A version of this paper appeared in the Proceedings of the Fifteenth Symposium on Operating Systems Principles

Extensibility, Safety and Performance in the *SPIN* Operating System

Brian N. Bershad   Stefan Savage   Przemysław Pardyak   Emin Gün Sirer
Marc E. Fiuczynski   David Becker   Craig Chambers   Susan Eggers

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

This paper describes the motivation, architecture and performance of *SPIN*, an extensible operating system. *SPIN* provides an extension infrastructure, together with a core set of extensible services, that allow applications to safely change the operating system's interface and implementation. Extensions allow an application to specialize the underlying operating system in order to achieve a particular level of performance and functionality. *SPIN* uses language and link-time mechanisms to inexpensively export fine-grained interfaces to operating system services. Extensions are written in a type safe language, and are dynamically linked into the operating system kernel. This approach offers extensions rapid access to system services, while protecting the operating system code executing within the kernel address space. *SPIN* and its extensions are written in Modula-3 and run on DEC Alpha workstations.

1 Introduction

SPIN is an operating system that can be dynamically specialized to safely meet the performance and functionality requirements of applications. *SPIN* is motivated by the need to support applications that present demands poorly matched by an operating system's implementation or interface. A poorly matched implementation prevents an application from working well, while a poorly matched interface prevents it from working at all. For example, the implementations of disk buffering and paging algorithms found in modern operating systems can be inappropriate for database applications, resulting in poor performance [Stonebraker 81]. General purpose network protocol implementations are frequently inadequate for supporting the demands of high performance parallel applications [von Eicken et al. 92]. Other applications, such as multimedia clients and servers, and realtime and fault tolerant programs, can also present demands that poorly match operating system services. Using *SPIN*, an application can extend the operating system's interfaces and implementations to provide a better match between the needs of the application and the performance and functional characteristics of the system.

1.1 Goals and approach

The goal of our research is to build a general purpose operating system that provides extensibility, safety and good performance. Extensibility is determined by the interfaces to services and resources that are exported to applications; it depends on an infrastructure that allows fine-grained access to system services. Safety determines the exposure of applications to the actions of others, and requires that access be controlled at the same granularity at which extensions are defined. Finally, good performance requires low overhead communication between an extension and the system.

The design of *SPIN* reflects our view that an operating system can be extensible, safe, and fast through the use of language and runtime services that provide low-cost, fine-grained, protected access to operating system resources. Specifically, the *SPIN* operating system relies on four techniques implemented at the level of the language or its runtime:

- *Co-location.* Operating system extensions are dynamically linked into the kernel virtual address space. Co-location enables communication between system and extension code to have low cost.

This research was sponsored by the Advanced Research Projects Agency, the National Science Foundation (Grants no. CDA-9123308 and CCR-9200832) and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship. Chambers was partially sponsored by a National Science Foundation Presidential Young Investigator Award. Sirer was supported by an IBM Graduate Student Fellowship. Fiuczynski was partially supported by a National Science Foundation GEE Fellowship.

- *Enforced modularity.* Extensions are written in Modula-3 [Nelson 91], a modular programming language for which the compiler enforces interface boundaries between modules. Extensions, which execute in the kernel's virtual address space, cannot access memory or execute privileged instructions unless they have been given explicit access through an interface. Modularity enforced by the compiler enables modules to be isolated from one another with low cost.

- *Logical protection domains.* Extensions exist within logical protection domains, which are kernel namespaces that contain code and exported interfaces. Interfaces, which are language-level units, represent views on system resources that are protected by the operating system. An in-kernel dynamic linker resolves code in separate logical protection domains at runtime, enabling cross-domain communication to occur with the overhead of a procedure call.

- *Dynamic call binding.* Extensions execute in response to system events. An event can describe any potential action in the system, such as a virtual memory page fault or the scheduling of a thread. Events are declared within interfaces, and can be dispatched with the overhead of a procedure call.

Co-location, enforced modularity, logical protection domains, and dynamic call binding enable interfaces to be defined and safely accessed with low overhead. However, these techniques do not guarantee the system's extensibility. Ultimately, extensibility is achieved through the system service interfaces themselves, which define the set of resources and operations that are exported to applications. *SPIN* provides a set of interfaces to core system services, such as memory management and scheduling, that rely on co-location to efficiently export fine-grained operations, enforced modularity and logical protection domains to manage protection, and dynamic call binding to define relationships between system components and extensions at runtime.

1.2 System overview

The *SPIN* operating system consists of a set of extension services and core system services that execute within the kernel's virtual address space. Extensions can be loaded into the kernel at any time. Once loaded, they integrate themselves into the existing infrastructure and provide system services specific to the applications that require them. *SPIN* is primarily written in Modula-3, which allows extensions to directly use system interfaces without requiring runtime conversion when communicating with other system code.

Although *SPIN* relies on language features to ensure safety *within* the kernel, applications can be written in any language and execute within their own virtual address space. Only code that requires low-latency access to system services is written in the system's safe extension language. For example, we have used *SPIN* to implement a UNIX operating system server. The bulk of the server is written in C, and executes within its own address space (as do applications). The server consists of a large body of code that implements the DEC OSF/1 system call interface, and a small number of *SPIN* extensions that provide the thread, virtual memory, and device interfaces required by the server.

We have also used extensions to specialize *SPIN* to the needs of individual application programs. For example, we have built a client/server video system that requires few control and data transfers as images move from the server's disk to the client's screen. Using *SPIN* the server defines an extension that implements a direct stream between the disk and the network. The client viewer application installs an extension into the kernel that decompresses incoming network video packets and displays them to the video frame buffer.

1.3 The rest of this paper

The rest of this paper describes the motivation, design, and performance of *SPIN*. In the next section we motivate the need for extensible operating systems and discuss related work. In Section 3 we describe the system's architecture in terms of its protection and extension facilities. In Section 4 we describe the core services provided by the system. In Section 5 we discuss the system's performance and compare it against that of several other operating systems. In Section 6 we discuss our experiences writing an operating system in Modula-3. Finally, in Section 7 we present our conclusions.

2 Motivation

Most operating systems are forced to balance generality and specialization. A general system runs many programs, but may run few well. In contrast, a specialized system may run few programs, but runs them all well. In practice, most general systems can, with some effort, be specialized to address the performance and functional requirements of a particular application's needs, such as interprocess communication, synchronization, thread management, networking, virtual memory and cache management [Draves et al. 91, Bershad et al. 92b, Stodolsky et al. 93, Bershad 93, Yuhara et al. 94, Maeda & Bershad 93, Felten 92, Young et al. 87, Harty & Cheriton 91, McNamee & Armstrong 90, Anderson et al. 92, Fall & Pasquale 94, Wheeler & Bershad 92, Romer et al. 94, Romer et al. 95, Cao et al. 94]. Unfortunately, existing system structures are not well-suited for specialization, often requiring a substantial programming effort to affect even a small change in system behavior. Moreover, changes intended to improve the performance of one class of applications can often degrade that of others. As a result, system specialization is a costly and error-prone process.

An extensible system is one that can be changed dynamically to meet the needs of an application. The need for extensibility in operating systems is shown clearly by systems such as MS-DOS, Windows, or the Macintosh Operating System. Although these systems were not designed to be extensible, their weak protection mechanisms have allowed application programmers to directly modify operating system data structures and code [Schulman et al. 92]. While individual applications have benefited from this level of freedom, the lack of safe interfaces to either operating system services or operating system extension services has created system configuration "chaos" [Draves 93].

2.1 Related work

Previous efforts to build extensible systems have demonstrated the three-way tension between extensibility, safety and performance. For example, Hydra [Wulf et al. 81] defined an infrastructure that allowed applications to manage resources through multi-level policies. The kernel defined the mechanism for allocating resources between processes, and the processes themselves implemented the policies for managing those resources. Hydra's architecture, although highly influential, had high overhead due to its weighty capability-based protection mechanism. Consequently, the system was designed with "large objects" as the basic building blocks, requiring a large programming effort to affect even a small extension.

Researchers have recently investigated the use of microkernels as a vehicle for building extensible systems [Black et al. 92, Mullender et al. 90, Cheriton & Zwaenepoel 83, Cheriton & Duda 94, Thacker et al. 88]. A microkernel typically exports a small number of abstractions that include threads, address spaces, and communication channels. These abstractions can be combined to support more conventional operating system services implemented as user-level programs. Application-specific extensions in a microkernel occur at or above the level of the kernel's interfaces. Unfortunately, applications often require substantial changes to a microkernel's implementation to compensate for limitations in interfaces [Lee et al. 94, Davis et al. 93, Waldspurger & Weihl 94].

Although a microkernel's communication facilities provide the infrastructure for extending nearly any kernel service [Barrera 91, Abrossimov et al. 89, Forin et al. 91], few have been so extended. We believe this is because of high communication overhead [Bershad et al. 90, Draves et al. 91, Chen & Bershad 93], which limits extensions mostly to coarse-grained services [Golub et al. 90, Stevenson & Julin 95, Bricker et al. 91]. Otherwise, protected interaction between system components, which occurs frequently in a system with fine-grained extensions, can be a limiting performance factor.

Although the performance of cross-domain communication has improved substantially in recent years [Hamilton & Kougiouris 93, Hildebrand 92, Engler et al. 95], it still does not approach that of a procedure call, encouraging the construction of monolithic, non-extensible systems. For example, the L3 microkernel, even with its aggressive design, has a protected procedure call implementation with overhead of nearly 100 procedure call times [Liedtke 92, Liedtke 93, Int 90]. As a point of comparison, the Intel 432 [Int 81], which provided hardware support for protected cross-domain transfer, had a cross-domain communication overhead on the order of about 10 procedure call times [Colwell 85], and was generally considered unacceptable.

