked in another application domain. For architectures that support tagged TLBs, this difference is whether the supervisor mode bit is set in the processor state register or a new context id is inserted into the MMU context register.

We refer to the mapping from an exception in one domain to a handler in another as a *portal*. A cross-domain call is then defined as a *portal traversal*.

In later sections we will examine the internal details of portals and cross-domain calls, particularly addressing performance and implementation issues. There we will address issues of how and when state is saved, how hardware exceptions are demultiplexed, and how the trap vector table is generalized. For now we assume that the internal details of cross-domain calls are represented by defining the portal types: *simple-trap*, *resume*, *suspend*, and *unsuspend*. These portal types use a common mechanism for recording the history of portal traversals. The state saved at each portal traversal is recorded in a *last-in first-out* queue, which we call a *CHAIN*.

- **simple-trap** The previous *program counter* (PC), *MMU context id* (CTXID) and hardware *trap vector address* (TVEC) are pushed onto the current CHAIN, and the hardware is updated with a new PC, CTXID, and TVEC. The general purpose register state is not saved in the portal implementation, but in applications themselves according to the *caller-save/callee-save* procedure call conventions used by the programming language. Similarly parameters are passed by-value in the registers.

- **resume** The PC, CTXID, and TVEC at the head of the CHAIN are restored. Depending on the exception the last instruction is re-executed or skipped.
- **suspend** Like a simple-trap, except that the processor registers are saved in the current CHAIN which is itself then saved in a table and an empty CHAIN is established. A token is passed to the handler which can be used to retrieve the saved CHAIN.
- **unsuspend** The current CHAIN is discarded and replaced by the saved CHAIN identified by a token passed as an argument. The previous state is restored.

We have implemented these portal types on a 40MHz SuperScalar SPARC, and measured the performance. To pass through a portal of type *simple* and then back through again (*simple-resume*) to the original domain takes 1.6μS, over half of which is due to hardware trap overhead. To pass through a series of *simple* portals and then returning takes 2.5μS per portal. The suspend/unsuspend pair of portal types require 4.9μS to pass through and immediately return. In comparison Solaris 2.3 running on the same hardware requires 9.0μS to perform a null-system call (*getpid*). These measurements suggest that the performance of a kernel-less operating system is able to exceed the performance of a kernel-based system on the same hardware. In a later section we will examine the reasons for higher costs in kernel-based systems.

3.1 A Simple Kernel-Less Thread Mechanism

Thread mechanisms are built from the simple cross-domain invocation using the *suspend* portal type, which allows a domain to be entered from one chain of cross-domain calls, yet resume a different one. Each of the chains represents a history of cross-domain calls. *suspend* portals save the current chain and *start a new one*. The new chain may be short-lived, or it may represent the beginning of a new application thread. A token is passed to the handler for the *suspend* portal. This token identifies a particular chain, and can later be passed to an *unsuspend* portal which will abandon the current chain and resume the suspended one. Each chain is marked with the identity of the domain that was issued the token, and only that domain can unsuspend the chain.

Using the suspend portal type we can build thread operations which execute in an application domain. For instance consider the handlers for thread yield and create:

```
thread_yield(token)
    ENQUEUE(token)
    INVOKE T_UNSUSPEND(DEQUEUE())

thread_create(exception_type,token)
    ENQUEUE(token)
    INVOKE(exception_type)
    INVOKE T_UNSUSPEND(DEQUEUE())
```

The *ENQUEUE/DEQUEUE* operations manipulate a circular list used to remember tokens that are not currently running. The *INVOKE* operation causes the specified exception. *T_UNSUSPEND* identifies an exception (likely a user trap instruction) that is mapped into an unsuspend portal.

The yield operation switches from one thread to another in round-robin fashion. The create operation launches a new thread by causing an exception which maps into a handler that executes the thread.

