A version of this paper appeared in the Proceedings of the Second Symposium on Operating Systems Design and Implementation

Dynamic Binding for an Extensible System

Przemysław Pardyak and Brian N. Bershad
{pardy,bershad}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195, USA

Abstract

An extensible system requires a means to dynamically bind extensions into executing code. The *SPIN* extensible operating system uses an event-based invocation mechanism to provide this functionality in a flexible, transparent, safe, and efficient way. Events offer a uniform model of extensibility, whereby the system's configuration can change without changing any of its components. Events are defined with the granularity and syntax of procedures but provide extended procedure call semantics such as conditional execution, multicast, and asynchrony. By installing a handler on an event, an extension's code can execute in response to activities at the granularity of procedure call. Our system uses run-time code generation to ensure that event delivery has low overhead and scales well with the number of handlers. This paper describes the design, use and performance of events in the *SPIN* operating system.

1 Introduction

SPIN is an extensible operating system that allows applications to define customized services to improve their performance, correctness or simplicity [Bershad et al. 95]. Applications dynamically add code to an executing system to provide new services, and they replace or augment old code to change existing ones. For example, an application may provide a new in-kernel file system, replace an existing paging policy, or add compression to network protocols. The system relies on a combination of compile-time, link-time, and run-time facilities to deliver good performance while providing the safety properties generally expected of a modern operating system.

The key to the system's architecture is a mechanism that integrates applications' extensions with the system. *SPIN* uses *events* to provide such a binding mechanism. In an event-based system, components announce some system occurrence by explicitly *raising* an event of a particular name. Other parties, interested in learning of the occurrence, register *event handlers* which execute in response to a raised event. Events are generally recognized as an effective technique for implementing loosely-coupled, flexible systems in which relationships between code components must be dynamically established [Reiss 90, Sullivan & Notkin 92].

SPIN's use of events as its integration mechanism is novel in that interaction between components is entirely event-based down to the level of procedure call. Every procedure name in the system potentially represents an event, every procedure invocation corresponds to the raising of an event, and every procedure executing corresponds to the handling of an event. Consequently, all relationships between code components in the system are subject to change simply by changing the set of handlers associated with any given event.

The system provides an *event dispatcher* that oversees event-based communication. The dispatcher is responsible for enabling services such as conditional execution, multicast, asynchrony, and access control. In order to achieve good performance, the dispatcher is bypassed entirely to eliminate overhead in the common case of a single handler per event, and relies on runtime code generation to offer low-overhead performance that scales well as more handlers are installed for a given event.

This research was sponsored by the Advanced Research Projects Agency, and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship.

1.1 Motivation

The kernel of the *SPIN* operating system defines only a few low-level services, such as device access, dynamic linking, and events – the subject of this paper. All other services, such as user-space threads and virtual memory, are provided as extensions which are dynamically bound into the kernel as needed by applications. Consequently, the system's extensibility mechanism (events) must serve a broad range of purposes, including transparent interposition [Jones 93], multicast [Heidemann & Popek 94], filtering [Ritchie 84], and conditional execution [Mogul et al. 87]. Moreover, the system must be responsive to rapid and frequent changes in configuration, since extensions may come and go with the frequency of individual applications.

Our goal in defining *SPIN*'s extension services has been to provide the system and extension programmer with a set of interfaces that enable the system to be changed easily, transparently, safely, and efficiently. By *ease*, we mean that the system should lend itself to programmers making small changes with little programming overhead. By *transparent*, we mean that global effects should be achievable through local modifications. By *safe*, we mean that restrictions must exist to ensure that neither accidental nor malicious influences damage the system's integrity. Finally, by *efficient*, we mean that the runtime cost of potential and actual modifications to the system's structure should be small enough to ensure that programmers not be motivated to seek out alternative techniques for implementing change.

1.2 Related work

All software systems are extensible in one way or another, but it is the extension model and its implementation that determine the applicability and effectiveness of extensibility. These two properties are jointly influenced by the set of services that the model implies, and the performance characteristics of those services. A rich set of services enables a variety of interaction styles between the kernel and extensions, and between extensions themselves. Good performance enables that interaction to be freely applied without concern for execution overhead, while poor performance inhibits new interactions. For example, cross-address space interprocess communication has been commonly used in many systems to extend a system's functionality [Jones 93, Patience 93, Black et al. 92, Vahdat et al. 94]. Unfortunately, the high cost of protecting system code from extension code has generally limited the utility of this approach to coarse grain system interfaces. Some systems have relied on fast kernel upcalls to create specialized system services with lower kernel overhead [Thekkath & Levy 94, Engler et al. 95]. However, this approach does not lend itself well to extending shared services since all communication must be channeled through the upcall mechanism.

Some systems have used dynamic linking to address the performance problems of cross-address space IPC [Orr et al. 93, Banerji & Cohn 94]. Dynamic linking tightly couples clients to a particular service implementation and does not provide for transparent routing of requests to alternate or supplementary services. As a result, dynamic linking does not allow multiple independent extensions to provide different parts of a service without explicit cooperation.

