Abstract

Application domains, such as multimedia, databases, and parallel computing, require operating system services with high performance and high functionality. Existing operating systems provide fixed interfaces and implementations to system services and resources. This makes them inappropriate for applications whose resource demands and usage patterns are poorly matched by the services provided. The SPIN operating system enables system services to be defined in an application-specific fashion through an extensible microkernel. It offers fine-grained control over a machine’s logical and physical resources to applications through run-time adaptation of the system to application requirements.

1 Introduction

This white paper describes an operating system called SPIN that will address the requirements of the coming generation of resource-intensive applications. In SPIN, these requirements are achieved through the use of application-specific services. An application-specific service is one that precisely satisfies the functional and performance requirements of an application or class of applications.

The key to application-specific services is an adaptable kernel that enables system resources to be efficiently and safely managed by the application. By efficient, we mean that capable applications execute more quickly and with less programming complexity than when using a more conventional platform, such as Ultrix or Mach. By safe, we mean that multiple applications may run at the same time, yet be protected from one another through hardware and software firewalls.

SPIN supports adaptability through an extensible microkernel that safely executes application-specific code either in the kernel or at user-level. In SPIN, an application specifies a service as an implementation partitioned into three components: an application-level component, which is linked into the application’s address space, a kernel-level component, which provides fast, specialized access to in-kernel services, and a user-level server component, which manages long-lived service state. The way in which the service is partitioned is determined by its safety, sharing, and performance requirements.

SPIN is structured around an extensible microkernel architecture. The microkernel exports interfaces that offer applications fine-grained control over a few fundamental system abstractions, such as processors, memory, and I/O. SPIN is extensible in that application programs and servers can install program sequences called spindles that execute in the kernel in response to hardware and software events, such as processor exceptions and context switches. Spindles
enable applications to define customized kernel interfaces and implementations with which application-specific services can be built.

1.1 Motivation

The next decade will bring a radical change to the way we do computing as applications that were at one time considered “niche services” such as large distributed databases, high-quality multimedia, and programs for massively parallel systems, become common. Although the application demands are changing substantially, the operating systems base on which those applications run is not. Consequently, application performance is frequently limited by today’s operating systems, which provide an inadequate interface to computer system services.

The key problem facing operating systems is how to support efficiently a range of applications with widely varying service demands. For example, current virtual memory page-replacement policies are based on application mixes from the 1970’s and early 1980’s [Babaoglu & Joy 81] that have good reference locality. These policies, though, are poorly suited for newer applications, such as information retrieval and multimedia, where page access patterns are seemingly random, or strictly sequential and temporally constrained [Kearns & DeFazio 89]. Similarly, current file system implementations assume that most files are accessed sequentially [Ousterhout et al. 85]. However, important applications such as information retrieval have file access patterns that are quite non-sequential [Gray & Reuter 92]. As a result, many database systems manage in-core disk caches manually because existing operating systems do such a poor job of meeting their needs [Stonebraker 81]. We believe that other performance-critical applications will follow the same route because no current operating system allows system resources to be efficiently and safely managed through tailored interfaces and implementations.

The resource demands of current and future applications can be met by an operating system architecture in which services can be implemented on an application-specific basis. As a consequence, a service can be defined long after the operating system has been written, compiled, and shipped as product. The service, which can come bundled with the application, relies on low-level interfaces exported by the operating system kernel. These interfaces, which themselves can be tailored, enable the service to allocate and manage system resources such as CPUs, disks, networks, and memory.

Contemporary commercial and research operating systems provide interfaces that are inadequate for use by application-specific services. By “inadequate,” we mean that one of the following statements can be made about the operating system:

- there are no interfaces through which application-specific services can exercise direct control over the logical and physical resources, or
- some interfaces for resource management exist, but they are clumsy, or inefficient, or both, or
- all applications have unconstrained access to resources, providing good performance when programs are well-behaved, but poor system stability when they are not.

In the first case, applications must suffer with whatever interfaces and abstractions are provided by the operating system. In the second case, the “right” abstraction can be realized, but at an intolerable performance cost. Finally, in the third case, any abstraction can be realized for a single program, but isolation between programs is not possible. In all three cases, a mismatch between the interfaces exported by the operating system kernel and those required by an application-specific service make such services infeasible.