The suspend portal that invokes *thread_yield* saves the head of the current chain, and passes *thread_yield* a token to it. The minimal thread implementation shown here only needs to remember the passed token and unsuspend a different one. The chain created by the *suspend* portal is short-lived.

thread_create is like *thread_yield* except that instead of immediately unsuspending a token, it uses the chain created by the suspend portal to run code belonging to the calling application. The application code is invoked by causing an exception that runs the application as its handler. The handler runs as the beginning of a new chain. Eventually the handler exits by traversing a *resume* portal, and *thread_create* is resumed. *thread_create* then traverses an *unsuspend* portal, which discards the chain that ran the handler.

Nothing in this thread model is concerned with thread stacks. The threads built using the suspend/unsuspend portaltypes can traverse protection domains. For reasons of robustness and integrity, some exception handlers will not trust the stack in use by the caller and wish to allocate their own. Thus a thread may not have a single stack. Some handlers may be implemented without stacks for efficiency. Stacks are useful in procedure-oriented languages because they provide a simple method of automatic storage allocation that supports recursion. In many cases, particularly where there is no recursion, the total amount of automatic storage required can be statically computed and simpler allocation strategies can give better performance.

A more general thread mechanism than shown here might extend *thread_yield* to be *thread_suspend* and *thread_signal* operations. *thread_suspend* is like *thread_yield* except that the the suspended thread is not runnable again until *thread_signal* is called by another thread to signal that some condition has become true. Other added functionality might replace the ENQUEUE/DEQUEUE operations by some form of prioritized scheduling. The *thread_create* operation could take parameters to pass to the application. Thread descriptors might be built to support the naming of threads, collecting statistics, management of thread stacks, and controlling scheduling and other behaviours.

What is significant about SPACE is that these heavier-weight mechanisms can coexist with the extremely lightweight mechanisms that were presented here[2]. This lightweight functionality will not suit every application, but *any* application with special requirements can extend threads without imposing the new implementation on *every* application.

Each create/exit of a lightweight SPACE thread takes $7.3\mu S$ on our SPARC platform. We measured trivial thread create/exit operations under Solaris.

```
null_thread() {}

...
thr_create(stack, sizeof(stack), null_thread, 0, 0, 0)
...
```

Each create/exit of a thread takes $512\mu S$ in Solaris, 50 times slower than in the kernel-less implementation, and the Solaris threads execute within only a single protection domain, where the SPACE threads can cross domains.

3.2 Kernel-less Virtual Memory Abstractions

Most processor architectures support virtual memory using in-memory data structures to represent the mappings between *virtual addresses* and *physical addresses*. For performance reasons these mappings are cached in a *memory management unit* (MMU), using *translation look-aside buffers* (TLBs). To avoid flushing the MMU when switching between address spaces, (e.g. processes) TLB entries are often tagged with an MMU context identifier (CTXID).

There is great variation between processors in the implementation details of these structures, particularly in the memory representations of page maps, the management of TLBs, and protection. Because these structures are primarily designed to implement protection domains for processes and to support virtual memory, the details are typically encapsulated in several layers of kernel code which adapt the hardware abstractions to the desired operating system abstractions[18].

Performance of virtual memory can be highly dependent on page replacement policies. Some virtual memory abstractions give applications hooks to control or influence these policies[19]. These extensions can be used to build new types of virtual memory at the application level, but only allow coarse grain access to the hardware paging mechanisms.

[2] An even lighter-weight implementation might build *thread_create* and *thread_yield* as portal types. However in the current system, implementation of portal types is restricted to trusted applications. A second optimization, which is more universal, is to require applications that invoke the lightweight thread services to first save all *live* registers

However virtual memory is not the only interesting use of processor memory management hardware. Appel & Li[20] have examined a number of other uses for virtual memory primitives, including concurrent garbage collection and persistent stores. To effectively use the MMU in these applications requires efficient access to these primitives without incurring the overhead of the operating system abstractions.

We measured an operation, suggested by the Appel & Li paper, in which an application causes a page protection fault and invokes an application-level handler which fixes the protection on the faulting page, and then protects a second page and returns. Under Solaris this is implemented with *sigaction* and *mprotect*. The time from the fault to the successful execution of the memory reference took 430μS under Solaris, and only 20μS in our kernel-less implementation.