Events [Reiss 90, Sullivan & Notkin 92] have been used in several systems including network operating systems [Bhatti & Schlichting 95], windowing systems such as X11, software engineering environments [Cagan 90], extensible applications such as Emacs, database servers [Sybase 96], and the Macintosh operating system. Events support flexible composition by allowing an extension to be executed in response to interactions between components of the system. For example, in database systems, users can provide procedures that execute in response to certain database modification events, or *triggers* [Sybase 96]. Under windowing systems and the Macintosh operating system, events are used primarily to offer programmers a rudimentary concurrent programming interface, without forcing them to use threads. Unfortunately, traditional event systems have usually required a unique invocation syntax and protection model, and have not been as efficient as systems based on procedure call. As a result, events have been limited to specific services of an operating system and have not been applied to provide a system-wide extensibility mechanism.

Programming languages provide a number of approaches to extensibility that have been successfully applied in operating systems. For example, modularity [Parnas 72] is used in Oberon [Mossenbock 94], object-orientation [Goldberg & Robson 83] in Spring [Hamilton & Kougiouris 93], and reflection [Maes 87] in Apertos [Yokote et al. 89]. These approaches enable extensibility through the language mechanisms (for example inheritance) to modify language defined components (objects). However, programming languages have not typically provided support for multicast, transparent interpositioning, filtering, conditional execution, or access control, which are necessary services in an extensible operating system.

1.3 The rest of this paper

In the rest of this paper, we describe the design, implementation, and performance of events within the *SPIN* operating system. In Section 2 we present the design of the system's extensibility mechanisms. In Section 3 we describe the implementation and performance of the event system. Finally, in Section 4 we conclude. Although this paper draws upon our experience with an extensible operating system, its ideas and mechanisms may be applicable to any software that admits dynamic restructuring [Sybase 96, Sun 95, Brockschmidt 94].

2 Events in *SPIN*

The *SPIN* kernel and its extensions are written in Modula-3, a modular, type-safe, ALGOL-like language. Extensions are incorporated into the system through a two-step process. First, the extension's code is dynamically linked into the operating system kernel [Sirer et al. 96]. The dynamic linker resolves all outstanding unresolved references in the extension code against a collection of interfaces explicitly exported by the system. This first phase is sufficient to allow the extension code to call into exported services, but does not provide a means for pre-existing code to interact with the extension code. That occurs during the second phase in which the extension registers handlers with events. The events are defined in interfaces against which the extension has been linked. Once installed, other extensions may link against the extension's exported interfaces and install handlers on the events it exports.

2.1 Defining events

Programs in Modula-3 are written as a composition of *modules* that communicate by procedure calls through *interfaces*. A procedure call can be viewed as one module's signal that another should execute some code. We exploit the basic relationship between procedure callers and implementations to define events. We treat a procedure call as an event raised by the caller and handled by one or more implementations. Because every procedure is implicitly an event, it is eligible for extension.

Events are described as Modula-3 procedure signatures, which has several implications. First, it preserves the "feel" of the language. Second, an event is a typed name, so it is protected from misuse and improper disclosure by the type system. For events declared in an interface, the event can be raised and in some cases handled by the modules that import

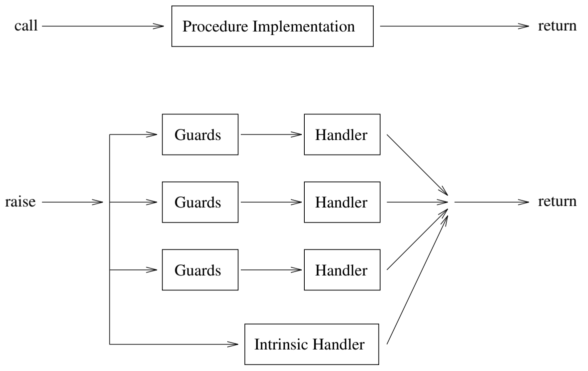

Figure 1: *Procedure call vs. event invocation. A procedure call is directed to a single implementation. In contrast, an event is conditionally dispatched to any one of a number of implementations. The intrinsic handler is the procedure that has the same name as the event.*

that interface. An event not declared in an interface can be explicitly passed between modules as a typed procedure pointer. Finally, an event can be raised with arguments and return a value, allowing events to be parameterized and generate results.

A *handler* is a procedure that executes in response to an event. Handlers are dynamically added to or removed from an event, and any number of them can be installed at any particular time. The same handler can be installed many times on many events, and is invoked independently for each of the installations. Optionally, a handler may be installed with a *closure*, which is an opaque data reference. When the handler executes in response to an event, the closure specified at installation time is passed to the handler. This allows the same handler to be associated with multiple events.

The procedure having the same name as the event itself is its *intrinsic* handler, and it is invoked whenever the event is raised (unless the handler has been explicitly deregistered). A typical model for changing the implementation of a single procedure within a module is to deregister the intrinsic handler and then register an alternate one.

Each event handler can be associated with a set of *guard* predicates that filter out unwanted handler invocations. For example, an extension that is interested in handling page fault events for its data segment can define a guard that checks whether the faulting address is in that segment. The guards are specified outside of the handler so that the same handler can be installed multiple times with different guards, and so that additional guards can be added to further restrict when the handler can run.