Some systems rely on "little languages" to safely extend the operating system interface through the use of interpreted code that runs in the kernel [Lee et al. 94, Mogul et al. 87, Yuhara et al. 94]. These systems suffer from three problems. First, the languages, being little, make the expression of arbitrary control and data structures cumbersome, and therefore limit the range of possible extensions. Second, the interface between the language's programming environment and the rest of the system is generally narrow, making system integration difficult. Finally, interpretation overhead can limit performance.

Many systems provide interfaces that enable arbitrary code to be installed into the kernel at runtime [Heidemann & Popek 94, Rozier et al. 88]. In these systems the right to define extensions is restricted because any extension can bring down the entire system; application-specific extensibility is not possible.

Several projects [Lucco 94, Engler et al. 95, Small & Seltzer 94] are exploring the use of *software fault isolation* [Wahbe et al. 93] to safely link application code, written in any language, into the kernel's virtual address space. Software fault isolation relies on a binary rewriting tool that inserts explicit checks on memory references and branch instructions. These checks allow the system to define protected memory segments without relying on virtual memory hardware. Software fault isolation shows promise as a co-location mechanism for relatively isolated code and data segments. It is unclear, though, if the mechanism is appropriate for a system with fine-grained sharing, where extensions may access a large number of segments. In addition, software fault isolation is only a protection mechanism and does not define an extension model or the service interfaces that determine the degree to which a system can be extended.

Aegis [Engler et al. 95] is an operating system that relies on efficient trap redirection to export hardware services, such as exception handling and TLB management, directly to applications. The system itself defines no abstractions beyond those minimally provided by the hardware [Engler & Kaashoek 95]. Instead, conventional operating system services, such as virtual memory and scheduling, are implemented as libraries executing in an application's address space. System service code executing in a library can be changed by the application according to its needs. *SPIN* shares many of the same goals as Aegis although its approach is quite different. *SPIN* uses language facilities to protect the kernel from extensions and implements protected communication using procedure call. Using this infrastructure, *SPIN* provides an extension model and a core set of extensible services. In contrast, Aegis relies on hardware protected system calls to isolate extensions from the kernel and leaves unspecified the manner by which those extensions are defined or applied.

Several systems [Cooper et al. 91, Redell et al. 80, Mossenbock 94, Organick 73] like *SPIN*, have relied on language features to extend operating system services. Pilot, for instance, was a single-address space system that ran programs written in Mesa [Geschke et al. 77], an ancestor of Modula-3. In general, systems such as Pilot have depended on the language for all protection in the system, not just for the protection of the operating system and its extensions. In contrast, *SPIN*'s reliance on language services applies only to extension code within the kernel. Virtual address spaces are used to otherwise isolate the operating system and programs from one another.

3 The *SPIN* Architecture

The *SPIN* architecture provides a software infrastructure for safely combining system and application code. The protection model supports efficient, fine-grained access control of resources, while the extension model enables extensions to be defined at the granularity of a procedure call. The system's architecture is biased towards mechanisms that can be implemented with low-cost on conventional processors. Consequently, *SPIN* makes few demands of the hardware, and instead relies on language-level services, such as static typechecking and dynamic linking.

Relevant properties of Modula-3

SPIN and its extensions are written in Modula-3, a general purpose programming language designed in the early 1990's. The key features of the language include support for interfaces, type safety, automatic storage management, objects, generic interfaces, threads, and exceptions. We rely on the language's support for objects, generic interfaces, threads, and exceptions for aesthetic reasons only; we find that these features simplify the task of constructing a large system.

The design of *SPIN* depends only on the language's safety and encapsulation mechanisms; specifically interfaces, type safety, and automatic storage management. An interface declares the visible parts of an implementation *module*, which defines the items listed in the interface. All other definitions within the implementation module are hidden. The compiler enforces this restriction at compile-time. Type safety prevents code from accessing memory arbitrarily. A pointer may only refer to objects of its referent's type, and array indexing operations must be checked for bounds violation. The first restriction is enforced at compile-time, and the second is enforced through a combination of compile-time and run-time checks. Automatic storage management prevents memory used by a live pointer's referent from being returned to the heap and reused for an object of a different type.

3.1 The protection model

A protection model controls the set of operations that can be applied to resources. For example, a protection model based on address spaces ensures that a process can only access memory within a particular range of virtual addresses. Address spaces, though, are frequently inadequate for the fine-grained protection and management of resources, being expensive to create and slow to access [Lazowska et al. 81].

Capabilities

All kernel resources in *SPIN* are referenced by capabilities. A capability is an unforgeable reference to a resource which can be a system object, an interface, or a collection of interfaces. An example of each of these is a physical page, a physical page allocation interface, and the entire virtual memory system. Individual resources are protected to ensure that extensions reference only the resources to which they have been given access. Interfaces and collections of interfaces are protected to allow different extensions to have different views on the set of available services.

Unlike other operating systems based on capabilities, which rely on special-purpose hardware [Carter et al. 94], virtual memory mechanisms [Wulf et al. 81], probabilistic protection [Engler et al. 94], or protected message channels [Black et al. 92], *SPIN* implements capabilities directly using pointers, which are supported by the language. A pointer is a reference to a block of memory whose type is declared within an interface. Figure 1 demonstrates the definition and use of interfaces and capabilities (pointers) in *SPIN*.

The compiler, *at compile-time*, prevents a pointer from being forged or dereferenced in a way inconsistent with its type. There is no run-time overhead for using a pointer, passing it across an interface, or dereferencing it, other than the overhead of going to memory to access the pointer or its referent. A pointer can be passed from the kernel to a user-level application, which cannot be assumed to be type safe, as an *externalized reference*. An externalized reference is an index into a per-application table that contains type safe references to in-kernel data structures. The references can later be recovered using the index. Kernel services that intend to pass a reference out to user level externalize the reference through this table and instead pass out the index.

Protection domains

A protection domain defines the set of accessible names available to an execution context. In a conventional operating system, a protection domain is implemented using virtual address spaces. A name within one domain, a virtual address, has no relationship to that same name in another domain. Only through explicit mapping and sharing operations is it possible for names to become meaningful between protection domains.

```
INTERFACE Console; (* An interface. *)
TYPE T <: REFANY;  (* Read as "Console.T is opaque." *)

CONST InterfaceName = "ConsoleService";
  (* A global name *)

PROCEDURE Open():T;
  (* Open returns a capability for the console. *)
PROCEDURE Write(t: T; msg: TEXT);
PROCEDURE Read(t: VAR msg: TEXT);
PROCEDURE Close(t: T);
END Console;
```

```
MODULE Console;  (* An implementation module. *)

(* The implementation of Console.T *)
TYPE Buf = ARRAY [0..31] OF CHAR;
REVEAL T = BRANDED REF RECORD   (* T is a pointer *)
        inputQ: Buf;            (* to a record *)
        outputQ: Buf;
        (* device specific info *)
    END;

(* Implementations of interface functions   *)
(* have direct access to the revealed type. *)
PROCEDURE Open():T = ...
END Console;
```

```
MODULE Gatekeeper; (* A client *)
IMPORT Console;

VAR c: Console.T;  (* A capability for *)
                   (* the console device *)
PROCEDURE IntruderAlert() =
  BEGIN
     c := Console.Open();
     Console.Write(c, "Intruder Alert");
     Console.Close(c);
  END IntruderAlert;

BEGIN
END Gatekeeper;
```

Figure 1: *The Gatekeeper module interacts with* SPIN*'s Console service through the Console interface. Although Gatekeeper.IntruderAlert manipulates objects of type Console.T, it is unable to access the fields within the object, even though it executes within the same virtual address space as the Console module.*

In *SPIN* the naming and protection interface is at the level of the language, not of the virtual memory system. Consequently, namespace management must occur at the language level. For example, if the name c is an instance of the type $Console.T$, then both c and $Console.T$ occupy a portion of some symbolic namespace. An extension that redefines the type $Console.T$, creates an instance of the new type, and passes it to a module expecting a $Console.T$ of the original type creates a type conflict that results in an error. The error could be avoided by placing all extensions into a global module space, but since modules, procedures, and variable names are visible to programmers, we felt that this would introduce an overly restrictive programming model for the system. Instead, *SPIN* provides facilities for creating, coordinating, and linking program-level namespaces in the context of protection domains.

```
INTERFACE Domain;

TYPE T <: REFANY;  (* Domain.T is opaque *)

PROCEDURE Create(coff:CoffFile.T):T;
(* Returns a domain created from the specified object
   file (``coff'' is a standard object file format). *)

PROCEDURE CreateFromModule():T;
(* Create a domain containing interfaces defined by the
   calling module.  This function allows modules to
   name and export themselves at runtime. *)

PROCEDURE Resolve(source,target: T);
(* Resolve any undefined symbols in the target domain
   against any exported symbols from the source.*)

PROCEDURE Combine(d1, d2: T):T;
(* Create a new aggregate domain that exports the
   interfaces of the given domains. *)

END Domain.
```

Figure 2: *The Domain interface. This interface operates on instances of type Domain.T, which are described by type safe pointers. The implementation of the Domain interface is unsafe with respect to Modula-3 memory semantics, as it must manipulate linker symbols and program addresses directly.*

A *SPIN* protection domain defines a set of names, or program symbols, that can be referenced by code with access to the domain. A domain, named by a capability, is used to control dynamic linking, and corresponds to one or more safe object files with one or more exported interfaces. An object file is safe if it is unknown to the kernel but has been signed by the Modula-3 compiler, or if the kernel can otherwise assert the object file to be safe. For example, *SPIN*'s lowest level device interface is identical to the DEC OSF/1 driver interface [Dig 93], allowing us to dynamically link vendor drivers into the kernel. Although the drivers are written in C, the kernel asserts their safety. In general, we prefer to avoid using object files that are "safe by assertion" rather than by compiler verification, as they tend to be the source of more than their fair share of bugs.