1.2 Adaptability in SPIN

Our goal in building SPIN is to provide applications with an adaptable kernel platform on top of which application-specific services can be built. The ideas underlying SPIN stem from research over the last several years that has addressed some of the fundamental performance problems that arise in modern operating system services, including interprocess communication, synchronization, thread management, networking, virtual memory, and cache management [Draves et al. 91, Bershad et al. 92, Stodolsky et al. 93, Bershad 93, Yuhara et al. 94, Maeda & Bershad 93,
In each case, the interfaces exported by a service were poorly matched to the needs of important applications. The solution to the performance problem came from enabling applications to adapt the behavior (interface and implementation) of system services to realize maximum performance. Each change, though, required careful and deliberate modifications of the operating system kernel.

In SPIN, adaptability is achieved with an extensible microkernel that allows an application to specify a service as an implementation partitioned into an application component, an application-specific kernel component, and possibly a user-level server component. The microkernel provides lightweight and portable abstractions of the physical hardware such as threads and virtual address spaces which are used by the higher-level services. By allowing applications to participate in the implementation of high-level services, we permit applications to make informed decisions about their resource requirements. By placing the implementation within an application component (application-level library), or a kernel-level code sequence, the service can be accessed with low latency.

The application-specific kernel components are called spindles (SPIN Dynamically Loaded Extensions), and enable applications to define the precise interface and implementation for kernel services that they require. Spindles enable a service to be partitioned across the user/kernel boundary in the most efficient manner that still satisfies its safety and sharing requirements. Specifically, installing code at the kernel level allows for flexible and rapid response to system hardware and software events. For example, an application program can install a code sequence that runs each time a thread from that program’s address space is preempted in response to an interrupt, a time-slice event, or a higher-priority thread. In the first two cases, the program can ensure system-wide or application-wide invariants about preemptability. In the third case, the application can enforce constraints that deny priority inversion. Although the code sequences execute in kernel-mode, their safety is verified by a trusted compiler.

1.3 Alternatives

Microkernel technology has been promoted as a solution to many of the adaptability requirements of demanding applications [Accetta et al. 86], and in the past few years there has been dramatic growth in the number and quality of microkernels [Phelan et al. 93, Hildebrand 92, Rozier et al. 88]. Current practice is to structure a microkernel-based operating system as one or more server address spaces that collectively implement operating system services [Golub et al. 90, Julin et al. 91, Rozier et al. 88, Khalidi & Nelson 93, Hildebrand 92]. However, it is often as difficult to modify a service in another address space as it is to modify one placed in the kernel, diminishing many of the flexibility advantages that favor microkernel architectures. In addition, the communication overhead incurred when contacting servers can result in poor performance [Maeda & Bershad 92, Maeda & Bershad 93]. These facts make it quite difficult to tailor an operating system service to the requirements of a particular resource intensive application using conventional microkernel technology.

1.4 The rest of this paper

The rest of this paper is structured as follows. In Section 2 we describe the SPIN architecture. In Section 3 we discuss the system’s extension language and compiler. In Section 4 we show how applications in several domains are facilitated by SPIN. In Section 5 we describe related work. In Section 6 we discuss the system’s status and directions.
2  

**SPIN: A system for application-specific services**

In this section we discuss the overall system composition of *SPIN* and present a concrete example of its use in structuring a service. Later, in Section 4 we broaden our discussion to show how *SPIN* can be applied to increase efficiency across a range of demanding application domains.

### 2.1 Specialization

An operating system kernel offers two general functions: it provides abstractions of the system’s physical and logical resources, and it implements a set of management policies for those resources. In the *SPIN* microkernel, the two functions are split. Low-level resource controllers provide lightweight and portable abstractions of the physical hardware, such as threads and virtual address spaces. They define interfaces providing access to a machine’s physical and logical resources, including a set of global resource allocation interfaces that allow applications to allocate and deallocate system resources while guaranteeing integrity and progress in cases of high load. The controllers themselves though do not contain any management policy. Policies are provided either by in-kernel application-specific services or by default kernel services. For applications that do not require application-specific management policies, *SPIN* is an ordinary microkernel with a set of well-defined general purpose services. For applications with special needs, *SPIN* provides a set of interfaces to low-level resources that may be combined in an arbitrary way to achieve the required level of efficiency.