Figure 1 illustrates the difference between conventional procedure invocation and event handling. A procedure call is an invocation of the procedure's implementation through the procedure's name. An event raise is a conditional invocation of a collection of handlers installed on the event through the event's name. Of course, an event with only an intrinsic handler is identical (in semantics and implementation) to a procedure call.

```
(* Interface to trap handling *)
INTERFACE MachineTrap;

(* declaration of the MachineTrap.Syscall event *)
PROCEDURE Syscall(strand: Strand.T;
                  VAR ms: MachineCPU.SavedState);

END MachineTrap.
```

```
(* Implementation of the Mach extension *)
MODULE MachEmulator;

(* the syscall routine that
   handles Mach system calls *)
PROCEDURE Syscall(strand: Strand.T;
                  VAR ms: MachineCPU.SavedState) =
  BEGIN
    CASE ms.v0 OF
    ...
    | -65 => (* vm_allocate *)
      VMHandlers.vm_allocate(strand, ms);
    ...
    END;
  END Syscall;

FUNCTIONAL
PROCEDURE SyscallGuard(
             strand: Strand.T;
             VAR ms: MachineCPU.SavedState)
             : BOOLEAN =
BEGIN
   RETURN IsMachTask(strand);
END SyscallGuard;

(* Initialization *)
BEGIN
  (* installation of the syscall handler *)
  Dispatcher.InstallHandler(
      MachineTrap.Syscall,      (* event *)
      SyscallGuard,             (* guard *)
      Syscall);               (* handler *)

END MachEmulator;
```

Figure 2: *The Mach extension's system call routine installed as a handler on an event that announces a system call. The keyword "FUNCTIONAL" is described shortly.*

2.2 An example

The implementation of system calls in *SPIN* demonstrates the use of events. The kernel provides no native system call handling facilities. Instead, the MachineTrap module, which implements basic trap handling, exports an event Syscall through the MachineTrap interface. The event takes two arguments, the thread (an object of type Strand.T) in whose context the system call was made, and the saved state of that thread. When a system call trap happens, the machine dependent part of the kernel saves the state of the trapping thread, changes the state of the system to allow safe execution in the kernel context, and raises the MachineTrap.Syscall event. The dispatcher then evaluates the guards for each handler registered for this event, invoking the handlers for which the guards evaluate to true.

An extension providing its own system call service handles the MachineTrap.Syscall event as shown in Figure 2. This example, which shows an extension that emulates the Mach system call interface, illustrates several aspects of event handling. First, the event itself, Syscall is defined as a typed procedure within the MachineTrap interface. Second, the extension code provides a handler for that event by registering a properly typed procedure through the dispatcher interface. The registration specifies the event name, any guard to be used, and the handler. In the example, the guard ensures that only system calls raised for threads executing as part of Mach tasks are handled by Syscall.[1]

2.3 Evaluating guards

The dispatcher determines which handlers should be invoked in response to a raised event by evaluating their guards. Guards must be strictly functional, free of any side-effects. Since they describe predicates on the system's state, including any parameters passed to the guard, this restriction does not impose any significant computational restriction. If the processing of an event requires some side-effecting computation, then it may be encoded directly within the handler, and not within the guard. Lack of side effects is important because it prevents a guard from modifying or saving its arguments. This allows the dispatcher to reorder or short-circuit guard execution entirely in order to improve performance. In addition, it prevents arguments from being "leaked" out and stored away during an event invocation.

[1]This example is somewhat contrived, since it shows the Syscall module limiting its own access to Mach system calls. In reality, additional guards are imposed to restrict visibility to events. We describe this facility in a later section.

A side-effect free procedure is declared as FUNCTIONAL within the source, which is verified by the compiler, and carried forward to execution time in the type information for the procedure.

Ordering handlers

Multiple handlers installed on the same event should be executed in an order that respects their dependencies. The dispatcher provides a deterministic mechanism for ordering handlers to avoid the need for explicit synchronization. Specifically, a set of ordering constraints can be associated with a handler. A handler installed with the First or Last constraint will be placed, respectively, at the beginning or the end of the handler list at the time it is installed. A handler can also be ordered relative to other handlers using a Before or After constraint. In addition, the dispatcher allows the ordering constraints associated with a given handler to be queried and dynamically changed.

Passing arguments

The dispatcher is responsible for passing arguments to guards and handlers. Generally, handlers and guards receive the arguments specified when an event is raised. If a handler or guard is installed with a closure, the closure is passed as an additional argument.

Handlers can be used as filters, wherein a single event can have multiple handlers, and handlers can modify the arguments seen by subsequent handlers. For example, an extension can provide the MS-DOS file name space over a UNIX file system by transparently converting file names from one standard to the other. Filters require that a handler be able to manipulate its arguments; consequently handlers explicitly installed as *filters* may specify some parameters as call-by-reference, rather than call-by-value. The filter can change these arguments and the dispatcher passes the new values to handlers and guards ordered after the filter.