Domains can be intersecting or disjoint, enabling applications to share services or define new ones. A domain is created using the *Create* operation, which initializes a domain with the contents of a safe object file. Any symbols exported by interfaces defined in the object file are exported from the domain, and any imported symbols are left unresolved. Unresolved symbols correspond to interfaces imported by code within the domain for which implementations have not yet been found.

The *Resolve* operation serves as the basis for dynamic linking. It takes a target and a source domain, and resolves any unresolved symbols in the target domain against symbols exported from the source. During resolution, text and data symbols are patched in the target domain, ensuring that, once resolved, domains are able to share resources at memory speed. Resolution only resolves the target domain's undefined symbols; it does not cause additional symbols to be exported. Cross-linking, a common idiom, occurs through a pair of *Resolve* operations.

The *Combine* operation creates linkable namespaces that are the union of existing domains, and can be used to bind together collections of related interfaces. For example, the domain *SpinPublic* combines the system's public interfaces into a single domain available to extensions. Figure 2 summarizes the major operations on domains.

The domain interface is commonly used to import or export particular named interfaces. A module that exports an interface explicitly creates a domain for its interface, and exports the domain through an in-kernel nameserver. The exported name of the interface, which can be specified within the interface, is used to coordinate the export and import as in many RPC systems [Schroeder & Burrows 90, Brockschmidt 94]. The constant *Console.InterfaceName* in Figure 1 defines a name that exporters and importers can use to uniquely identify a particular version of a service.

Some interfaces, such as those for devices, restrict access at the time of the import. An exporter can register an authorization procedure with the nameserver that will be called with the identity of the importer whenever the interface is imported. This fine-grained control has low cost because the importer, exporter, and authorizer interact through direct procedure calls.

3.2 The extension model

An extension changes the way in which a system provides service. All software is extensible in one way or another, but it is the extension model that determines the ease, transparency, and efficiency with which an extension can be applied. *SPIN*'s extension model provides a controlled communication facility between extensions and the base system, while allowing for a variety of interaction styles. For example, the model allows extensions to passively monitor system activity, and provide up-to-date performance information to applications. Other extensions may offer hints to the system to guide certain operations, such as page replacement. In other cases, an extension may entirely replace an existing system service, such as a scheduler, with a new one more appropriate to a specific application.

Extensions in *SPIN* are defined in terms of events and handlers. An event is a message that announces a change in the state of the system or a request for service. An event handler is a procedure that receives the message. An extension installs a handler on an event by explicitly registering the handler with the event through a central *dispatcher* that routes events to handlers.

Event names are protected by the domain machinery described in the previous section. An event is defined as a procedure exported from an interface and its handlers are defined as procedures having the same type. A handler is invoked with the arguments specified by the event raiser.[1] The kernel is preemptive, ensuring that a handler cannot take over the processor.

The right to call a procedure is equivalent to the right to raise the event named by the procedure. In fact, the two are indistinguishable in *SPIN*, and any procedure exported by an interface is also an event. The dispatcher exploits this similarity to optimize event raise as a direct procedure call where there is only one handler for a given event. Otherwise, the dispatcher uses dynamic code generation [Engler & Proebsting 94] to construct optimized call paths from the raiser to the handlers.

The primary right to handle an event is restricted to the default implementation module for the event, which is the module that statically exports the procedure named by the event. For example, the module *Console* is the default implementation module for the event *Console.Open()* shown in Figure 1. Other modules may request that the dispatcher install additional handlers or even remove the primary handler. For each request, the dispatcher contacts the primary implementation module, passing the event name provided by the installer. The implementation module can deny or allow the installation. If denied, the installation fails. If allowed, the implementation module can provide a *guard* to be associated with the handler. The guard defines a predicate, expressed as a procedure, that is evaluated by the dispatcher prior to the handler's invocation. If the predicate is true when the event is raised, then the handler is invoked; otherwise the handler is ignored.

Guards are used to restrict access to events at a granularity finer than the event name, allowing events to be dispatched on a per-instance basis. For example, the *SPIN* extension that implements IP layer processing defines the event *IP.PacketArrived(pkt: IP.Packet)*, which it raises whenever an IP packet is received. The IP module, which defines the default implementation of the *PacketArrived* event, upon each installation, constructs a guard that compares the type field in the header of the incoming packet against the set of IP protocol types that the handler may service. In this way, IP does not

[1] The dispatcher also allows a handler to specify an additional closure to be passed to the handler during event processing. The closure allows a single handler to be used within more than one context.

have to export a separate interface for each event instance. A handler can stack additional guards on an event, further constraining its invocation.

There may be any number of handlers installed on a particular event. The default implementation module may constrain a handler to execute synchronously or asynchronously, in bounded time, or in some arbitrary order with respect to other handlers for the same event. Each of these constraints reflects a different degree of trust between the default implementation and the handler. For example, a handler may be bounded by a time quantum so that it is aborted if it executes too long. A handler may be asynchronous, which causes it to execute in a separate thread from the raiser, isolating the raiser from handler latency. When multiple handlers execute in response to an event, a single result can be communicated back to the raiser by associating with each event a procedure that ultimately determines the final result [Pardyak & Bershad 94]. By default, the dispatcher mimics procedure call semantics, and executes handlers synchronously, to completion, in undefined order, and returns the result of the final handler executed.

4 The core services

The *SPIN* protection and extension mechanisms described in the previous section provide a framework for managing interfaces between services within the kernel. Applications, though, are ultimately concerned with manipulating resources such as memory and the processor. Consequently, *SPIN* provides a set of *core* services that manage memory and processor resources. These services, which use events to communicate between the system and extensions, export interfaces with fine-grained operations. In general, the service interfaces that are exported to extensions within the kernel are similar to the secondary internal interfaces found in conventional operating systems; they provide simple functionality over a small set of objects. In *SPIN* it is straightforward to allocate a single virtual page, a physical page, and then create a mapping between the two. Because the overhead of accessing each of these operations is low (a procedure call), it is feasible to provide them as interfaces to separate abstractions, and to build up higher level abstractions through direct composition. By contrast, traditional operating systems aggregate simpler abstractions into more complex ones, because the cost of repeated access to the simpler abstractions is too high.

4.1 Extensible memory management

A memory management system is responsible for the allocation of virtual addresses, physical addresses, and mappings between the two. Other systems have demonstrated significant performance improvements from specialized or "tuned" memory management policies that are accessible through interfaces exposed by the memory management system. Some of these interfaces have made it possible to manipulate large objects, such as entire address spaces [Young et al. 87, Khalidi & Nelson 93], or to direct expensive operations, for example page-out [Harty & Cheriton 91, McNamee & Armstrong 90], entirely from user level. Others have enabled control over relatively small objects, such as cache pages [Romer et al. 94] or TLB entries [Bala et al. 94], entirely from the kernel. None have allowed for fast, fine-grained control over the physical and virtual memory resources required by applications. *SPIN*'s virtual memory system provides such control, and is enabled by the system's low-overhead invocation and protection services.

The *SPIN* memory management interface decomposes memory services into three basic components: physical storage, naming, and translation. These correspond to the basic memory resources exported by processors, namely physical addresses, virtual addresses, and translations. Application-specific services interact with these three services to define higher level virtual memory abstractions, such as address spaces.

Each of the three basic components of the memory system is provided by a separate service interface, described in Figure 3. The *physical address service* controls the use and allocation of physical pages. Clients raise the *Allocate* event to request physical memory with a certain size and an optional series of attributes that reflect preferences for machine specific parameters such as color or contiguity. A physical page represents a unit of high speed storage. It is not, for most purposes, a nameable entity and may not be addressed directly from an extension or a user program. Instead, clients of the physical address service receive a capability for the memory. The *virtual address service* allocates capabilities for virtual addresses, where the capability's referent is composed of a virtual address, a length, and an address space identifier that makes the address unique. The *translation service* is used to express the relationship between virtual addresses and physical memory. This service interprets references to both virtual and physical addresses, constructs mappings between the two, and installs the mappings into the processor's memory management unit (MMU).

The translation service raises a set of events that correspond to various exceptional MMU conditions. For example, if a user program attempts to access an unallocated virtual memory address, the *Translation.BadAddress* event is raised. If it accesses an allocated, but unmapped virtual page, then the *Translation.PageNotPresent* event is raised. Implementors of higher level memory management abstractions can use these events to define services, such as demand paging, copy-on-write [Rashid et al. 87], distributed shared memory [Carter et al. 91], or concurrent garbage collection [Appel & Li 91].