The application-level handler which implements the Appel & Li example can instead implement general purpose pagefault handling as part of a kernel-less implementation of virtual memory. Different address spaces (or even regions of address spaces) need not necessarily invoke the same pagefault handler, allowing multiple virtual memory implementations to coexist.

3.3 Kernel-less Operating System Services

Kernel-less operating system services have been successfully built using the microkernel/server approach, as in Mach 3.0. The goals of robustness and extensibility have been achieved for the services that were removed from the kernel. However microkernels still contain significant abstractions for threads, virtual memory, and inter-process communication. The result is that performance of Mach servers is much worse than for systems based on full-kernels. After significant efforts to improve performance, Mach researchers have retreated from the microkernel/server approach and are now dynamically loading server code directly into application address spaces (*co-location*).

There are two problems that co-location solves. The first is the cost of the Mach abstractions for communication. Even after extensive tuning, remote procedure calls (RPC) are still four times slower[9] than in systems like Spring[21]. Dynamic loading of services allows RPC to be avoided, as in Bridge/SFI[7] and systems like SPIN[5].

The second problem is that services implemented in distinct address space require parameters to be copied between address spaces. Services in the application address space can directly access parameters (given sufficient protection mechanisms).

Co-location is one of the advantages that kernel-based systems inherently have. Hardware memory management supports two protection domains within every address space: *user* and *supervisor*. Code executing in the supervisor domain can typically access all data in the user domain.

We have proposed an alternative implementation of co-location[1], which uses overlapping hardware address spaces to build an arbitrary number of domains within each address space. Services in co-located domains are accessed via cross-domain calls through portals. Services can directly reference any page within the common address space for which they have been granted permission.

On architectures with tagged TLBs and hierarchical page table structures, the cost of this implementation is the additional page tables that must be added to describe the distinct domains, as we show in[2], and the corresponding increase in the size of the TLB working set within the MMU.

In our kernel-less implementation, invocation of a null operating system service (co-located or not) takes only the cost of the cross-domain call (1.6μS on our SPARC platform). A *getpid()* call to a kernel-based system, Solaris, takes 9.0μS on the same platform, yet this is on hardware designed to support the kernel-based approach.

The overhead in Solaris is due, indirectly, to its complexity. There is a very large amount of code in the kernel, and to make its management tractable, various abstractions are maintained: *proc* table entries, the *user* structure, run-time environment (register windows, the kernel stack), and LWP (light-weight processes). Requesting a kernel service normally requires passing through an *abstraction-adaptation* layer (i.e. *locore.s*), even if few of the abstractions are important for a particular service.

Because there is so much sharing of abstractions, kernel-based systems achieve robustness by insuring that the abstraction assumptions are maintained. If an inconsistency is detected at a time it is unexpected, the kernel has difficulty recovering, and the system panics. Robustness in kernel-less operating systems can instead be achieved by executing groups of services in protected domains — each with appropriate restart mechanisms to recover from internal errors.

There have been several attacks on the overhead problem in the context of kernel-based systems. One has been to write special case code in locore.s for services whose performance is too critical to pass through the abstraction-adaptation layer. Another has been to dynamically synthesize kernel-code at run-time[14].

A kernel-less system does not eliminate the abstraction-adaptation layer. To do so would make it infeasible to construct service-rich operating systems. Where a kernel-less operating system gains in performance is by providing extensibility that allows customized versions of operating system services to be built that can take advantage of the specific circumstances of a particular application.

Kernel-less systems are likely to still have large monolithic implementations of operating system services. One of the costs associated with the decomposition of kernel services in microkernels is that the number of cross-domain calls can increase because services access other services with cross-domain calls instead of procedure calls. Even with no sofware overhead, the hardware trap architecture on current processors imposes a not insignificant cost. Though user applications can invoke services as cheaply in kernel-less systems as with kernel-based systems, monolithic implementations have the advantage that internally services can access each other with procedure calls.