Handling results

All events, even those that return values, may have zero, one, or many handlers. If only one handler fires, then the value returned from that handler is passed on to the event raiser, mimicking procedure call. In case no handler runs, a runtime exception is thrown at the point the event is raised, enabling the event raiser to recover. Alternatively, an event may have installed on it a *default handler*, which executes only when no other handler fires. In case several handlers run, it is necessary to determine which, if any, of the results should be returned in response to the event raise. This is performed by a *result handler*, which is called separately for each result; it ultimately determines the final result of the invocation. For example, the system defines a VM.PageFault event, which is raised on any page fault. Its return value is a boolean indicating whether the page is accessible. If the page is inaccessible, the VM system crashes the application. The default handler for this event relies on a trusted default paging service provided by VM. The result handler for this event returns the logical-or of all the handler results. Previous structuring approaches have addressed the problem of handling multiple results in a similar fashion [Pardyak & Bershad 94, Cooper 85].

2.4 Safety issues

The fundamental property of the system's event model is that every procedure call can potentially be dispatched to any one of a number of handlers. In practice, though, this property is overly lax, as it would enable any extension to potentially undermine the behavior of any interface within the system. In the rest of this section, we describe the protection mechanisms in place which ensure that *SPIN*, as an extensible system, can be managed in a controlled fashion. First, though, we informally describe the typechecking rules governing event and handler specification.

Typechecking

During installation, the dispatcher checks the types of handlers and guards to guarantee the type safety of event interactions. Several rules govern the type checking of handler installation. The handler's argument types and return value must be the same as that of the event's. For the guard, argument types must be the same as the event's, but the return value must be a boolean (indicating whether or not the guard evaluates true). A procedure installed as a handler with a closure must take an additional first argument of some reference type, and the type of the associated closure must be a subtype of that reference type. Although the arguments to the event and handler must be the same, a handler acting as a filter is allowed to specify some of its parameters as "by reference" rather than "by value." To preserve the initial value of the parameter from the standpoint of the event raiser, the dispatcher first makes a copy of the arguments and passes to the filter a reference to the copy.

2.5 Access control

We rely on a set of static and dynamic access control mechanisms to ensure that events are not misused by extensions. There are several aspects to the issue of access control with respect to events, such as: who is allowed to raise or handle an event, which handlers are allowed to handle a particular occurrence of an event, and who can manipulate an event's default or result handlers.

Fundamentally, our approach is to assert that some set of functions have *authority* over an event, and that those functions may directly affect the way in which that event can be used by the system. The authority may in turn, rely on notions of identity, context, or passwords to further implement more specific notions of access control.

An event's authority could be defined in many different ways. We selected a strategy that was easy and efficient to implement, and could be easily described to programmers. The authority for an event is the module that defines the event's intrinsic handler. Recall that the intrinsic handler for an event is that procedure having the same name as the event. For example, if some module M implements a procedure M.P, then M (rather, all code within M) is the authority over event operations on M.P. This approach allows the programmer to locate control over an event with the code responsible for defining the event in the first place. In addition, authority can be easily checked using type information maintained by the kernel runtime.

An event's authority specifies an authorization procedure for operations that concern the event. The dispatcher calls back into the authorization procedure every time the set of handlers and guards associated with the event is manipulated. An authorizer is passed information describing the operation that is being requested, some context information describing the requestor, and, optionally, an opaque reference passed in by the requestor that can be used to bootstrap a richer authorization protocol such as one based on passwords. The authorizer evaluates the request and then allows or prevents the operation.

So that authority can be demonstrated, we added a new type to the language runtime that describes compilation units (interfaces and modules), as well as operations for obtaining instances of those types [Hsieh et al. 96]. The operations guarantee that the identity of a module can be obtained only inside of that module. Upon receipt of a module descriptor and an event, the dispatcher checks that the described module in fact defines the intrinsic handler for the event, and, if so, considers the caller an authority.

The simplest form of authorization concerns dynamic linking. When a module requests that it be dynamically linked against some other module, that module's authorizer is consulted and the linkage is permitted or denied. Denial prevents the requester from accessing any of the symbols, and hence events, exported by any of the modules governed by the authorizer. This strategy is sufficient for preventing modules from inappropriately raising events defined by other modules.

Authorization also occurs whenever a module attempts to install a handler for an event. The dispatcher, upon the installer's request, first contacts the authorizer to ensure that the requestor and the handler are legitimate. At this point, the authorizer has several options. It can deny the request, returning an error to the installer. Or, it can allow the request, and possibly apply some execution property, such as ordering constraints, onto the handler to ensure that previously installed handlers continue to operate as expected. Lastly, the authorizer can *impose* an additional set of guards on the handler. These guards act as those described in the previous section – they must evaluate to true in order for the handler to execute.

Imposed guards limit access to an event by dynamically filtering them before they are delivered to a handler. Any number of guards can be imposed on a handler, and they can be added and removed dynamically. Figure 3 demonstrates the use of authorities and imposed guards for the MachineTrap.Syscall event described earlier. The main initialization routine of the MachineTrap module, as an authority over the event, provides the dispatcher with a procedure that will be called whenever any other module attempts to install a handler for the Syscall event. On installation, the dispatcher calls AuthorizeSyscall, which in in turn imposes a guard on the event ensuring that only system calls for the installing thread's address space will be seen by the new handler. Any system call that occurs for threads not executing as part of the installing thread's address space will not be seen by the handler being installed.