All management policies are defined by *embedded implementations* called spindles. A spindle is a code sequence that is installed dynamically into the operating system kernel by or on behalf of an application. Spindles run in response to a particular system event, such as a system call, an exception, or a context switch. They can also be activated by user code or by events generated within other spindles.

All interaction between an application (or, more likely, an application-level library) and the operating system is through spindles. The spindle interfaces allow applications to manipulate at a fine-grain level the resources granted by the kernel controllers. Spindles enable applications to define new system calls through a composition of internal kernel interfaces. They also enable applications to discover changes in the state of the hardware and the operating system, thereby enabling them to react to changes in resource allocations and demands with low-latency and high-efficiency. In effect, the application can *specialize* the operating system kernel to provide the type of service or management policy required, without paying the overhead of crossing several interface and protection boundaries at each service invocation.

### 2.2 Components

The *SPIN* operating system consists of a set of low-level resource controllers with associated interfaces, a set of pre-defined spindles offering default system functions, and machinery to install and run spindles. Three kernel mechanisms enable these components: one for associating spindles with particular specific events in the kernel, one for executing, at kernel-level, those spindles associated with an event when that event occurs, and one that verifies the integrity of spindles when they are installed into the kernel.

Figure 1 illustrates the basic structure of *SPIN*. Although the system’s structure is similar to that of a traditional microkernel, providing interfaces to services such as threads, address spaces, and memory, the similarity is only superficial. In a conventional microkernel, the highest-level system services are implemented as a set of layered abstractions, with the highest layer exporting a few basic interfaces. Applications built to the microkernel interface are consequently constrained to using the services at the highest layer. While these services may be composed to provide a rich set of higher-level abstractions (for example, a UNIX process under Mach is defined through a composition of an address space, some memory, and a thread), the low-level behavior of each abstraction is essentially fixed. In contrast using *SPIN*, high-level interfaces required by applications and application-specific services are defined in terms of the lower-level interfaces available to spindles. A program defines its own interface to the kernel through a spindle that executes in the kernel. In turn, that spindle has access to a large set of kernel interfaces. For example, a program can construct a spindle that provides an interface for creating and then starting a new thread of control. The spindle itself uses more primitive interfaces (create thread, allocate stack, set initial thread state, start thread) to accomplish this.
Figure 1: This figure illustrates the basic structure of services on the SPIN microkernel. Services are split into external servers that maintain global service information, application-specific libraries that provide fast access paths to services, and spindles that allow low latency access to kernel resources. In the figure, each service is depicted as having these three components. A trusted compiler and code verifier ensures that spindles are unable to compromise the integrity of the system. (Different stipple patterns denote different services.)

At a basic level, a spindle can be used to define the implementation of a traditional system call, serving as a wrapper around underlying kernel services. At a more sophisticated level, a spindle can enable an application to monitor and react to changes in global resource allocation without involving costly transfers to user-level code. With this structure, high-level operating system services, such as filing, networking, virtual memory management, and fine-grained thread management, can be efficiently implemented as part of each application’s address-space instead of as part of the kernel or a dedicated server.

2.3 Service partitioning

Microkernel-based systems reflect a tension between modularity and efficiency. Services are moved out of the kernel to user-space to achieve higher modularity, but additional overhead for communication is incurred whenever the services are accessed. This overhead can be greatly reduced in SPIN by allowing parts of a service to be located in the address spaces that use them. Figure 2 shows the differences between the decomposition of services on a microkernel and under SPIN and their influence on communication overhead in a system. There are three cases illustrated.

**case I**: Placing parts of a server within an application in the form of an application-level library allows application requests to be served without crossing any protection boundaries. This optimizes system performance for common operations that do not need kernel interaction or access to data in other domains.

**case II**: Operations that require tight integration with the kernel can be installed as a spindle, and thus may access internal kernel interfaces without incurring the overhead of protection boundary crossings.
**case III**: Installing server code at the kernel level allows for flexible and rapid response to system hardware and software events without costly upcalls to a server.