The cross-domain calls in SPACE are modeled on traditional procedure calls. These calls can be replaced by procedure calls simply by merging co-located protection domains. A collection of interdependent system services can be implemented using domains during development, and then merged into a monolithic structure during final test and deployment in order to improve performance, without changing any of the external interfaces used by clients of the services.

4 SPACE

A commonly held view, that has driven recent operating systems research, is that crossing domain boundaries is too expensive, and thus other ways of solving the functionality and performance problems must be found. One direction that researchers have looked is embedding application code within the kernel domain, using language and linking techniques to build protection domains in software[5, 6]. So far these approaches have had some success, though at the cost of higher run-time overhead, as well as increasing the cost of the the cross-domain calls that the techniques do not eliminate.

These techniques are applicable to kernel-less systems just as much as to kernel-based systems. Instead of dynamically installing application code in the kernel, they can be used to install application code into servers.

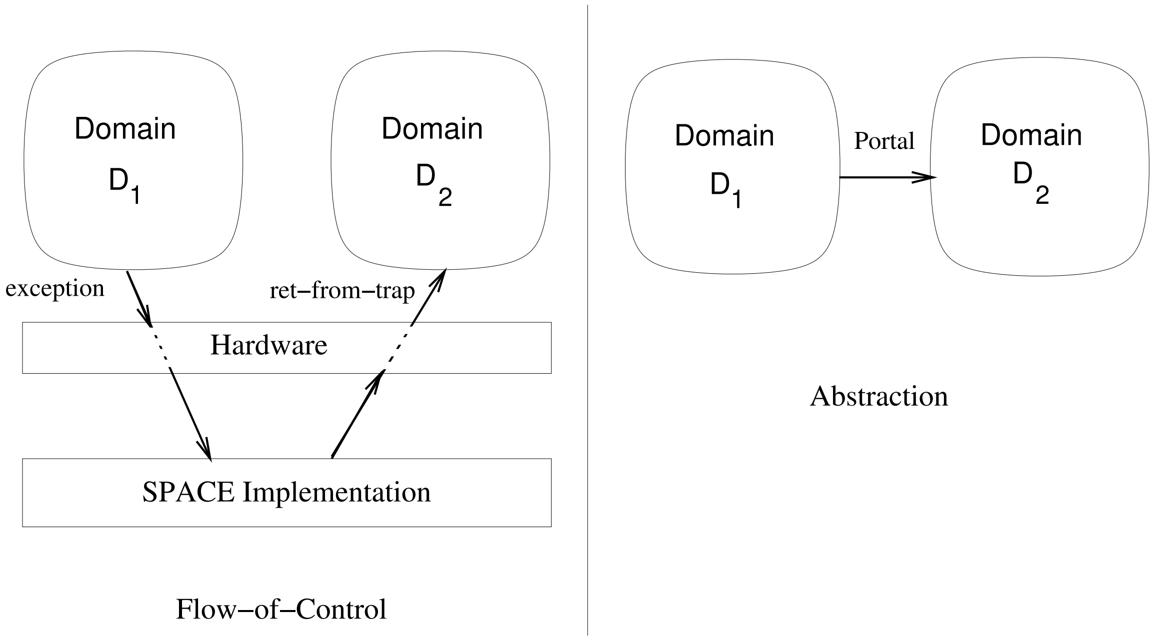
Two factors are key for deciding when software protection domains are valuable. The first is how greatly the number of cross-domain calls can be reduced by the scheme. If taken to the extreme, the software protection approach would run all applications in a single hardware protection domain. Systems like the B6700 directly supported this approach in the context of a segmented architecture. However without segmentation, single-address space systems have problems with program linking, address-space allocation, and limitations on the manipulation of the address space (e.g. unix fork is impossible)[22]. Short of building single-address space systems, hardware domain crossings will continue to occur. Moving application functionality into the kernel can reduce the frequency of cross-domain calls, but will not eliminate them — particularly for existing applications which use a standard API.