The physical page service may at any time reclaim physical memory by raising the *PhysAddr.Reclaim* event. The interface allows the handler for this event to volunteer an alternative page, which may be of less importance than the candidate page. The translation ser-

```
INTERFACE PhysAddr;

TYPE T <: REFANY; (* PhysAddr.T is opaque *)

PROCEDURE Allocate(size: Size; attrib: Attrib): T;
(* Allocate some physical memory with particular
   attributes. *)

PROCEDURE Deallocate(p: T);

PROCEDURE Reclaim(candidate: T): T;
(* Request to reclaim a candidate page.  Clients
   may handle this event to nominate
   alternative candidates. *)

END PhysAddr.
```

```
INTERFACE VirtAddr;

TYPE T <: REFANY; (* VirtAddr.T is opaque *)

PROCEDURE Allocate(size: Size; attrib: Attrib): T;
PROCEDURE Deallocate(v: T);
END VirtAddr.
```

```
INTERFACE Translation;
IMPORT PhysAddr, VirtAddr;

TYPE T <: REFANY;  (* Translation.T is opaque *)

PROCEDURE Create(): T;
PROCEDURE Destroy(context: T);
(* Create or destroy an addressing context *)

PROCEDURE AddMapping(context: T; v: VirtAddr.T;
          p: PhysAddr.T; prot: Protection);
(* Add [v,p] into the named translation context
   with the specified protection. *)

PROCEDURE RemoveMapping(context: T; v: VirtAddr.T);

PROCEDURE ExamineMapping(context: T;
          v: VirtAddr.T): Protection;

(* A few events raised during *)
(* illegal translations *)
PROCEDURE PageNotPresent(v: T);
PROCEDURE BadAddress(v: T);
PROCEDURE ProtectionFault(v: T);

END Translation.
```

Figure 3: *The interfaces for managing physical addresses, virtual addresses, and translations.*

vice ultimately invalidates any mappings to a reclaimed page.

The *SPIN* core services do not define an address space model directly, but can be used to implement a range of models using a variety of optimization techniques. For example, we have built an extension that implements UNIX address space semantics for applications. It exports an interface for copying an existing address space, and for allocating additional memory within one. For each new address space, the extension allocates a new context from the translation service. This context is subsequently filled in with virtual and physical address resources obtained from the memory allocation services. Another kernel extension defines a memory management interface supporting Mach's *task* abstraction [Young et al. 87]. Applications may use these interfaces, or they may define their own in terms of the lower-level services.

4.2 Extensible thread management

An operating system's thread management system provides applications with interfaces for scheduling, concurrency, and synchronization. Applications, though, can require levels of functionality and performance that a thread management system is unable to deliver. User-level thread management systems have addressed this mismatch[Wulf et al. 81, Cooper & Draves 88, Marsh et al. 91, Anderson et al. 92], but only partially. For example, Mach's user-level C-Threads implementation [Cooper & Draves 88] can have anomalous behavior because it is not well-integrated with kernel services [Anderson et al. 92]. In contrast, *scheduler activations*, which are integrated with the kernel, have high communication overhead [Davis et al. 93].

In *SPIN* an application can provide its own thread package and scheduler that executes within the kernel. The thread package defines the application's execution model and synchronization constructs. The scheduler controls the multiplexing of the processor across multiple threads. Together these packages allow an application to define arbitrary thread semantics and to implement those semantics close to the processor and other kernel services.

Although *SPIN* does not define a thread model for applications, it does define the structure on which an implementation of a thread model rests. This structure is defined by a set of events that are raised or handled by schedulers and thread packages. A scheduler multiplexes the underlying processing resources among competing contexts, called *strands*. A strand is similar to a thread in traditional operating systems in that it reflects some processor context. Unlike a thread though, a strand has no minimal or requisite kernel state other than a name. An application-specific thread package defines an implementation of the strand interface for its own threads.

Together, the thread package and the scheduler implement the control flow mechanisms for user-space contexts. Figure 4 describes this interface. The interface contains two events, *Block* and *Unblock*, that can be raised to signal changes in a strand's execution state. A disk driver can direct a scheduler to block the current strand during an I/O operation, and an interrupt handler can unblock a strand to signal the completion of the I/O operation. In response to these events, the scheduler can communicate with the thread package managing the strand using *Checkpoint* and *Resume* events, allowing the package to save and restore execution state.

```
INTERFACE Strand;

TYPE T <: REFANY;  (* Strand.T is opaque *)

PROCEDURE Block(s:T);
(* Signal to a scheduler that s is not runnable. *)

PROCEDURE Unblock(s: T);
(* Signal to a scheduler that s is runnable. *)

PROCEDURE Checkpoint(s: T);
(* Signal that s is being descheduled and that it
   should save any processor state required for
   subsequent rescheduling. *)

PROCEDURE Resume(s: T);
(* Signal that s is being placed on a processor and
   that it should reestablish any state saved during
   a prior call to Checkpoint. *)

END Strand.
```

Figure 4: *The Strand Interface. This interface describes the scheduling events affecting control flow that can be raised within the kernel. Application-specific schedulers and thread packages install handlers on these events, which are raised on behalf of particular strands. A trusted thread package and scheduler provide default implementations of these operations, and ensure that extensions do not install handlers on strands for which they do not possess a capability.*

Application-specific thread packages only manipulate the flow of control for application threads executing outside of the kernel. For safety reasons, the responsibility for scheduling and synchronization within the kernel belongs to the kernel. As a thread transfers from user mode to kernel mode, it is checkpointed and a Modula-3 thread executes in the kernel on its behalf. As the Modula-3 thread leaves the kernel, the blocked application-specific thread is resumed.

A global scheduler implements the primary processor allocation policy between strands. Additional application-specific schedulers can be placed on top of the global scheduler using *Checkpoint* and *Resume* events to relinquish or receive control of the processor. That is, an application-specific scheduler presents itself to the global scheduler as a thread package. The delivery of the *Resume* event indicates that the new scheduler can schedule its own strands, while *Checkpoint* signals that the processor is being reclaimed by the global scheduler.

The *Block* and *Unblock* events, when raised on strands scheduled by application-specific schedulers, are routed by the dispatcher to the appropriate scheduling implementation. This allows new scheduling policies to be implemented and integrated into the kernel, provided that an application-specific policy does not conflict with the global policy. While the global scheduling policy is replaceable, it cannot be replaced by an arbitrary application, and its replacement can have global effects. In the current implementation, the global scheduler implements a round-robin, preemptive, priority policy.

We have used the strand interface to implement as kernel extensions a variety of thread management interfaces including DEC OSF/1 kernel threads [Dig 93], C-Threads [Cooper & Draves 88], and Modula-3 threads. The implementations of these interfaces are built directly from strands and not layered on top of others. The interface supporting DEC OSF/1 kernel threads allows us to incorporate the vendor's device drivers directly into the kernel. The C-Threads implementation supports our UNIX server, which uses the Mach C-Threads interface for concurrency. Within the kernel, a trusted thread package and scheduler implements the Modula-3 thread interface [Nelson 91].

4.3 Implications for trusted services

The processor and memory services are two instances of *SPIN*'s core services, which provide interfaces to hardware mechanisms. The core services are *trusted*, which means that they must perform according to their interface specification. Trust is required because the services access underlying hardware facilities and at times must step outside the protection model enforced by the language. Without trust, the protection and extension mechanisms described in the previous section could not function safely, as they rely on the proper management of the hardware. Because trusted services mediate access to physical resources, applications and extensions must trust the services that are trusted by the *SPIN* kernel.

In designing the interfaces for *SPIN*'s trusted services, we have worked to ensure that an extension's failure to use an interface correctly is isolated to the extension itself (and any others that rely on it). For example, the *SPIN* scheduler raises events that are handled by application-specific thread packages in order to start or stop threads. Although it is in the handler's best interests to respect, or at least not interfere with, the semantics implied by the event, this is not enforced. An application-specific thread package may ignore the event that a particular user-level thread is runnable, but only the application using the thread package will be affected. In this way, the failure of an extension is no more catastrophic than the failure of code executing in the runtime libraries found in conventional systems.

5 System performance

In this section we show that *SPIN* enables applications to compose system services in order to define new kernel services that perform well. Specifically, we evaluate the performance of *SPIN* from four perspectives:

- *System size.* The size of the system in terms of lines of code and object size demonstrates that advanced runtime services do not necessarily create an operating system kernel of excessive size. In addition, the size of the system's extensions shows that they can be implemented with reasonable amounts of code.

- *Microbenchmarks.* Measurements of low-level system services, such as protected communication, thread management and virtual memory, show that *SPIN*'s extension architecture enables us to construct communication-intensive services with low overhead. The measurements also show that conventional system mechanisms, such as a system call and cross-address space protected procedure call, have overheads that are comparable to those in conventional systems.

- *Networking.* Measurements of a suite of networking protocols demonstrate that *SPIN*'s extension architecture enables the implementation of high-performance network protocols.

- *End-to-end performance.* Finally, we show that end-to-end application performance can benefit from *SPIN*'s architecture by describing two applications that use system extensions.

We compare the performance of operations on three operating systems that run on the same platform: *SPIN* (V0.4 of August 1995), DEC OSF/1 V2.1 which is a monolithic operating system, and Mach 3.0 which is a microkernel. We collected our measurements on DEC Alpha 133MHz AXP 3000/400 workstations, which are rated at 74 SPECint 92. Each machine has 64 MBs of memory, a 512KB unified external cache, an HP C2247-300 1GB disk-drive, a 10Mb/sec Lance Ethernet interface, and a FORE TCA-100 155Mb/sec ATM adapter card connected to a FORE ASX-200 switch. The FORE cards use programmed I/O and can maximally deliver only about 53Mb/sec between a pair of hosts [Brustoloni & Bershad 93]. We avoid comparisons with operating systems running on different hardware as benchmarks tend to scale poorly for a variety of architectural reasons [Anderson et al. 91]. All measurements are taken while the operating systems run in single-user mode.