Conventional microkernels with fixed interfaces can use the first optimization, but the other two require new capabilities that are provided by the spindle mechanism.

![Comparison of a traditional microkernel server with a partitioned service in SPIN.](image)

**Figure 2:** *Comparison of a traditional microkernel server with a partitioned service in SPIN.*

### 2.4 An example

As an example of how components of SPIN fit together, consider the structure of a user-level virtual memory manager [McNamee & Armstrong 90, Harty & Cheriton 92]. A user-level virtual memory manager enables an application to control the set and contents of physical page frames that are currently backing a given piece of virtual memory. An application can request, say, 100 pages of physical memory from the system’s physical page manager. Those pages are granted in bulk, and the application creates spindles that rely on fine-grained kernel interfaces to the physical mapping layer to control access and to shuffle pages between disk and memory. By defining a spindle to handle page faults, the pager can inexpensively be notified of changes to a page’s access patterns. For example, the pager can implement LRU-clock with simulated reference bits by defining a spindle that sets a bit in the application’s address space on a reference to a page by defining a spindle associated with the page-write trap that sets a bit in the applications. The pager can even define logical pages that are smaller than the machine’s physical pages, detecting writes, for example, to sub-pages in order to collect fine-grained reference information [Hosking & Moss 93].
3 Spindle language and compiler issues

An operating system interface is much like a programming language in that it defines a primitive set of operations available to the programmer [Lampson 84]. In SPIN, the operating system interface is in fact an actual programming language through which applications can define and install new interfaces that match their requirements. The language and its implementation have the following goals:

- **Expressiveness.** There should be few limitations to the kinds of interface extensions that applications can install into the kernel.

- **Safety.** Spindles installed by applications must be checked to ensure that in-kernel interfaces are respected (otherwise system integrity can be lost) and that spindles installed by other applications are not adversely affected.

- **Performance.** The quality and efficiency of spindles must be no worse than that of the systems “native” code, otherwise service providers will have an incentive to modify the kernel through more intrusive mechanisms. Additionally, the process of installing a spindle should not be prohibitively expensive.

- **Scalability.** Since each application may install dozens, or even hundreds of spindles, system performance should not degrade with the number of spindles.

In the following subsections, we describe the characteristics of an extension language, checker, and compiler that satisfy these requirements.

3.1 The extension language

A spindle is similar to a module: it contains one or more procedures sharing local state that can persist across spindle invocations. A spindle interacts with the rest of the kernel by invoking operations on a set of abstract data types that define the spindle’s interface to the kernel. These abstract data types grant a spindle access to low-level parts of the kernel-level data structures, such as the process table or the TLB, as well as access to memory in the application’s address space.

Spindles are written in a type-safe, object-based programming language. Type safety helps guarantee that spindles do not violate the integrity of the kernel. Object-based languages support clearly-specified interfaces to the abstract data types that model kernel resources available to spindles. Some spindles are designated as entry points that are directly callable from the application; other procedures are invoked by the kernel in response to system events. A spindle registers interest in a kernel event and is invoked when the event occurs.

3.2 The extension checker

Spindles execute in kernel mode and are granted access to some low-level kernel routines and data structures. SPIN verifies that these spindles do not compromise system integrity. The kernel, along with a trusted service outside the kernel, verifies at installation time that a spindle does not violate any of its restrictions, and fulfills its obligations (if any). Since spindles are written in a type-safe language with a well-defined interface to kernel services, traditional type checking can ensure that spindles only invoke legal operations on the abstract data types defined in the spindle/kernel interface. Additionally, kernel operations exported to spindles can be guarded with a predicate expression that must be true for access to be legal. The spindle compiler ensures that these predicates are satisfied through a combination of static analysis and dynamic checks.

3.3 The extension compiler

Spindles are compiled into an executing system at run-time. We rely on aggressive compiler technology to ensure that the SPIN microkernel extended with user-defined spindles performs as well as a non-extensible monolithic operating system with services built-in to the kernel. Good performance can be achieved with well-understood optimizing
compiler technology, such as intraprocedural data flow analysis, symbolic evaluation, and inline expansion. These techniques can eliminate much of the overhead of the extension language: the compiler can inline-expand calls in spindles to kernel operations, replacing them with direct data structure accesses or even constants, and the compiler can attempt to evaluate predicate expressions guarding kernel operations in the context of the spindle code preceding the call. With this technology, spindles can be installed and executed quickly.