The second factor is the cost of hardware domain crossings. If cross-domain calls cost on the order of a procedure call, software protection domains would be unnecessary. The reduction of these costs is not only important for crossing between domains, but also for handling exceptions. Exceptions incur very similar costs to cross-domain calls.

Where software protection domains attempt to minimize the frequency of crossing hardware domains, the kernel-less approach assumes that the costs of crossing hardware domains can be dramatically reduced. In this section we present the SPACE approach which we used to make the measurements in section 3.

4.1 The SPACE mechanisms

The figure below depicts the Flow-of-Control in a cross-domain control transfer initiated by an exception which takes the control flow from domain D_1 into the SPACE Implementation. The SPACE Implementation determines the domain and the address of the handler based on the domain D_1 and the exception. The cross-domain control transfer is complete when the SPACE Implementation executes a hardware *return-from-trap* instruction to being executing in the domain D_2.



The abstraction of this control transfer is called a *portal*. Different portals can be defined for each domain by parameterizing a set of *portal types*. Portal types determine the details of a specified cross-domain control transfer, including how state is saved/restored and what parameters are passed.

4.2 Details of Cross-Domain Calls

To move from one domain to another requires that some state be preserved (analogous to saving the PC on a procedure call stack). Minimally SPACE needs to preserve the PC, the MMU context-id, and trap table pointer. SPACE also records the continuation address within the SPACE Implementation. This is the address of the code which will restore execution in the original domain when the invoked domain returns.

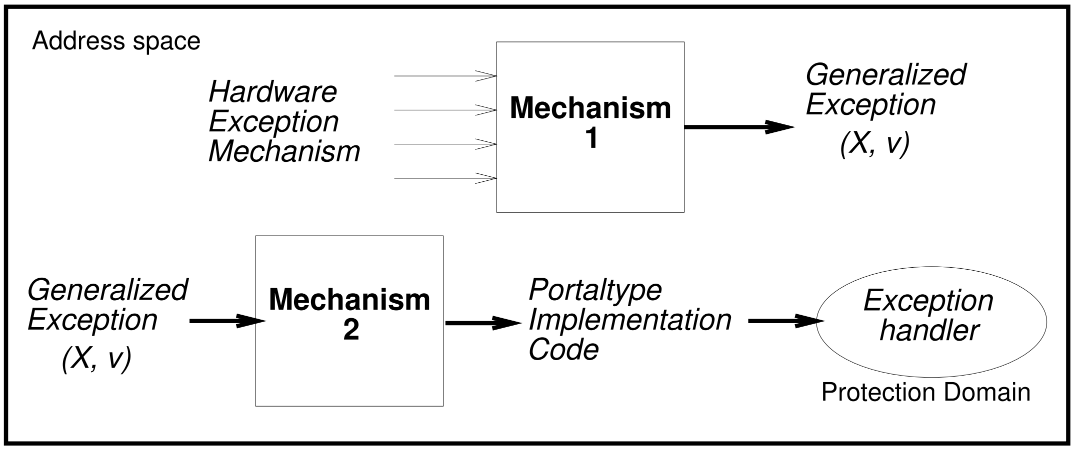
If trap registers are available, this minimal state can be saved without any memory references. But if the new domain performs yet-another cross-domain call, the available trap registers will not be available and a data structure (PCB) must be allocated to hold the processor state.

A sequence of cross-domain calls and returns will result in a chain of PCBs topped by state stored in the trap registers.

Cross-domain calls are implemented as portals which instantiate a particular portal type. The portal types are implemented as parameterized code templates and assembly code sequences. These templates are used to synthesize code when a portal is initialized. This synthesis allows parameters (such as MMU context id and location of the domain's trap table) to be bound when the domain is created. Use of static code and parameter lookup would add memory references, slowing execution.

5 The SPACE Model

SPACE generalizes the processor architecture by abstracting the hardware notions of *address spaces*, *privilege modes*, and *exceptions*. These mechanisms are used to construct software using *spaces*, *domains*, and *portals*. We describe the nature of the underlying mechanisms used by SPACE, and explain how these mechanisms provide fundamental extensibility. In a subsequent section we suggest how these mechanisms can be directly implemented in hardware.