5.1 System components

SPIN runs as a standalone kernel on DEC Alpha workstations. The system consists of five main components, *sys, core, rt, lib* and *sal,* that support different classes of service. Table 1 shows the size of each component in source lines, object bytes, and percentages. The first component, *sys*, implements the extensibility machinery, domains, naming, linking, and dispatching. The second component, *core*, implements the virtual memory and scheduling services described in the previous section, as well as device management, a disk-based and network-based file system, and a network debugger [Redell 88]. The third component, *rt*, contains a version of the DEC SRC Modula-3 runtime system that supports automatic memory management and exception processing. The fourth component, *lib*, includes a subset of the standard Modula-3 libraries and handles many of the more mundane data structures (lists, queues, hash tables, etc.) generally required by any operating system kernel. The final component, *sal*, implements a low-level interface to device drivers and the MMU, offering functionality such as "install a page table entry," "get a character from the console," and "read block 22 from SCSI unit 0." We build *sal* by applying a few dozen file diffs against a small subset of the files from the DEC OSF/1 kernel source tree. This approach, while increasing the size of the kernel, allows us to track the vendor's hardware without requiring that we port *SPIN* to each new system configuration.

Component	Source size		Text size		Data size	
	lines	%	bytes	%	bytes	%
sys	1646	2.5	42182	5.2	22397	5.0
core	10866	16.5	170380	21.0	89586	20.0
rt	14216	21.7	176171	21.8	104738	23.4
lib	1234	1.9	10752	1.3	3294	.8
sal	37690	57.4	411065	50.7	227259	50.8
Total kernel	65652	100	810550	100	447274	100

Table 1: *This table shows the size of different components of the system. The sys, core and rt components contain the interfaces visible to extensions. The column labeled "lines" does not include comments. We use the DEC SRC Modula-3 compiler, release 3.5.*

5.2 Microbenchmarks

Microbenchmarks reveal the overhead of basic system functions, such a protected procedure call, thread management, and virtual memory. They define the bounds of system performance and provide a framework for understanding larger operations. Times presented in this section, measured with the Alpha's internal cycle counter, are the average of a large number of iterations, and may therefore be overly optimistic regarding cache effects [Bershad et al. 92a].

Protected communication

In a conventional operating system, applications, services and extensions communicate using two protected mechanisms: system calls and cross-address space calls. The first enables applications and kernel services to interact. The second enables interaction between applications and services that are not part of the kernel. The overhead of using either of these mechanisms is the limiting factor in a conventional system's extensibility. High overhead discourages frequent interaction, requiring that a system be built from coarse-grained interfaces to amortize the cost of communication over large operations.

SPIN's extension model offers a third mechanism for protected communication. Simple procedure calls, rather than system calls, can be used for communication between extensions and the core system. Similarly, simple procedure calls, rather than cross-address procedure calls, can be used for communication between applications and other services installed into the kernel.

In Table 2 we compare the performance of the different protected communication mechanisms when invoking the "null procedure call" on DEC OSF/1, Mach, and *SPIN*. The null procedure call takes no arguments and returns no results; it reflects only the cost of control transfer. The protected in-kernel call in *SPIN* is implemented as a procedure call between two domains that have been dynamically linked. Although this test does not measure data transfer, the overhead of passing arguments between domains, even large arguments, is small because they can be passed by reference. System call overhead reflects the time to cross the user-kernel boundary, execute a procedure and return. In Mach and DEC OSF/1, system calls flow from the trap handler through to a generic, but fixed, system call dispatcher, and from there to the requested system call (written in C). In *SPIN*, the kernel's trap handler raises a *Trap.SystemCall* event which is dispatched to a Modula-3 procedure installed as a handler. The third line in the table shows the time to perform a protected, cross-address space procedure call. DEC OSF/1 supports cross-address space procedure call using sockets and SUN RPC. Mach provides an optimized path for cross-address space communication using messages [Draves 94]. *SPIN*'s cross-address space procedure call is implemented as an extension that uses system calls to transfer control in and out of the kernel and cross-domain procedure calls within the kernel to transfer control between address spaces.

Operation	DEC OSF/1	Mach	*SPIN*
Protected in-kernel call	n/a	n/a	.13
System call	5	7	4
Cross-address space call	845	104	89

Table 2: *Protected communication overhead in microseconds. Neither DEC OSF/1 nor Mach support protected in-kernel communication.*

The table illustrates two points about communication and system structure. First, the overhead of protected communication in *SPIN* can be that of procedure call for extensions executing in the kernel's address space. *SPIN*'s protected in-kernel calls provide the same functionality as cross-address space calls in DEC OSF/1 and Mach, namely the ability to execute arbitrary code in response to an application's call. Second, *SPIN*'s extensible architecture does not preclude the use of traditional communication mechanisms having performance comparable to that in non-extensible systems. However, the disparity between the performance of a protected in-kernel call and the other mechanisms encourages the use of in-kernel extensions.

SPIN's in-kernel protected procedure call time is conservative. Our Modula-3 compiler generates code for which an intermodule call is roughly twice as slow as an intramodule call. A more recent version of the Modula-3 compiler corrects this disparity. In addition, our compiler does not perform inlining, which can be an important optimization when calling many small procedures. These optimizations do not affect the semantics of the language and will therefore not change the system's protection model.

Thread management

Thread management packages implement concurrency control operations using underlying kernel services. As previously mentioned, *SPIN*'s in-kernel threads are implemented with a trusted thread package exporting the Modula-3 thread interface. Application-specific extensions also rely on threads executing in the kernel to implement their own concurrent operations. At user level, thread management overhead determines the granularity with which threads can be used to control concurrent user-level operations.

Table 3 shows the overhead of thread management operations for kernel and user threads using the different systems. *Fork-Join* measures the time to create, schedule, and terminate a new thread, synchronizing the termination with another thread. *Ping-Pong* reflects synchronization overhead, and measures the time for a pair of threads to synchronize with one another; the first thread signals the second and blocks, then the second signals the first and blocks.

We measure kernel thread overheads using the native primitives provided by each kernel (*thread_sleep* and *thread_wakeup* in DEC OSF/1 and Mach, and locks with condition variables in *SPIN*). At user-level, we measure the performance of the same program using C-Threads on Mach and *SPIN*, and P-Threads, a C-Threads superset, on DEC OSF/1. The table shows measurements for two implementations of C-Threads on *SPIN*. The first implementation, labeled "layered," is implemented as a user-level library layered on a set of kernel extensions that implement Mach's kernel thread interface. The second implementation, labeled "integrated," is structured as a kernel extension that exports the C-Threads interface using system calls. The latter version uses *SPIN*'s strand interface, and is integrated with the scheduling behavior of the rest of the kernel. The table shows that *SPIN*'s extensible thread implementation does not incur a performance penalty when compared to non-extensible ones, even when integrated with kernel services.

Virtual memory

Applications can exploit the virtual memory fault path to extend system services [Appel & Li 91]. For example, concurrent and generational garbage collectors can use write faults to maintain invariants or collect reference information. A longstanding problem with fault-based

Operation	DEC OSF/1 kernel	DEC OSF/1 user	Mach kernel	Mach user	*SPIN* kernel	*SPIN* user layered	*SPIN* user integrated
Fork-Join	198	1230	101	338	22	262	111
Ping-Pong	21	264	71	115	17	159	85

Table 3: *Thread management overhead in microseconds.*

strategies has been the overhead of handling a page fault in an application [Thekkath & Levy 94, Anderson et al. 91]. There are two sources of this overhead. First, handling each fault in a user application requires crossing the user/kernel boundary several times. Second, conventional systems provide quite general exception interfaces that can perform many functions at once. As a result, applications requiring only a subset of the interface's functionality must pay for all of it. *SPIN* allows applications to define specialized fault handling extensions to avoid user/kernel boundary crossings and implement precisely the functionality that is required.

Table 4 shows the time to execute several commonly referenced virtual memory benchmarks [Appel & Li 91, Engler et al. 95]. The line labeled *Dirty* in the table measures the time for an application to query the status of a particular virtual page. Neither DEC OSF/1 nor Mach provide this facility. The time shown in the table is for an extension to invoke the virtual memory system; an additional 4 microseconds (system call time) is required to invoke the service from user level. *Trap* measures the latency between a page fault and the time when a handler executes. *Fault* is the perceived latency of the access from the standpoint of the faulting thread. It measures the time to reflect a page fault to an application, enable access to the page within a handler, and resume the faulting thread. *Prot1* measures the time to increase the protection of a single page. Similarly, *Prot100* and *Unprot100* measure the time to increase and decrease the protection over a range of 100 pages. Mach's unprotection is faster than protection since the operation is performed lazily; *SPIN*'s extension does not lazily evaluate the request, but enables the access as requested. *Appel1* and *Appel2* measure a combination of traps and protection changes. The *Appel1* benchmark measures the time to fault on a protected page, resolve the fault in the handler, and protect another page in the handler. *Appel2* measures the time to protect 100 pages, and fault on each one, resolving the fault in the handler (*Appel2* is shown as the average cost per page).

SPIN outperforms the other systems on the virtual memory benchmarks for two reasons. First, *SPIN* uses kernel extensions to define application-specific system calls for virtual memory management. The calls provide access to the virtual and physical memory interfaces described in the previous section, and install handlers for *Translation.ProtectionFault* events that occur within the application's virtual address space. In contrast, DEC OSF/1 requires that applications use the UNIX signal and *mprotect* interfaces to manage virtual memory, and Mach requires that they use the external pager interface [Young et al. 87]. Neither signals nor external pagers, though, have especially efficient implementations, as the focus of each is generalized functionality [Thekkath & Levy 94]. The second reason for *SPIN*'s dominance is that each virtual memory event, which requires a series of interactions between the kernel and the application, is reflected to the application through a fast in-kernel protected procedure call. DEC OSF/1 and Mach, though, communicate these events by means of more expensive traps or messages.