Advanced compilation technology, such as partial evaluation [Jones et al. 89, Consel 90, Weise et al. 91, Jones et al. 93] can blend together multiple spindle routines and the surrounding kernel code to reduce the overheads of maintaining large numbers of spindles. It can also reduce the cost of crossing from the kernel’s execution environment to the spindle’s. Partial evaluation is a program transformation technique that specializes program code with respect to some of its argument values. In our context, for example, if several spindles are associated with the same kernel event, the compiler can specialize the event dispatcher to produce a single code sequence tuned just for the spindles installed at that time.

4 **SPIN** and application domains

Application-specific techniques enabled by **SPIN**’s structure are relevant to application domains that have high performance requirements. Examples include general purpose high-performance computing, multicomputer-based multiprocessing, shared memory multiprocessing, multimedia, and databases/information retrieval. The performance of traditional operating system services for these problems has been poor, either for the demanding application, or all other applications running at the same time. In this section, we describe the operating system requirements of these domains and discuss the role of application-specific solutions. We then describe a number of key techniques enabled by **SPIN** that satisfy these requirements.

**General purpose high-performance computing**

Many operating system services such as synchronization and scheduling, virtual memory and interprocess communication are generally important for any application requiring high performance. For example, any program that uses threads internally as a program structuring device [Hauser et al. 93] can benefit from fast synchronization [Bershad et al. 92]. Any program that interacts with an operating system server can benefit from fast interprocess communication [Bershad et al. 90, Draves et al. 91]. Many applications have become sensitive to the degree to which their memory access patterns are satisfied by an architecture’s fast memory system (cache and translation lookaside buffer) [Chen & Bershad 93]. For example, some compilers now use static blocking algorithms based on the cache size in order to maximize the cache hit rate during well-formed data intensive computations [Lam et al. 91]. Programs with irregular structure must rely on more dynamic information, for example, cache, TLB, or page fault rates. These miss rates may be easy to determine, but potentially expensive to communicate to the application. Application-specific code in the kernel that can track a program’s memory system behavior and provide guidance and feedback to the runtime can result in improved program performance.

**Parallel processing**

Application-specific communication protocols [Felten 92], scheduling, and virtual memory management can improve the performance of parallel programs running on a distributed memory multicomputer. Fast communication is required to transmit messages from one processor to another. Appropriate scheduling and synchronization support can ensure that all threads in a multicomputer program run at the same time to avoid unnecessary stalls due to scheduling anomalies [Zahorjan & McCann 90, Ousterhout 84]. Application-specific virtual memory services can ensure that unanticipated page faults do not delay processors involved in a cooperative computation, thereby delaying other processors.

Shared memory multiprocessing applications require fine-grained scheduling control, lightweight threads [Anderson et al. 92], synchronization [Anderson et al. 89], and information about memory system behavior. Multiprocessor applications can rely on application-specific thread schedulers, or **user-level thread management packages**, for high performance in the presence of relatively fine-grain parallelism [Anderson et al. 89]. Implementing user-level threads on top of
existing operating system mechanisms (such as kernel threads) can be difficult. In previous work, we addressed these difficulties with a mechanism called *scheduler activations*. Scheduler activations rely on the operating system kernel to convey information about kernel-level scheduling events to applications [Anderson et al. 92].

**Multimedia**

Multimedia applications, such as video-on-demand, video-conferencing, virtual reality, and interactive learning, impose special demands on the scheduling, communication and memory allocation policies of an operating system. In general, real-time systems implement a simple fixed priority ordering [Hildebrand 92]. Conventional operating systems, however, support either a single scheduling policy to arbitrate among competing activities, or multiple policies that promote fairness but favor interactive activities [Black 90]. Some systems provide multiple scheduling policies but only from among a few fixed policies set at kernel-build time [Tokuda et al. 90]. Flexible, application-specific scheduling, though, has been shown to provide critical performance benefits for both time-constrained and non-real-time activities [Anderson 93, Anderson et al. 92].