Generalized Exception and Portal Type Mechanisms

5.1 Generalized Exceptions

Exceptions are extended in SPACE through the mechanisms shown in figure 2.

The first mechanism translates a hardware exception and related information (e.g. processor exception registers) into a generalized exception and parameter pair *(X,v)*. The parameter *v* normally identifies the particular instance of the *generalized exception X* (e.g. trap number). However for exceptions generated by memory references, *v* is the virtual address that caused the hardware exception.

This generalized exception mechanism decouples the hardware view of exceptions from the requirements of the application. Hardware exceptions which satisfy application requirements are mapped into generalized exceptions without overhead. Examples of exceptions which may be directly mapped include trap-instructions and arithmetic exceptions.

In other cases a hardware exception may not be suitable for an application. Perhaps the application was originally designed for another architecture. Perhaps the hardware exception needs to be demultiplexed into several cases that are handled differently. A generalized exception can be synthesized from a hardware exception and auxiliary processor state (e.g. special registers and access bits in page table entries).

5.2 Portals

The second mechanism maps generalized exceptions into portals. A *portal* associates an *(X,v)* pair with a *generalized-exception handler*. It specifies a new value for the program counter, as well as a new address space and privilege mode in which to execute. For memory exceptions, where *v* is a virtual address, different handlers can be associated with different parts of an address space. This supports the coexistence of distinct virtual memory implementations within an application. Portals are generalizations of the trap vector mechanism used in most processors.

5.3 Portal Types

Portals also specify the *portal type* implementation to use for traversing the portal to the exception handler and later returning. The portal type determines the conventions for saving state, synthesizing parameters, and resuming the interrupted execution when the handler returns from processing the exception.

Portal type implementations do not necessarily involve cross-domain calls. Some portal types may execute to completion without ever entering a new protection domain, *short-circuiting* the normal portal mechanism. Such portal types are useful for extension of the processor instruction set, as well as implementation of short operations critical to application performance. Short-circuit portal types are also used to implement specialized operations, such as *return-from-exception*.

5.4 Portal-based Capabilities

In kernel-based systems, applications access system services through a rigid system interface. Protected services are implemented by maintaining internal state about each client which is used to gate access to services.

In SPACE each domain can have a customized system interface, built using per-domain copies of the hardware trap vector. Domains can only access portals that are connected to their trap vector, so the trap vectors implement a capability mechanism. Higher-level protection mechanisms are used when modifying a domain's trap vector to validate access to services. Once an exception-to-portal mapping is established, permissions do not need to be verified when the portal is traversed.

Specialized trap vectors are also useful for implementing dynamically customized services. For example a service that opens a file might establish read/write operations tailored to that file or class of files, as in Synthesis[14].

6 Hardware trap mechanisms for SPACE

The low overhead of cross-domain calls in SPACE exposes the underlying cost of crossing hardware domain boundaries. In principle these costs should be low on RISC processors[3]. However designers have had little incentive to trim these costs. The controllers for modern microprocessors are immensely complex, particularly in superscalar processors like the *SPARC* and *UltraSPARC*. Keeping exceptions generated in one protection domain from interrupting the processor after it switches to another increases the complexity of the controllers. Typically operating system overhead completely masks the hardware costs, but not in kernel-less systems using SPACE. The lower the hardware costs, the more useful protection domains become for decomposing services and exposing interfaces.

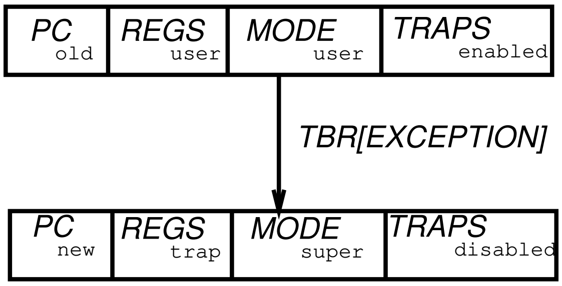
On the SuperSPARC processor we used for our experiments, the minimal cost of a round-trip through the trap mechanism (i.e. the trapvector contains a *rett/jmp* pair) is 14 clocks. The cost is much higher than the costs of the jmp/returns involved (6 clocks). A fast cross-domain call in SPACE requires 64 clocks. 28 of these are spent in the hardware trap mechanism. The 16 clocks over the cost of the control transfers represent an overhead of 25%.