Operation	DEC OSF/1	Mach	*SPIN*
Dirty	n/a	n/a	2
Fault	329	415	29
Trap	260	185	7
Prot1	45	106	16
Prot100	1041	1792	213
Unprot100	1016	302	214
Appel1	382	819	39
Appel2	351	608	29

Table 4: *Virtual memory operation overheads in microseconds. Neither DEC OSF/1 nor Mach provide an interface for querying the internal state of a page frame.*

5.3 Networking

We have used *SPIN*'s extension architecture to implement a set of network protocol stacks for Ethernet and ATM networks [Fiuczynski & Bershad 96]. Figure 5 illustrates the structure of the protocol stacks, which are similar to the *x*-kernel's [Hutchinson et al. 89] except that *SPIN* permits user code to be dynamically placed within the stack. Each incoming packet is "pushed" through the protocol graph by events and "pulled" by handlers. The handlers at the top of the graph can process the message entirely within the kernel, or copy it out to an application. The *RPC* and *A.M.* extensions, for example, implement the network transport for a remote procedure call package and active messages [von Eicken et al. 92]. The *video* extension provides a direct path for video packets from the network to the framebuffer. The *UDP* and *TCP* extensions support the Internet protocols.[2] The *Forward* extension provides transparent UDP/IP and TCP/IP forwarding for packets arriving on a specific port. Finally, the *HTTP* extension implements the HyperText Transport Protocol [Berners-Lee et al. 94] directly within the kernel, enabling a server to respond quickly to HTTP requests by splicing together the protocol stack and the local file system.

Latency and Bandwidth

Table 5 shows the round trip latency and reliable bandwidth between two applications using UDP/IP on DEC

[2] We currently use the DEC OSF/1 TCP engine as a *SPIN* extension, and manually assert that the code, which is written in C, is safe.

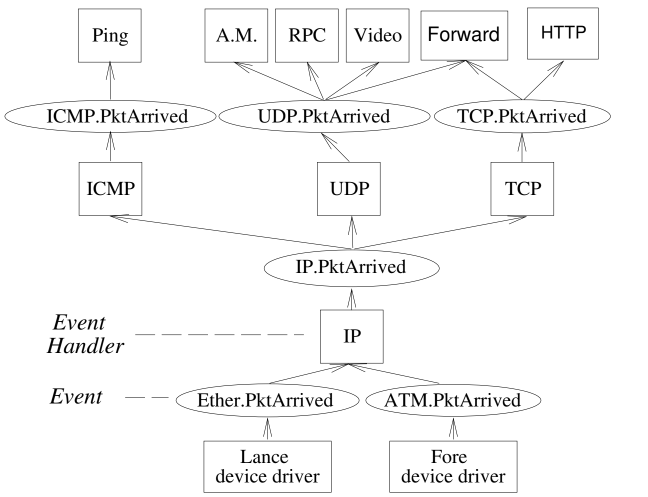

Figure 5: *This figure shows a protocol stack that routes incoming network packets to application-specific endpoints within the kernel. Ovals represent events raised to route control to handlers, which are represented by boxes. Handlers implement the protocol corresponding to their label.*

OSF/1 and *SPIN*. For DEC OSF/1, the application code executes at user level, and each packet sent involves a trap and several copy operations as the data moves across the user/kernel boundary. For *SPIN*, the application code executes as an extension in the kernel, where it has low-latency access to both the device and data. Each incoming packet causes a series of events to be generated for each layer in the UDP/IP protocol stack (Ethernet/ATM, IP, UDP) shown in Figure 5. For *SPIN*, protocol processing is done by a separately scheduled kernel thread outside of the interrupt handler. We do not present networking measurements for Mach, as the system neither provides a path to the Ethernet more efficient than DEC OSF/1, nor supports our ATM card.

	Latency		Bandwidth	
	DEC OSF/1	*SPIN*	DEC OSF/1	*SPIN*
Ethernet	789	565	8.9	8.9
ATM	631	421	27.9	33

Table 5: *Network protocol latency in microseconds and receive bandwidth in Mb/sec. We measure latency using small packets (16 bytes), and bandwidth using large packets (1500 for Ethernet and 8132 for ATM).*

The table shows that processing packets entirely within the kernel can reduce round-trip latency when compared to a system in which packets are handled in user space. Throughput, which tends not to be latency sensitive, is roughly the same on both systems.

We use the same vendor device drivers for both DEC OSF/1 and *SPIN* to isolate differences due to system architecture from those due to the characteristics of the underlying device driver. Neither the Lance Ethernet driver nor the FORE ATM driver are optimized for latency [Thekkath & Levy 93], and only the Lance Ethernet driver is optimized for throughput. Using different device drivers we achieve a round-trip latency of 337 μsecs on Ethernet and 241 μsecs on ATM, while reliable ATM bandwidth between a pair of hosts rises to 41 Mb/sec. We estimate the minimum round trip time using our hardware at roughly 250μsecs on Ethernet and 100μsecs on ATM. The maximum usable Ethernet and ATM bandwidths between a pair of hosts are roughly 9 Mb/sec and 53Mb/sec.

Protocol forwarding

SPIN's extension architecture can be used to provide protocol functionality not generally available in conventional systems. For example, some TCP redirection protocols [Balakrishnan et al. 95] that have otherwise required kernel modifications can be straightforwardly defined by an application as a *SPIN* extension. A forwarding protocol can also be used to load balance service requests across multiple servers.

In *SPIN* an application installs a node into the protocol stack which redirects all data and control packets destined for a particular port number to a secondary host. We have implemented a similar service using DEC OSF/1 with a user-level process that splices together an incoming and outgoing socket. The DEC OSF/1 forwarder is not able to forward protocol control packets because it executes above the transport layer. As a result it cannot maintain a protocol's end-to-end semantics. In the case of TCP, end-to-end connection establishment and termination semantics are violated. A user-level intermediary also interferes with the protocol's algorithms for window size negotiation, slow start, failure detection, and congestion control, possibly degrading the overall performance of connections between the hosts. Moreover, on the user-level forwarder, each packet makes two trips through the protocol stack where it is twice copied across the user/kernel boundary. Table 6 compares the latency for the two implementations, and reveals the additional work done by the user-level forwarder.

	TCP		UDP	
	DEC OSF/1	*SPIN*	DEC OSF/1	*SPIN*
Ethernet	2080	1420	1607	1344
ATM	1730	1067	1389	1024

Table 6: *Round trip latency in microseconds to route 16 byte packets through a protocol forwarder.*

5.4 End-to-end performance

We have implemented several applications that exploit *SPIN*'s extensibility. One is a networked video system that consists of a server and a client viewer. The server is structured as three kernel extensions, one that uses the local file system to read video frames from the disk, another that sends the video out over the network, and a third that registers itself as a handler on the *SendPacket* event, transforming the single send into a multicast to a list of clients. The server transmits 30 frames per second to each client. On the client, an extension awaits incoming video packets, decompresses and writes them directly to the frame buffer using the structure shown in Figure 5.

Because each outgoing packet is pushed through the protocol graph only once, and not once per client stream, *SPIN*'s server can support a larger number of clients than one that processes each packet in isolation. To show this, we measure processor utilization as a function of the number of clients for the *SPIN* server and for a server that runs on DEC OSF/1. The DEC OSF/1 server executes in user space and communicates with clients using sockets; each outgoing packet is copied into the kernel and is pushed through the kernel's protocol stack into the device driver. We determine processor utilization by measuring the progress of a low-priority idle thread that executes on the server.

Using the FORE interface, we find that both *SPIN* and DEC OSF/1 consume roughly the same fraction of the server's processor for a given number of clients. Although the *SPIN* server does less work in the protocol stack, the majority of the server's CPU resources are consumed by the programmed I/O that copies data to the network one word at a time. Using a network interface that supports DMA, though, we find that the *SPIN* server's processor utilization grows less slowly than the DEC OSF/1 server's. Figure 6 shows server processor utilization as a function of the number of supported client streams when the server is configured with a Digital T3PKT adapter. The "T3" is an experimental network interface that can send 45 Mb/sec using DMA. We use the same device driver in both operating systems. At 15 streams, both *SPIN* and DEC OSF/1 saturate the network, but *SPIN* consumes only half as much of the processor. Compared to DEC OSF/1, *SPIN* can support more clients on a faster network, or as many clients on a slower processor.

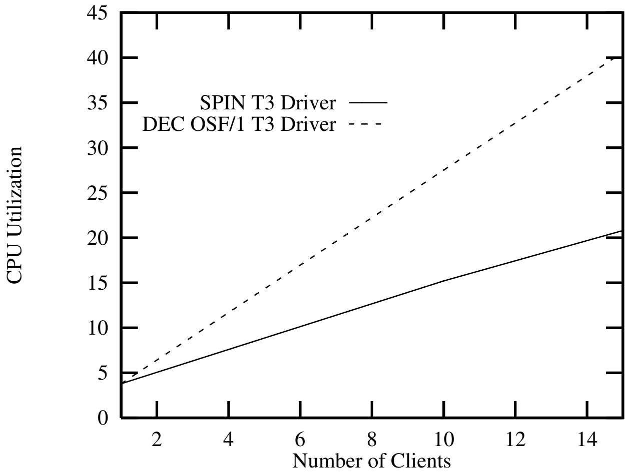

Figure 6: *Server utilization as a function of the number of client video streams. Each stream requires approximately 3 Mb/sec.*

Another application that can benefit from *SPIN*'s architecture is a web server. To service requests quickly, a web server should cache recently accessed objects, not cache large objects that are infrequently accessed [Chankhunthod et al. 95], and avoid double buffering with other caching agents [Stonebraker 81]. A server that does not itself cache but is built on top of a conventional caching file system avoids the double buffering problem, but is unable to control the caching policy. In contrast, a server that controls its own cache on top of the file system's suffers from double buffering.