The example shows that the policies and granularity for event filtering are under the control of an event's authority. Obviously, different authorities may impose different restrictions, depending on the kind of service that is being offered through the event. For example, our networking code imposes

```
MODULE MachineTrap;
...

(* the authorizer for the MachineTrap.Syscall event *)
PROCEDURE AuthorizeSyscall (
            binding : Dispatcher.Binding)
            : BOOLEAN =
  BEGIN
    (* impose a guard to ensure that the extension only
       fires for the currently address space context *)
    Dispatcher.ImposeGuard(binding,
                          ImposedSyscallGuard,
                          GetCurrentAddressSpace());
    RETURN TRUE;
  END AuthorizeSyscall;


(* imposed guard for handlers
   installed on the MachineTrap.Syscall event.
   validSpace is the address space for which our
         associated handler has access;
   strand is the thread context that invoked the
        system call;
   ms is the machine state at the time of
   the call.  *)

PROCEDURE ImposedSyscallGuard (
            validSpace : REFANY;
            strand : Strand.T;
            VAR ms : MachineCPU.SavedState)
            : BOOLEAN =
  BEGIN
    RETURN Space(strand) = validSpace;
  END ImposedSyscallGuard;


(* Initialization.
   We define an authorizer (AuthorizeSyscall)
   over the Syscall event. We present a
   descriptor (THIS_MODULE()) which, at runtime,
   can be checked to ensure that MachineTrap.Syscall
   is intrinsically defined within this module. *)
BEGIN
  (* install the authorizer by demonstrating
     authority *)
  Dispatcher.InstallAuthorizerForEvent(
      AuthorizeSyscall,
      MachineTrap.Syscall,
      THIS_MODULE());
END MachineTrap.
```

Figure 3: *The authorization code that protects the event announcing a system call (MachineTrap.Syscall). The validSpace argument passed into the imposed guard is a closure that was provided as the final argument to Dispatcher.ImposeGuard within AuthorizeSyscall.*

a guard that restricts an application's extension to receive packets only when the packets' destination is for a port that had been previously assigned to the application [Fiuczynski & Bershad 96].

2.6 Denial of service

Access control mechanisms ensure that only approved handlers execute in response to published events, but do not address the problem of service denial by those handlers. Specifically, we must be sensitive to the problem of resource hoarding, which occurs whenever an extension acquires "too much" of some system resource as a consequence of using events.

Runaway handlers

A handler that never returns can prevent the event raiser and other handlers from making progress. This can occur when a module raises an event which is being handled by another module that can not be trusted to return in a timely fashion. In terms of a view on a layered system, this happens whenever control "flows up." The kernel is itself preemptible, so a runaway handler only prevents the event raiser from making progress; other threads in the system continue to run.

We offer two solutions to the problem of runaway handlers; one preventative, but expensive, and the other corrective, but cheap. The expensive solution is to run asynchronously. An asynchronous event or handler detaches the execution of an event raiser from the handler. Either the event itself, or a specific handler, can be specified as *asynchronous*. If the event is asynchronous, then all handlers execute in a separate thread, and the event raiser proceeds without blocking. At times it may be advantageous to execute only some of the invoked handlers asynchronously, so the dispatcher allows any handler to be invoked asynchronously. For example, if a file system is extended with lazy replication, the original code should perform the write synchronously, but the replication can be done asynchronously.

Asynchronous execution incurs the additional expense of spawning off a new thread of control for each raised event, but is sufficient for procedures that are part of slow operations, and for which replies are not necessary. For example, our in-kernel UNIX server uses asynchronous events to implement a per-application system call tracer, and our virtual memory system uses asynchronous events for page-in requests.

The asynchrony of either the event or the handler is specified during handler installation. *SPIN* does not allow the use of asynchrony where it would violate the semantics of a procedure call. Since no immediate result is returned from an asynchronous event raise, an attempt to raise an event asynchronously that returns a result will raise an exception unless a default handler is installed. Because asynchronous threads execute on different stacks, arguments can not be passed by reference; they may be incidentally destroyed before they go out of scope. Therefore, it is illegal to define as asynchronous an event that takes an argument by reference, or to install an asynchronous handler on such an event.

When performance is critical, runaway handlers can be avoided by a corrective measure: termination. Handlers that execute beyond a certain time period (specified by the event's authority) are terminated. Of course, early termination introduces a variety of safety problems with regard to the type system and program correctness. For example, if a handler is terminated in the middle of allocating some memory, or updating a critical data structure, then the entire system could be left in an unsafe state.

We address the interaction between early termination and safety by restricting the installation of procedures that might be terminated to those that explicitly invite termination. Handlers willing to be terminated must be explicitly declared as EPHEMERAL within the source [Hsieh et al. 96]. The expectation is that an EPHEMERAL handler returns quickly, but, if not, will be terminated. An authorizer can determine whether or not a particular handler is in fact EPHEMERAL, and refuse installation if it is not. Program correctness can not be affected by the termination of an EPHEMERAL handler because, by definition, such termination is correct. Moreover, termination is localized to affect only EPHEMERAL handlers; the compiler ensures that EPHEMERAL procedures may only interact with other such procedures. Consequently, any subsystem can protect itself from early termination simply by exporting no EPHEMERAL handlers.