6.1 Hardware support for cross-domain calls

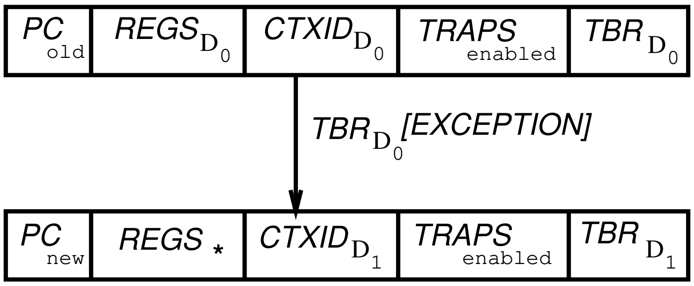
Hardware trap overhead is the critical cost when implementing high-speed cross-domain calls, accounting for 56% of the costs of fast SPACE cross-domain calls on the SPARC.

The SPARC hardware trap mechanism performs an indexed branch into the trap vector, determined by the *trap base register* (TBR). The trap mechanism also switches to the trap register set, saves the program counters, and changes the processor mode to supervisor (with traps disabled):

[3] Some architectures (e.g. PA/Risc[23]) already provide lower-overhead paths through the hardware trap architecture for invoking services.



The generalized mechanisms in SPACE implement the following transition (from domain D_0 to D_1):

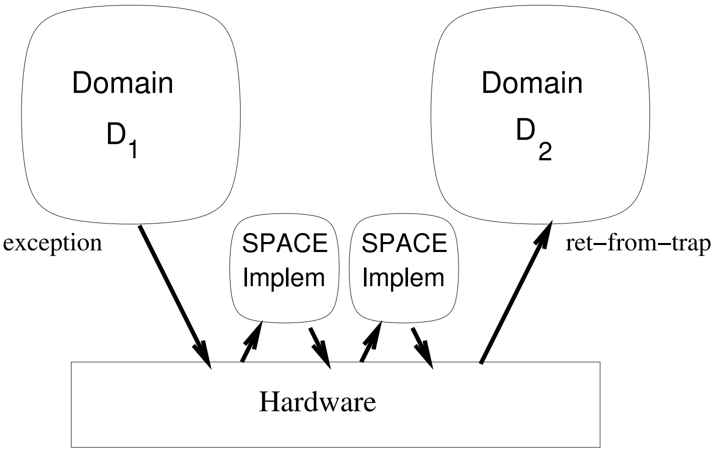


The differences from the hardware implementation are the following:

- The registers are saved/modified in an exception-dependent way. In the simplest case they are not modified at all, and state saving is left to the application handler.
- Instead of changing between hardware modes, the MMU context id is changed changing between hardware protection domains.
- The set of mappings are themselves changed by substituting a new value in the trap base register.

To implement the SPACE mechanisms in hardware requires that the MMU CTXID and new TBR value be available in the trap table as well as the new program counter. However the SPACE mechanisms are actually much more general than can be implemented by hardware. Hardware exceptions may need to be demultiplexed (synthesized exceptions). The transition between domains may depend on more than just the exception type (e.g. the faulting address). There may be specific requirements for saving register state generating parameters.

If the cost of the hardware mechanism is sufficiently reduced, it becomes feasible to eliminate the abstraction adaptation layer in the current version of SPACE in favor of one that invokes application level handlers to provide for more complex domain transitions:



The small boxes shown in the figure implement the parts of portals not supported directly by hardware, such as register saving. These code fragments are synthesized, as in the current SPACE system, but run as applications rather than in hardware control mode. With such a system the need for control mode could be effectively eliminated, and instead replaced by protections enforced by the MMU.