In terms of memory resources, multimedia applications use large amounts of data (audio and video streams) with access patterns that interact poorly with locality-based page replacement algorithms [Anderson 93, Nakajima et al. 92]. Application-specific virtual memory management policies can solve this problem. High-level information about media direction, edit cuts, and temporal constraints are directly relevant to page replacement decisions. When presenting a video stream, for example, an application can sequentially prefetch video frames directly from disk into memory-resident buffers. Information about synchronization between media streams can also be specified to prevent unnecessary replacement of pages that are interdependent.

**Databases and information retrieval**

Databases and information retrieval applications impose severe demands on the filing and memory services of an operating system. The speed of disks and memory, which are at the base of any file service, have not kept pace with processor speeds; any leverage that can be applied to increase their performance is critically important to end-application performance.

Filesystem performance can benefit from application-specific information in several ways. The application can provide hints about future usage to the filesystem to help it schedule disk traffic [Gibson et al. 92]. This can result in more effective prefetching policies and lower buffer cache miss rates. An effective prefetching policy can also remove virtual memory remapping operations from the critical path, since disk blocks are already mapped into the application address space when they are needed. In addition, the application can inform the kernel about how it will use the buffer cache, so that the kernel can make informed decisions about physical memory allocation [Stonebraker 81].

### 4.1 Some application-specific services enabled by *SPIN*

The application domains described in the previous subsection can be enabled by operating system services that are customized to the program’s needs. Below, we detail some specific techniques.

**Extensible interprocess communication**

An extensible IPC interface enables applications and servers to define their own semantics for interprocess communication enabling the best tradeoff between performance and functionality. To receive a message, an application installs a spindle that can recognize a message destined for it as the receiver. To send a message, an application formats the message for the spindle, traps to the kernel, and presents the kernel with a block of data intended for a particular address space, which is represented in the kernel by a spindle waiting to receive a message. Upon executing the spindle, the kernel delivers data to the corresponding receiver. The responsibility for interpreting the contents of the message belongs with the receiver.
Application-level protocol processing

IPC is an example of a more general style of interaction in which the applications implement a sophisticated communication protocol such as TCP/IP entirely within application-level libraries. Spindles implement low-level data packet dispatching mechanisms that couple remote communication with application-level protocol processing.

Fast, simple communication

For many multicomputer applications, the per-message processing required by an application is substantially less than that needed to transmit the message reliably [Felten 93]. Low-latency message passing spindles that run at interrupt-level can substantially improve parallel system performance. For example, active messages [von Eicken et al. 92] are simple interrupt handlers that can be written as spindles, allowing them to run safely in a general purpose computer system where integrity is as important as performance.

Application-specific file systems and buffer cache management

An application-specific buffer cache manager for a file system can be implemented as a library that allows applications to access resident data with a simple procedure call but without data copying. A library-based buffer cache manager can provide for application-specific buffer management policies to ensure a high cache hit rate. Spindles can monitor page access patterns, and notify applications of changes in the current availability of the virtual memory pages used to contain buffer cache pages. A long-lived server can act as a caretaker for the buffer cache, guarding its contents as processes start and terminate, and ensuring consistency across multiple readers and writers.

User-level scheduling

In SPIN, a complete user-level scheduler can be implemented with a per-program spindle that emulates scheduler activations. The spindle, executed on every kernel-level thread context switch, sends a message to the thread’s address space that reflects the change in scheduling state. A user-level library, in turn, implements application-specific thread management primitives.

Synchronization

Synchronization mechanisms coordinate the activity of multiple threads of control that share memory. Synchronization mechanisms that assume no contention for shared resources can have lower overhead than pessimistic ones that assume that contention will occur [Massalin & Pu 89, Stodolsky et al. 93]. In the general case, these optimistic strategies require some form of kernel support to ensure correctness in the presence of an oblivious kernel scheduler [Bershad et al. 92, Bershad 93, Alemany & Felten 92]. SPIN will make arbitrary synchronization strategies possible by reflecting scheduling decisions up to applications as they occur, or by providing an in-kernel rollback mechanism. An application that requires lightweight synchronization can install a spindle that executes on every thread preemption, ensuring the correctness of optimistic synchronization mechanisms.