We tend to use EPHEMERAL handlers where synchronous interaction is required between low levels of the kernel and simple extension code. For example, our network interrupt handlers communicate incoming packets to extensions through handlers marked as EPHEMERAL. A terminated handler in this case simply causes a packet to be lost. As another example, extensions that manage user-space threads rely on EPHEMERAL handlers to save and restore thread state during context switches. Premature termination results in the termination of the user-space thread, which is followed by a termination of the user-space task itself.

Too many handlers

In theory, there can be no limit to the number of handlers or guards associated with any given event. In practice, though, events having more than one handler or guard consume some amount of kernel memory. Consequently, an extension could exhaust the system's memory by installing a large number of handlers on an event. Presently, *SPIN* denies additional installations when memory is low, relying on individual authorizers to locally enforce restrictions. We do not consider this to be a good long term solution, though, and realize that it is an instance of the more general problem of resource management given a shared pool. We are currently experimenting with different strategies for accounting and resource reclamation and hope to report on these in a future paper.

2.7 Summary

SPIN's event services provide a few mechanisms that allow the program to codify their intents about component relationships. These relationships are defined through explicit requests on a dispatcher service, and are exercised using the syntax of procedure invocation. Although events allow any relationship to be defined dynamically, most are in fact static, and most procedures are used as procedures (no guard, single handler), and not as events. Where more flexible compositions are required, though, the event mechanism we have described allows them to be forged "from a distance." Lastly, the event facilities include access control and protection services to guard against their misuse and abuse. It must be clear, though, that a certain degree of vigilance and paranoia is required when designing the components of an extensible system.

3 Implementation and performance

In this section we describe the implementation and performance of the event services discussed in this paper. The *SPIN* operating system and its extensions run on DEC Alpha Workstations and Intel X86 Personal Computers[2]. The system is about two and a half years old, and has currently about 100 different extensions of various sizes and complexity, in-

[2] We present performance numbers only from the more mature Alpha version.

cluding six different file systems, two different operating system emulators, the Internet protocol suite, and a collection of integrated applications, including a distributed transaction system and a web server. Events are used to implement a number of critical system services, including system calls and emulation, threads and scheduling, networking, sockets, and tty services. With our Unix emulator running, the current version of the system installs handlers on 57 different events, with an average of 3.5 handlers per event.

In order to achieve good performance for events, the dispatcher is tightly coupled with the Modula-3 runtime. An event with only an intrinsic handler is dispatched directly as a procedure call (which, in fact, it is). For richer events, the dispatcher relies on direct access to the compiler-generated information to hide the event mechanisms behind the Modula-3 procedure syntax and semantics. For example, the dispatcher uses runtime type information to type-check procedures when they are installed as handlers and guards. It directly manipulates runtime data-structures such as module and interface descriptors, and method dispatch tables to ensure that events can be dispatched with the overhead of procedure call.

For events having guards, multiple handlers, or other extended semantics, the dispatcher replaces the implementation of the intrinsic procedure with one that performs the appropriate dispatch. The dispatcher maintains a separate list of handlers for each event. When a handler is added to or removed from an event, the list is regenerated and used for subsequent event dispatches. The dispatch itself is performed by a routine that iterates over the handler list and invokes handlers according to the specified event properties. A handler can be installed and removed from an event dynamically and without disrupting on-going interactions. The handler lists are updated atomically with respect to event dispatch by using a single memory access to replace the old list with the new one.

We use run-time code generation to build a specialized and optimized version of the dispatch routine. Our approach is similar to that taken in previous systems which have used run-time code generation for late-bound kernel code [Massalin & Pu 89]. We specialize the code to the number of arguments in each event, and unroll the dispatch loop to transform handler invocations from indirect procedure calls through a list of handlers to direct procedure calls. We also inline the code of small guards and handlers directly into the dispatch routine. Finally, we use peephole optimizations to improve the quality of the generated code.

The dispatcher itself consists of about 3400 lines of Modula-3 code, and the run-time code generator and optimizer take another 7600 lines of Modula-3, C, and Alpha assembler code. The rest of the kernel is substantially larger, and includes the Modula-3 runtime libraries (30000 lines), device drivers and debugging support for the family of DEC Alpha workstations (340000 lines)[3], and other core kernel services including dynamic linking, scheduling, and memory management (29000 lines).

3.1 Performance

We report on several aspects of event performance. Our measurements are intended to demonstrate that events have low overhead, and that the overhead scales well with the number of handlers installed on a particular event. Lastly, we show the impact of event management on overall application performance. We collected our measurements on DEC Alpha 133MHz AXP 3000/400 workstation, which is rated at 74 SPECint 92. We ran our experiments on *SPIN* V1.26 of September 1996.

Latency

Table 1 shows the latency of synchronous event handling as a function of the number of guards and handlers installed on an event. Guards compare a global variable to a constant and return true, and handlers return without performing any work. We measured latency in the case where guards and handlers execute out of line, and in the case where the dispatcher inlines them. In practice, inlining occurs only when there are relatively few handlers, and when they are relatively small. The table shows that dispatch overhead is primarily a function of the number of procedure calls that occur during each raised event. The simple case, where an event has only an intrinsic handler is implemented as a straight Modula-3 procedure call from the raiser to the handler, and performs as shown in the table. For a small number of handlers it is the same order of magnitude as a procedure call. When additional guards or handlers are installed, overhead grows linearly with the number of guards and handlers, with each incurring an overhead of roughly one procedure call. Asynchronous events, which have not been optimized, introduce an additional latency of between 38 and 90 $\mu secs$ per event raised. The additional time is spent creating the asynchronous thread. In terms of its impact on

[3] We borrow all of this code directly from the manufacturer's source tree.