7 Conclusions

We have applied the SPACE mechanisms to building low-level system services. These service abstractions are usually considered fundamental, requiring implementation within kernels. Yet we have implemented them as applications and achieved performance that is an order-of-magnitude better than a conventional kernel-based implementation.

The advantages of kernel-less operating systems are suggested by microkernel/server operating systems, but these systems have been unable to overcome the penalties of the abstractions contained in the microkernel. In a kernel-less operating system, even the most fundamental mechanisms can be replaced on a per-application basis.

The critical requirement is that cross-domain calls be made very inexpensive. Otherwise approaches, such as software protection domains, are required which minimize the frequency of cross-domain calls.

The SPACE mechanisms exceed the performance of kernel-based mechanisms because modularity and the use of capabilities for security eliminate the need to support the abstraction adaptation layer found in monolithic kernels.

The cost difference between procedure calls and cross-domain calls is significant in current processors. This gives performance advantages to aggregating highly interdependent system services. In SPACE we advocate constructing such aggregates by merging together co-located domains and replacing cross-domain calls with procedure calls. During testing and debugging the cross-domain calls have the advantage of fault containment, that may be less critical when the system is deployed.

We expect that over time, software trap models from systems like SPACE will push hardware trap costs lower, particularly for the synchronous exceptions commonly used to invoke system services. Ultimately the hardware privilege mode mechanisms will be replaced by a protection model that relies on capabilities granted by the MMU.

Bibliography

[1] Dave Probert, John Bruno, and Murat Karaorman. Space: A new approach to operating system abstraction. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 133–137, October 1991.

[2] Dave Probert and John Bruno. Building fundamentally extensible application-specific operating systems in space. Technical Report TRCS95-06, Computer Science Dept., UC Santa Barbara, March 1995.

[3] Patrick Bridges. Current operating systems projects and os-related research. http://www.cs.arizona.edu/people/bridges/oses.html.

[4] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 179–193, November 1994.

[5] Brian Bershad, Stefan Savage, and et. al. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[6] Steve Lucco. Adaptable binary programs. Technical Report CMU-CS-94-137, CMU, 1994.

[7] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, July 1993.

[8] In-Kernel Servers on Mach 3.0: Implementation and Performance. Jay lepreau and mike hibler and bryan ford and jeff law. In *Proc. of the 3rd Usenix Mach Symposium*, Santa Fe, NM, April 1993.

[9] Michael Condict, David Mitchell, and Franklin Reynolds. Optimizing performance of mach-based systems by server co-location: A detailed design, August 1993.

[10] G. Muller C. Bryce. Matching micro-kernels to modern applications using fine-grained memory protection. In *Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 272–279, San Antonio (Tx), October 1995.

[11] B. N. Bershad, T. E. Anderson, E D. Lazowska, and H.E. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[12] Virgil Bourassa and John Zahorjan. Implementing lightweight remote procedure calls in the mach 3.0 operating system. Technical Report UW-CSE-95-02-01, Dept. of Comp. Sci. and Eng., University of Washington, Seattle, 1995.

[13] Arindam Banerji and David L. Cohn. Protected shared libraries. To appear in IWOOOS, December 1994.

[14] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[15] T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 314–324, December 1995.

[16] A. B. Montz and et. al. Scout: A communications-oriented operating system. Technical Report 94-20, Dept. of Computer Science, Univ. of Arizona, June 1994.

[17] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[18] Berny Goodheart and James Cox. *The Magic Garden Explained*. Prentice Hall, 1994.

[19] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computer*, 37(8):896–908, August 1988.

[20] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *ASPLOS IV*, pages 96–107, April 1991.

[21] J. Mitchell and et. al. An overview of the spring system. In *Proceedings of Compcon Spring 1994*, February 1994.

[22] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *OOPSLA*, October 1992.

[23] Hewlett Packard. *PA-RISC 1.3 Architecture and Instruction Set Reference Manual, Fourth Edition*. Hewlett Packard, 1995.