SPIN allows a server to both control its cache and avoid the problem of double buffering. A *SPIN* web server implements its own hybrid caching policy based on file type: LRU for small files, and no-cache for large files which tend to be accessed infrequently. The client-side latency of an HTTP transaction to a *SPIN* web server running as a kernel extension is 5 milliseconds when the requested file is in the server's cache. Otherwise, the server goes through a non-caching file system to find the file. A comparable user-level web server on DEC OSF/1 that relies on the operating system's caching file system (no double buffering) takes about 8 milliseconds per request for the same cached file.

5.5 Other issues

Scalability and the dispatcher

SPIN's event dispatcher matches event raisers to handlers. Since every procedure in the system is effectively an event, the latency of the dispatcher is critical. As mentioned, in the case of a single synchronous handler, an event raise is implemented as a procedure call from the raiser to the handler. In other cases, such as when there are many handlers registered for a particular event, the dispatcher takes a more active role in event delivery. For each guard/handler pair installed on an event, the dispatcher evaluates the guard and, if true, invokes the handler. Consequently, dispatcher latency depends on the number and complexity of the guards, and the number of event handlers ultimately invoked. In practice, the overhead of an event dispatch is linear with the number of guards and handlers installed on the event. For example, round trip Ethernet latency, which we measure at 565 μsecs, rises to about 585 μsecs when 50 additional guards and handlers register interest in the arrival of some UDP packet but all 50 guards evaluate to false. When all 50 guards evaluate to true, latency rises to 637 μsecs. Presently, we perform no guard-specific optimizations such as evaluating common subexpressions [Yuhara et al. 94] or representing guard predicates as decision trees. As the system matures, we plan to apply these optimizations.

Impact of automatic storage management

An extensible system cannot depend on the correctness of unprivileged clients for its memory integrity. As previously mentioned, memory management schemes that allow extensions to return objects to the system heap are unsafe because a rogue client can violate the type system by retaining a reference to a freed object. *SPIN* uses a trace-based, mostly-copying, garbage collector [Bartlett 88] to safely reclaim memory resources. The collector serves as a safety net for untrusted extensions, and ensures that resources released by an extension, either through inaction or as a result of premature termination, are eventually reclaimed.

Clients that allocate large amounts of memory can trigger frequent garbage collections with adverse global effects. In practice, this is less of a problem than might be expected because *SPIN* and its extensions avoid allocation on fast paths. For example, none of the measurements presented in this section change when we disable the collector during the tests. Even in systems without garbage collection, generalized allocation is avoided because of its high latency. Instead, subsystems implement their own allocators optimized for some expected usage pattern. *SPIN* services do this as well and for the same reason (dynamic memory allocation is relatively expensive). As a consequence, there is less pressure on the collector, and the pressure is least likely to be applied during a critical path.

Size of extensions

Table 7 shows the size of some of the extensions described in this section. *SPIN* extensions tend to require an amount of code commensurate with their functionality. For example, the *Null syscall* and *IPC* extensions, are conceptually simple, and also have simple implementations. Extensions tend to import relatively few (about a dozen) interfaces, and use the domain and event system in fairly stylized ways. As a result, we have not found building extensions to be exceptionally difficult. In contrast, we had more trouble correctly implementing a few of our benchmarks on DEC OSF/1 or Mach, because we were sometimes forced to follow circuitous routes to achieve a particular level of functionality. Mach's external pager interface, for instance, required us to implement a complete pager in user space, although we were only interested in discovering write protect faults.

Component	Source size lines	Text size bytes	Data size bytes
NULL syscall	19	96	656
IPC	127	1344	1568
CThreads	219	2480	1792
DEC OSF/1 threads	305	2304	3488
VM workload	263	5712	1472
IP	744	19008	13088
UDP	1046	23968	16704
TCP	5077	69040	9840
HTTP	392	5712	4176
TCP Forward	187	4592	2080
UDP Forward	138	4592	2144
Video Client	95	2736	1952
Video Server	304	9228	3312

Table 7: *This table shows the size of some different system extensions described in this paper.*

6 Experiences with Modula-3

Our decision to use Modula-3 was made with some care. Originally, we had intended to define and implement a compiler for a safe subset of C. All of us, being C programmers, were certain that it was infeasible to build an efficient operating system without using a language having the syntax, semantics and performance of C. As the design of our safe subset proceeded, we faced the difficult issues that typically arise in any language design or redesign. For each major issue that we considered in the context of a safe version of C (type semantics, objects, storage management, naming, etc.), we found the issue already satisfactorily addressed by Modula-3. Moreover, we understood that the definition of our service interfaces was more important than the language with which we implemented them.

Ultimately, we decided to use Modula-3 for both the system and its extensions. Early on we found evidence to abandon our two main prejudices about the language: that programs written in it are slow and large, and that C programmers could not be effective using another language. In terms of performance, we have found nothing remarkable about the language's code size or execution time, as shown in the previous section. In terms of programmer effectiveness, we have found that it takes less than a day for a competent C programmer to learn the syntax and more obvious semantics of Modula-3, and another few days to become proficient with its more advanced features. Although anecdotal, our experience has been that the portions of the *SPIN* kernel written in Modula-3 are much more robust and easier to understand than those portions written in C.

7 Conclusions

The *SPIN* operating system demonstrates that it is possible to achieve good performance in an extensible system without compromising safety. The system provides a set of efficient mechanisms for extending services, as well as a core set of extensible services. Co-location, enforced modularity, logical protection domains and dynamic call binding allow extensions to be dynamically defined and accessed at the granularity of a procedure call.

In the past, system builders have only relied on the programming language to translate operating system policies and mechanisms into machine code. Using a programming language with the appropriate features, we believe that operating system implementors can more heavily rely on compiler and language runtime services to construct systems in which structure and performance are complementary.

Additional information about the *SPIN* project is available at *http://www-spin.cs.washington.edu*, an Alpha workstation running *SPIN* and the HTTP extension described in this paper.

Acknowledgements

Many people have contributed to the *SPIN* project. David Dion has been responsible for bringing up the system's UNIX server. Jan Sanislo made it possible for us to use the DEC OSF/1 SCSI driver from *SPIN*. Anthony Lamarca, Dylan McNamee, Geoff Voelker, and Alec Wolman assisted in understanding system performance on DEC OSF/1 and Mach. David Nichols, Hank Levy, and Terri Watson provided feedback on earlier drafts of this paper. David Boggs provided us with the T3 cards that we used in the video server experiment. Special thanks are due to DEC SRC, who provided us with much of our compiler infrastructure.

References

[Abrossimov et al. 89] Abrossimov, V., Rozier, M., and Shapiro, M. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 123–136, Litchfield Park, AZ, December 1989.

[Anderson et al. 91] Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 108–120, Santa Clara, CA, April 1991.

[Anderson et al. 92] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Appel & Li 91] Appel, W. and Li, K. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 96–107, Santa Clara, CA, April 1991.

[Bala et al. 94] Bala, K., Kaashoek, M. F., and Weihl, W. E. Software Prefetching and Caching for Translation Lookaside Buffers. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 243–253, Monterey, CA, November 1994.

[Balakrishnan et al. 95] Balakrishnan, H., Seshan, S., Amir, E., and Katz., R. H. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of the First ACM Conference on Mobile Computing and Networking*, November 1995.

[Barrera 91] Barrera, J. S. A Fast Mach Network IPC Implementation. In *Proceedings of the Second USENIX Mach Symposium*, pages 1–11, Monterey, CA, November 1991.

[Bartlett 88] Bartlett, J. F. Compacting Garbage Collection with Ambiguous Roots. Technical Report WRL-TR-88-2, Digital Equipment Corporation Western Research Labs, February 1988.

[Berners-Lee et al. 94] Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., and Secretr, A. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.

[Bershad 93] Bershad, B. N. Practical Considerations for Non-Blocking Concurrent Objects. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 264–274, Pittsburgh, PA, May 1993.

[Bershad et al. 90] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[Bershad et al. 92a] Bershad, B. N., Draves, R. P., and Forin, A. Using Microbenchmarks to Evaluate System Performance. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 148–153, Key Biscayne, FL, April 1992.

[Bershad et al. 92b] Bershad, B. N., Redell, D. D., and Ellis, J. R. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 223–233, Boston, MA, October 1992.

[Black et al. 92] Black, D. L. et al. Microkernel Operating System Architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, Seattle, WA, April 1992.

[Bricker et al. 91] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and Rozier, M. A New Look at Microkernel-based UNIX Operating Systems: Lessons in Performance and Compatibility. In *Proceedings of the EurOpen Spring'91 Conference*, Tromsoe, Norway, May 1991.

[Brockschmidt 94] Brockschmidt, K. *Inside OLE 2*. Microsoft Press, 1994.

[Brustoloni & Bershad 93] Brustoloni, J. C. and Bershad, B. N. Simple Protocol Processing for High-Bandwidth Low-Latency Networking. Technical Report CMU-CS-93-132, Carnegie Mellon University, March 1993.

[Cao et al. 94] Cao, P., Felten, E. W., and Li, K. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–177, Monterey, CA, November 1994.

[Carter et al. 91] Carter, J. B., Bennett, J. K., and Zwaenepoel, W. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–64, Pacific Grove, CA, October 1991.