Number of arguments	Modula 3 procedure call (intrinsic)	Number of event handlers							
		1		5		10		50	
		no inline	inline	no inline	inline	no inline	inline	no inline	inline
0	0.10	0.37	0.23	1.18	0.41	2.15	0.63	11.69	2.48
1	0.13	0.39	0.24	1.25	0.45	2.32	0.72	11.51	2.87
5	0.14	0.97	0.42	1.61	1.55	2.88	1.32	14.45	5.65

Table 1: *The overhead of event dispatching on Alpha workstations. Most events with non-intrinsic handlers in* SPIN *have between one and ten handlers. Times for both inlined and non-inlined handlers and guards are shown. All times are in* μsecs.

basic system services (microbenchmarks), we have measured event processing overhead to be on the order to 10–15% for operations such as system call and thread management.

Installation overhead

Each time a new handler is installed for an event, the dispatcher regenerates the data structures and code associated with that event. Consequently, the overhead to install n handlers is $O(n^2)$, although in practice this quadratic is not a serious problem. Most events have few handlers on them, and even with many handlers, overhead is tolerable. The time to install a single handler is about 150 $\mu secs$, whereas to install 100 handlers on the same event takes about 30 milliseconds. We've not yet made any effort to reduce this overhead. Intrinsic handlers – that is, most procedures in the system – are defined without any runtime overhead. Nevertheless, we may ultimately, need to rely on a more incremental (and economical) approach to installation.

3.2 End-to-end effects

We ran two more experiments to better understand the end-to-end effects of event management on overall performance. In the first, we show how networking latencies are affected by flipping packets back and forth between two machines. In the second, we calculate a breakdown of where time is spent in the system when displaying a Postscript image using X11 on *SPIN*.

Networking

SPIN's networking support is implemented in terms of events and guards[Fiuczynski & Bershad 96]. Guards are used to filter packets from the network by discriminating on fields in the protocol header (e.g., guards may discriminate on the UDP or TCP port destination field). The round-trip latency of a protocol reflects the overhead induced by the protocol as it transfers bytes from sender to receiver and back to the sender. Table 2 shows the round-trip latency for small (8 byte) UDP/IP messages between applications running on a pair of AXP 3000/400 machines running *SPIN* and connected by a 10Mb/s Ethernet. To highlight the overhead of event handling, we install additional guards on the Udp.PacketArrived event that evaluate to false. That is, the experiment has one active endpoint and many inactive ones, yet all guards are evaluated for each packet that arrives from the network.

1 guard	5 guards	10 guards	50 guards
475	481	487	530

Table 2: *Network UDP roundtrip time as a function of the number of guards installed on a packet event. Only one guard evaluates to true. Times are in* μsecs.

The measurements show that each additional guard introduces an overhead of about one μsec to the round-trip latency. Note that we do not presently reorder guard evaluation in order to accommodate packet trains, as is done by many packet filter implementations. In addition, we presently do not optimize the guard decision tree, which would be effective for the port comparison required by this example. We are currently working on a strategy by which this type of guard optimization can be easily expressed.

Application performance

In order to assess the impact of event handling on overall performance, we measured the *SPIN* kernel running Digital's X11 server while displaying a Postscript version of this paper. The Postscript was being processed on another machine running ghostview, an X11 Postscript previewer, and the page images were sent over the network to the *SPIN*

Event name	raised	time	handlers	guards
Ether.PacketArrived	2536	0.03	4	3
Ip.PacketArrived	2529	0.02	6	5
Udp.PacketArrived	24	0.00	6	5
Tcp.PacketArrived	2505	0.01	2	1
OsfNet.DelTcpPortHandler	3	0.00	1	0
OsfNet.AddTcpPortHandler	3	0.00	1	0
MachineTrap.Syscall	3976	0.03	3	2
Strand.Run	7936	0.03	4	3
Events.EventNotify	595	0.00	2	2

Table 3: *Major events raised while previewing a document. Most of the events are related to protocol processing. Strand.Run occurs during each scheduling operation, and Event.EventNotify is raised by our implementation of the Unix* `select` *system call. The "raised," "time," "handlers," and "guards" columns refer to the number of times a given event was raised, the total time in seconds spent handling the event, and the number of handlers and guards installed on the event.*

machine.[4] This setup stresses several kernel extensions including our Digital UNIX emulator supporting X11 and our TCP/IP networking stack [Fiuczynski & Bershad 96]. We instrumented the kernel and extension code to generate call graph information with counts and elapsed times. Table 3 reports on the kind and number of the most commonly occurring events during the system's execution.

The total time required to preview the document was 23.5 seconds. Much of the time (12.52 seconds) was spent idling, 4.2 seconds were spent processing in the X11 server, and 6.8 seconds were spent executing in the kernel. Of that kernel time, 0.12 seconds were spent raising and dispatching events, or roughly .5% of total execution time, 1% of total CPU time, and 1.7% of total kernel time. These numbers are pessimistic because of dilation effects due to profiling, which tend to exaggerate the impact of fast routines (such as handlers and guards) on overall performance.