Real-time scheduling policies

Spindles make it possible to implement a framework that allows applications to implement their own scheduling policies associated with low-level context switch and I/O events.

Application-specific virtual memory

Application-specific virtual memory can be implemented by providing an application with physical memory pages. The application can create a spindle that provides access to low-level mapping operations, and another one that reflects page faults up to an application-specific library.
Runtime systems with memory system feedback

Spindles enable low-level performance information to be inexpensively reflected back to applications. For example, a runtime system can install a spindle that decrements a counter each time an application takes a TLB miss within a particular range of virtual addresses. When the counter reaches zero, the spindle can notify the application, enabling it to restructure its virtual memory usage to reduce the load on the TLB.

4.2 Summary

Table 1 characterizes the relationship between application domains and some application-specific services that are enabled by SPIN. The table shows the extent to which SPIN’s structure facilitates high-performance applications in each of the domains.

<table>
<thead>
<tr>
<th>technique</th>
<th>service and/or application</th>
<th>general purpose computing</th>
<th>multicomputing</th>
<th>multiprocessing</th>
<th>multimedia</th>
<th>database</th>
</tr>
</thead>
<tbody>
<tr>
<td>extensible IPC</td>
<td></td>
<td>✓</td>
<td>✓/✓</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓/✓/✓</td>
</tr>
<tr>
<td>application level protocol</td>
<td></td>
<td>✓</td>
<td>✓/✓</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓/✓/✓</td>
</tr>
<tr>
<td>processing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fast simple network communication</td>
<td></td>
<td>✓/✓</td>
<td>✓/✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>application specific file systems</td>
<td></td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>✓/✓/✓</td>
</tr>
<tr>
<td>synchronization</td>
<td></td>
<td>✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>application controlled virtual memory</td>
<td></td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>✓/✓/✓</td>
</tr>
<tr>
<td>real-time scheduling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓/✓</td>
<td></td>
</tr>
<tr>
<td>scheduler activations</td>
<td></td>
<td>✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memory system feedback</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
</tr>
</tbody>
</table>

Table 1: This table illustrates the applicability of different techniques enabled by spindles, and their importance for different application classes. One tick denotes some applicability, two ticks denote significant improvements in performance, and three ticks denote critical improvements.

5 Related work

5.1 Extensibility and the operating system

Extensibility has been the “holy grail” of operating systems design since “THE”, one of the first modular operating systems [Dijkstra 66]. An inflexible module structure and poor performance proved to be a substantial drawback of these systems. Early personal computer operating systems [Redell et al. 80], which ran all system services and applications in a single address space, enabled applications to have good performance while being tightly coupled with the operating system. However, these systems offered no protection against rogue or buggy applications, making them inappropriate for multiuser environments.
Extensible services

Previous research into extensible system services has addressed file systems [Rees et al. 86, Bershad & Pinkerton 88], scheduling [Anderson et al. 92], communication [Bershad et al. 91], and user-level memory management [Krueger et al. 93, McNamee & Armstrong 90, Harty & Cheriton 92, Sechrest & Park 91]. No system has provided an efficient way to compose multiple resources in a coherent manner. For example, with previous systems, in order for an application to cooperate with the kernel in making fine-grained CPU and memory allocation decisions, control must be transferred from the kernel to the application and back one or more times each time the kernel changes the global resource allocation. This context switching overhead can put a high lower bound on the allocation granularity possible in the system. In contrast to this approach, SPIN provides a single framework in which extensible services can be built.

Packet filters

The packet filter offers an example of kernel extensibility [Mogul et al. 87]. A packet filter is run against each incoming network packet to demultiplex data packets to higher level protocol software. With careful design, the packet filter is able to support protocol processing for a large number of applications [Yuhara et al. 94]. SPIN generalizes on the notion of the packet filter, enabling richer, more complex services to be safely installed into the kernel.

Dynamic linking

In systems such as Spring [Khalidi & Nelson 93], Chorus [Rozier et al. 88], and OSF/1 the kernel can be modified at run time with a new set of interface implementations for heavyweight services like device drivers or the Unix file system. Pure object code is downloaded “on-the-fly” from user-level into the kernel, exposing the kernel to protection violations. In contrast, with SPIN, extensibility is at the interface level (new kernel interfaces can be defined by applications), fine-grained (particular events within particular applications and threads), and safe (the extensions are validated both dynamically and statically).