[Carter et al. 94] Carter, N. P., Keckler, S. W., and Dally, W. J. Hardware Support for Fast Capability-Based Addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 319–327, San Jose, CA, October 1994.

[Chankhunthod et al. 95] Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M., and Worrell, K. A Hierarchical Internet Object Cache. Technical Report CU-CS-766-95, DCS University of Colorado, July 1995.

[Chen & Bershad 93] Chen, J. B. and Bershad, B. N. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 120–133, Asheville, NC, December 1993.

[Cheriton & Duda 94] Cheriton, D. R. and Duda, K. J. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–194, Monterey, CA, November 1994.

[Cheriton & Zwaenepoel 83] Cheriton, D. R. and Zwaenepoel, W. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140, Bretton Woods, NH, October 1983.

[Colwell 85] Colwell, R. The Performance Effects of Functional Migration and Architectural Complexity in Object-Oriented Systems. Technical Report CMU-CS-85-159, Carnegie Mellon University, August 1985.

[Cooper & Draves 88] Cooper, E. C. and Draves, R. P. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[Cooper et al. 91] Cooper, E., Harper, R., and Lee, P. The Fox Project: Advanced Development of Systems Software. Technical Report CMU-CS-91-178, Carnegie Mellon University, August 1991.

[Davis et al. 93] Davis, P.-B., McNamee, D., Vaswani, R., and Lazowska, E. Adding Scheduler Activations to Mach 3.0. In *Proceedings of the Third USENIX Mach Symposium*, pages 119–136, Santa Fe, NM, April 1993.

[Dig 93] Digital Equipment Corporation. *DEC OSF/1 Writing Device Drivers: Advanced Topics*, 1993.

[Draves 93] Draves, R. The Case for Run-Time Replaceable Kernel Modules. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 160–164, Napa, CA, October 1993.

[Draves 94] Draves, R. P. Control Transfer in Operating System Kernels. Technical Report CMU-CS-94-142, Carnegie Mellon University, May 1994.

[Draves et al. 91] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, CA, October 1991.

[Engler & Kaashoek 95] Engler, D. and Kaashoek, M. F. Exterminate All Operating System Abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 78–83, Orcas Island, WA, May 1995.

[Engler & Proebsting 94] Engler, D. R. and Proebsting, T. A. DCG: An Efficient, Retargettable Dynamic Code Generation System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 263–272, San Jose, CA, October 1994.

[Engler et al. 94] Engler, D., Kaashoek, M. F., and O'Toole, J. The Operating System Kernel as a Secure Programmable Machine. In *Proceedings of the 1994 ACM European SIGOPS Workshop*, September 1994.

[Engler et al. 95] Engler, D. R., Kaashoek, M. F., and Jr, J. O. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[Fall & Pasquale 94] Fall, K. and Pasquale, J. Improving Continuous-Media Playback Performance with In-Kernel Data Paths. In *Proceedings of the First IEEE International Conference on Multimedia Computing and Systems*, pages 100–109, Boston, MA, May 1994.

[Felten 92] Felten, E. W. The Case for Application-Specific Communication Protocols. In *Intel Supercomputer Systems Technology Focus Conference*, pages 171–181, April 1992.

[Fiuczynski & Bershad 96] Fiuczynski, M. and Bershad, B. An Extensible Protocol Architecture for Application-Specific Networking. In *Proceedings of the 1996 Winter USENIX Conference*, San Diego, CA, January 1996.

[Forin et al. 91] Forin, A., Golub, D., and Bershad, B. N. An I/O System for Mach 3.0. In *Proceedings of the Second USENIX Mach Symposium*, pages 163–176, Monterey, CA, November 1991.

[Geschke et al. 77] Geschke, C., Morris, J., and Satterthwaite, E. Early Experiences with Mesa. *Communications of the ACM*, 20(8):540–553, August 1977.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the 1990 Summer USENIX Conference*, pages 87–95, June 1990.

[Hamilton & Kougiouris 93] Hamilton, G. and Kougiouris, P. The Spring Nucleus: A Microkernel for Objects. In *Proceedings of the 1993 Summer USENIX Conference*, pages 147–159, Cincinnati, OH, June 1993.

[Harty & Cheriton 91] Harty, K. and Cheriton, D. R. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 187–197, Santa Clara, CA, April 1991.

[Heidemann & Popek 94] Heidemann, J. and Popek, G. File-System Development with Stackable Layers. *Communications of the ACM*, 12(1):58–89, February 1994.

[Hildebrand 92] Hildebrand, D. An Architectural Overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[Hutchinson et al. 89] Hutchinson, N. C., Peterson, L., Abbott, M. B., and O'Malley, S. RPC in x-kernel: Evaluating New Design Techniques. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 91–101, Litchfield Park, AZ, December 1989.

[Int 81] Intel Corporation. *Introduction to the iAPX 432 Architecture*, 1981.

[Int 90] Intel Corporation. *i486 Microprocessor Programmer's Reference Manual*, 1990.

[Khalidi & Nelson 93] Khalidi, Y. A. and Nelson, M. An Implementation of UNIX on an Object-Oriented Operating System. In *Proceedings of the 1993 Winter USENIX Conference*, pages 469–480, San Diego, CA, January 1993.

[Lazowska et al. 81] Lazowska, E. D., Levy, H. M., Almes, G. T., Fischer, M., Fowler, R., and Vestal, S. The Architecture of the Eden System. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 148—159, December 1981.

[Lee et al. 94] Lee, C. H., Chen, M. C., and Chang, R. C. HiPEC: High Performance External Virtual Memory Caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–164, Monterey, CA, November 1994.

[Liedtke 92] Liedtke, J. Fast Thread Management and Communication Without Continuations. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 213–221, Seattle, WA, April 1992.

[Liedtke 93] Liedtke, J. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, December 1993.

[Lucco 94] Lucco, S. High-Performance Microkernel Systems. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 199, Monterey, CA, November 1994.

[Maeda & Bershad 93] Maeda, C. and Bershad, B. N. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, Asheville, NC, December 1993.

[Marsh et al. 91] Marsh, B., Scott, M., LeBlanc, T., and Markatos, E. First-Class User-Level Threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–121, Pacific Grove, CA, October 1991.

[McNamee & Armstrong 90] McNamee, D. and Armstrong, K. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Mach Symposium*, pages 17–29, Burlington, VT, October 1990.

[Mogul et al. 87] Mogul, J., Rashid, R., and Accetta, M. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987.

[Mossenbock 94] Mossenbock, H. Extensibility in the Oberon System. *Nordic Journal of Computing*, 1(1):77–93, February 1994.

[Mullender et al. 90] Mullender, S. J., Rossum, G. V., Tanenbaum, A. S., Renesse, R. V., and van Staveren, H. Amoeba – A Distributed Operating System for the 1990's. *IEEE Computer*, pages 44–54, May 1990.

[Nelson 91] Nelson, G., editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[Organick 73] Organick, E., editor. *Computer System Organization: The B5700/B6700 Series*. Academic Press, 1973.

[Pardyak & Bershad 94] Pardyak, P. and Bershad, B. A Group Structuring Mechanism for a Distributed Object Oriented Language Objects. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pages 312–219, Poznan, Poland, June 1994.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 31–39, Palo Alto, CA, April 1987.

[Redell 88] Redell, D. Experience with Topaz Teledebugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, October 1988.

[Redell et al. 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Romer et al. 94] Romer, T. H., Lee, D., and Bershad, B. N. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–266, Monterey, CA, November 1994.

[Romer et al. 95] Romer, T., Ohlrich, W., Karlin, A., and Bershad, B. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the Twenty-Third International Symposium on Computer Architecture*, pages 176–187, 1995.

[Rozier et al. 88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Giend, M., Guillemont, M., Herrmann, F., Leonard, P., Langlois, S., and Neuhauser, W. The Chorus Distributed Operating System. *Computing Systems*, 1(4):305–370, 1988.

[Schroeder & Burrows 90] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Schulman et al. 92] Schulman, A., Maxey, D., and Pietrek, M. *Undocumented Windows*. Addison-Wesley, 1992.

[Small & Seltzer 94] Small, C. and Seltzer, M. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.

[Stevenson & Julin 95] Stevenson, J. M. and Julin, D. P. Mach-US: Unix On Generic OS Object Servers. In *Proceedings of the 1995 Winter USENIX Conference*, New Orleans, LA, January 1995.

[Stodolsky et al. 93] Stodolsky, D., Bershad, B. N., and Chen, B. Fast Interrupt Priority Management for Operating System Kernels. In *Proceedings of the Second USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 105–110, San Diego, CA, September 1993.

[Stonebraker 81] Stonebraker, M. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: a Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Thekkath & Levy 93] Thekkath, C. A. and Levy, H. M. Limits to Low-Latency RPC. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[Thekkath & Levy 94] Thekkath, C. A. and Levy, H. M. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 145–156, San Jose, CA, October 1994.

[von Eicken et al. 92] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[Wahbe et al. 93] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.

[Waldspurger & Weihl 94] Waldspurger, C. A. and Weihl, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, November 1994.

[Wheeler & Bershad 92] Wheeler, B. and Bershad, B. N. Consistency Management for Virtually Indexed Caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 124–136, Boston, MA, October 1992.

[Wulf et al. 81] Wulf, W. A., Levin, R., and Harbison, S. P. *Hydra/C.mmp: An Experimental Computer System*. McGraw–Hill, 1981.

[Young et al. 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, Austin, TX, November 1987.

[Yuhara et al. 94] Yuhara, M., Bershad, B. N., Maeda, C., and Moss, J. E. B. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, pages 153–165, San Francisco, CA, January 1994.