4 Conclusions

We have described the design and implementation of an event mechanism for an extensible operating system. In order to achieve good performance, we have designed the event mechanism "close" to the programming language, allowing us to leverage off of fast default paths, while still providing richer composition semantics where necessary. Our performance measurements demonstrate that event processing overhead is small, both in an absolute sense, where it is close to procedure call latency, and in an relative sense, where it represents only a small fraction of service and overall execution time.

More information on the *SPIN* project is available at http://www-spin.cs.washington.edu, an Alpha workstation running *SPIN* with a WEB server extension. Our home page also includes pointers to pages that document the dispatcher interfaces described in this paper.

[4] *SPIN*'s UNIX emulator still has trouble with a few UNIX programs. Ghostview is one of them.

Acknowledgments

The work described in this paper is part of a large operating system effort at the University of Washington. None of what we have described here could have been done without the input, assistance, and feedback of all the other members of the *SPIN* group, especially including David Becker, Marc Fiuczynski, Charlie Garrett, and Yasushi Saito. In particular, we owe special thanks to Stefan Savage and Emin Gün Sirer who contributed to the initial design of the event system. Wilson Hsieh helped design and implement many of the language extensions used by the dispatcher. Jan Sanislo implemented the system's support services that enable us to run X11 on top of *SPIN*. Dylan McNamee, Gene Morgan, Sandy Morgan and Stefan Savage all helped with earlier drafts of this paper. Finally, we owe special thanks to John Wilkes, our shepherd, for his guidance and patience.

References

[Banerji & Cohn 94] A. Banerji and D. L. Cohn. Protected Shared Libraries. Technical Report 37, University of Notre Dame, 1994.

[Bershad et al. 95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 1995.

[Bhatti & Schlichting 95] N. T. Bhatti and R. D. Schlichting. A System For Constucting Configurable High-Level Protocols. In *Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols*, August 1995.

[Black et al. 92] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. 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.

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

[Cagan 90] M. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, June 1990.

[Cooper 85] E. C. Cooper. Replicated Distributed Programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 314–324, December 1985.

[Engler et al. 95] D. R. Engler, M. F. Kaashoek, and J. James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, December 1995.

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

[Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language And Its Implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

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

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

[Hsieh et al. 96] W. Hsieh, M. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B. Bershad. Language Support for Extensible Systems. In *Proceedings of the First Workshop on Compiler Support for Systems Software*, pages 127–133, February 1996.

[Jones 93] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Call Interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93, Asheville, NC, December 1993.

[Maes 87] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications*, pages 147–155, December 1987. Published as ACM SIGPLAN Notices, volume 22, number 12.

[Massalin & Pu 89] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 191–201, Litchfield Park, AZ, December 1989.

[Mogul et al. 87] J. Mogul, R. Rashid, and M. Accetta. 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] H. Mossenbock. Extensibility in the Oberon System. *Nordic Journal of Computing*, 1(1):77–93, February 1994.

[Orr et al. 93] D. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg. Fast and Flexible Shared Libraries. In *Proceedings of the 1993 Winter USENIX Conference*, June 1993.

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

[Parnas 72] D. L. Parnas. On the Criteria to Be Used in Decompositing Systems into Modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[Patience 93] S. Patience. Redirecting Systems Calls in Mach 3.0, An Alternative to the Emulator. In *Proceedings of the Third USENIX Mach Symposium*, pages 57–73, Santa Fe, NM, April 1993.

[Reiss 90] S. P. Reiss. Connecting Tools Using Message Passing in the Field Envirnoment. *IEEE Software*, 7(4):57–66, July 1990.

[Ritchie 84] D. M. Ritchie. A Stream Input-Output System. *Bell Labs Technical Journal*, 63(8, Part 2):1897–1910, October 1984.

[Sirer et al. 96] E. Sirer, M. Fiuczynski, P. Pardyak, and B. Bershad. Safe Dynamic Linking in an Extensible Operating System. In *Proceedings of the First Workshop on Compiler Support for Systems Software*, pages 141–148, February 1996.

[Sullivan & Notkin 92] K. Sullivan and D. Notkin. Reconciling Environment Integration and Software Evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[Sun 95] Sun Microsystems. *The Java^{TM} Language Specification*, 1.0 beta edition, October 1995.

[Sybase 96] Sybase. Sybase SQL Server 11. Technical report, Sybase, Inc., August 1996. http://www.sybase.com/products/system11/sqlsrv11.html.

[Thekkath & Levy 94] C. A. Thekkath and H. M. Levy. 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.

[Vahdat et al. 94] A. Vahdat, P. Ghormley, and T. Anderson. Efficient, Portable and Robust Extension of Operating System Functionality. Technical Report UCB CS-94-842, University of California, Berkeley, December 1994.

[Yokote et al. 89] Y. Yokote, F. Teraoka, and M. Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In S. Cook, editor, *Proceedings of the 1989 European Conference on Object Oriented Programming*, pages 89–106, Nottingham, July 1989.