Synthesis

The Synthesis system [Massalin & Pu 89] improved performance through the use of highly specialized and dynamically constructed interface implementations. For instance, a file open operation in Synthesis would return a handle to a piece of code optimized for accessing the opened file. However, both the interfaces and the scope of their implementation were limited to that which was pre-defined by the Synthesis kernel itself. This differs from SPIN, where applications are able to define both the interface to, and implementation of, system services. Hence, a SPIN application could create a file open interface whose implementation is optimized not only for the particular file, but also for the access patterns of the application.

5.2 Language and compiler work

The Fox Project

The Fox project [Cooper et al. 91] applies advanced compiler technology to system software development. The primary focus of the research is on the extensions necessary to use the Standard ML programming language [Milner et al. 89] in support of systems programming. Standard ML is a type-safe programming language with a rich module system that enables many of types of extensions that are available in SPIN. The Fox project has focused on improving the performance of the Standard ML compiler in the context of a standalone network service, and has not developed a general operating system structure.

Reflective systems

Several systems have used reflection to create adaptable systems. A reflective system is one that includes mechanisms to monitor and modify its own behavior as it executes. In the Apertos operating system [Yokote et al. 91], for example, users customize the system’s behavior by choosing among several reflective mechanisms for kernel services. Kiczales
et al. have studied general meta-object protocols, which are interfaces to languages and systems that enable users to customize and extend the system’s behavior [Kiczales et al. 91]. Although SPIN does not support general reflection, it does provide a controlled mechanism by which services can augment the kernel with their own specialized code sequences (spindles). Previous reflective systems have suffered high overhead from the extra layers of abstraction. SPIN relies on dynamic compilation and partial evaluation to ensure good performance.

Compiler optimization techniques

Runtime compilation of code has been explored in several experimental systems, ranging from a Smalltalk environment [Deutsch & Schiffman 84] to the implementation of bit-transfer operations in raster graphics systems [Pike et al. 85] to debugging [Kessler 90]. Runtime code generation and optimization has also been used to produce speedups in traditional applications [Keppel et al. 93]. The Self and Cecil systems [Ungar & Smith 87, Chambers & Ungar 91, Chambers 93, Chambers 92] include automatic mechanisms for determining where optimizations can be cost effective, and may choose not to optimize in cases where performance will not be improved. SPIN will rely on similar heuristics. There has been little work in applying partial evaluation techniques at runtime, as required by SPIN. Most partial evaluation techniques are oriented towards static analyses. Moreover, existing techniques for partial evaluation do not address the problem of increased code size, which occurs when generating specialized instances of code.

6 Status and directions

We are developing SPIN in the context of the Mach 3.0 microkernel and the OSF/1 Unix server running on DEC Alpha workstations. We are partitioning the system statically into a SPIN component and a native (OSF/1) component. Existing OSF/1 binaries will continue to run by accessing the OSF/1 services that manage the native-component. SPIN will manage the SPIN component across applications that have been explicitly marked to run within SPIN. This approach allows us to slowly migrate away from a mixed-mode system to one that runs SPIN natively. With this, we will provide a set of OSF/1 services using the SPIN primitives.

The advantages of the approaches taken in SPIN are not restricted to microkernel-based systems. Any system that provides a core set of services behind a fixed interface is subject to inadequate performance when faced with the “wrong” application. The flexible structures and solutions provided in SPIN are therefore also appropriate for a monolithic system.

We intend to use SPIN both as a research target, enabling us to explore resource management mechanisms as we construct the system, and as a research vehicle, enabling us to explore resource management policies, as we use the system. SPIN will support applications in traditional domains, such as UNIX-style workstation computing, and newer domains, such as multimedia and multiprocessing. While we intend to use SPIN at the University of Washington as a self-hosting system, and to make it available in its pristine form to other universities and industrial sites, we expect that additional value will come with the transfer of a few key mechanisms and interfaces to commercial systems, such as OSF/1 and Windows-NT. These systems, as their application base grows, will be required to provide an application programming interface that facilitates fine-grained resource control.

References


