Dealing With Disaster: Surviving Misbehaved Kernel Extensions

Margo I. Seltzer, Yasuhiro Endo, Christopher Small, Keith A. Smith
Harvard University

Dealing With Disaster: Surviving Misbehaved Kernel Extensions

Margo I. Seltzer, Yasuhiro Endo, Christopher Small, Keith A. Smith
Harvard University

Abstract

Today's extensible operating systems allow applications to modify kernel behavior by providing mechanisms for application code to run in the kernel address space. The advantage of this approach is that it provides improved application flexibility and performance; the disadvantage is that buggy or malicious code can jeopardize the integrity of the kernel. It has been demonstrated that it is feasible to use safe languages, software fault isolation, or virtual memory protection to safeguard the main kernel. However, such protection mechanisms do not address the full range of problems, such as resource hoarding, that can arise when application code is introduced into the kernel.

In this paper, we present an analysis of extension mechanisms in the VINO kernel. VINO uses software fault isolation as its safety mechanism and a lightweight transaction system to cope with resource-hoarding. We explain how these two mechanisms are sufficient to protect against a large class of errant or malicious extensions, and we quantify the overhead that this protection introduces.

We find that while the overhead of these techniques is high relative to the cost of the extensions themselves, it is low relative to the benefits that extensibility brings.

1 Introduction

Many of today's research operating systems share the goal of providing applications with a richer and more powerful interface to kernel functionality. There are several approaches undergoing exploration and analysis today. The Scout system [9] supports static specialization: Scout administrators can run a kernel that has been specialized for a particular workload on a particular machine. By targeting a particular kernel for a particular workload, Scout can take advantage of advanced compiler optimization techniques, efficient kernel code paths, and a smaller system footprint. The extensible systems, such as SPIN [4] and VINO [15] allow applications to download code into the kernel to provide improved functionality and performance. Synthetix [13] provides improved flexibility and functionality by identifying commonly executed paths and producing optimized versions of them, but does not allow applications to modify or extend the kernel.

In this paper, we concentrate on the class of extensible systems. An extensible system is one that permits clients to modify the behavior of a shared server by loading client-specific extension code into the server. Such extensibility is useful in a wide range of systems. Database clients might extend their server by loading code into it to support new data types [8]. In a traditional operating system, user applications can exploit extensibility to customize the policies and functionality implemented by the kernel (e.g., the eviction policy for the file cache, or the delivery order for signals and other asynchronous events). Finally, in a microkernel operating system, the same extensions that are of interest in a traditional operating system can be loaded into the relevant system servers rather than the operating system kernel. Although we focus on extensibility in the context of a monolithic operating system kernel, the issues and technology discussed in this paper are relevant to these other classes of extensible systems.

An extensible operating system supports the downloading of application extensions, or *grafts,* into the kernel. Allowing applications to load code into the kernel spells immediate disaster unless the kernel is well-protected against buggy or malicious grafts. Safeguarding the kernel from errant grafts requires two different forms of protection. First, the kernel must guarantee that grafts do not misuse memory by reading inappropriate data (e.g., device registers or another user's data), writing inappropriate data, or executing bad instructions. This problem has been addressed by the use of safe languages such as Modula-3 [11], as used by SPIN, software fault isolation [20], as used by VINO, or virtual memory address domains, as used by Mach [1]. Second, the kernel must ensure that grafts do not consume resources to the extent that they jeopardize the acceptable performance of the kernel and other applications. This problem has been less well researched and is the topic of this paper.

In Section 2, we discuss the different ways that grafts can inadvertently or intentionally jeopardize system integrity. In Section 3, we discuss VINO's graft architecture and how it addresses the issues raised in Section 2. In Section 4, we quantify the cost of VINO's mechanisms. In Section 5, we present related work. In

This work was sponsored in part by grants from Sun Microsystems Laboratories, Bellcore, the Sloan Foundation, and the National Science Foundation.

Section 6, we discuss the lessons we have learned in building VINO, and we conclude in Section 7.

2 How Grafts Misbehave

Our model of grafts is that they are similar to regular processes that run inside the kernel. SFI is used instead of the traditional VM mechanisms to prevent illegal data accesses. Each graft receives its own heap and stack, and when a graft changes kernel state (e.g., by opening a file), the kernel records the fact so that any such modifications can be undone if the graft misbehaves. In a trusting world, these precautions are sufficient to avoid or cope with innocent errors such as access through an invalid pointer. However, in an untrusting world, kernel extensions might be malicious, seeking to destroy system integrity, performance, or security. In building an extensible system, we strive to prevent grafts from causing more damage than can be caused by a maliciously written user-level program. Therefore, we consider it unacceptable for a bug in a graft to crash the kernel, while it might be tolerable for a graft to loop infinitely, so long as it consumes only as much processing time as would a user-level program with the same infinite loop.

There are three reasons why a maliciously written graft is more dangerous than a maliciously written process. The first is that grafts run in supervisor mode. If no special care is taken, grafts have the potential to wreak havoc with the kernel. The second is that grafts are given access to a more powerful interface. Though still restricted, grafts have access to more kernel entry points than ordinary processes, including some of the kernel's synchronization points, providing grafts with simple and effective ways to sabotage the system. The more restrictive the graft interface, the easier it is to protect against malice, but the more limited the functionality of the grafts. It is a continuing struggle to determine the correct balance between expressive power and simplicity in designing a safe graft interface. The final reason that a graft can be more dangerous than a process is that once a graft is installed, the system relies on the correct operation of the graft to provide system services, and the graft's actions can potentially affect all the processes on the system. This problem is shared by other extension models, such as upcalls. Depending on the function that a graft or user-level server provides, a malicious one can prevent the system from making progress or can cause it to crash.

We have identified five classes of misbehavior that grafts might exhibit. Each is described below. We use the classifications to derive nine guiding principles for the construction of a stable, extensible operating system.

2.1 Illegal Data Access

Because grafts run in supervisor mode, we cannot use conventional virtual memory mechanisms to prevent grafts from making illegal memory accesses. Safe languages and software fault isolation provide mechanisms for limiting the data accessible to a graft. We must also provide a mechanism that allows the kernel to determine whether a graft has been processed or compiled by such a tool.

It is not sufficient to prevent a graft from accessing information to which it is not entitled; we must also ensure that a graft cannot execute a kernel function that can provide the graft with information to which it is not entitled. This means that any interface that returns actual data to its caller (as opposed to meta-data) cannot be called by a graft. In general, the kernel can pass meta-data (such as buffer headers) freely to grafts, so long as the data to which they refer (e.g., the actual data buffers) are protected.

2.2 Resource Hoarding

Grafts can consume system resources. They can attempt to loop infinitely, allocate excessive amounts of memory, or flood the network with packets. Because the interface given to grafts is more powerful than that given to user-level code, there is the potential for grafts to cause more serious damage. For example, if grafts are allowed to acquire kernel locks, they can block kernel progress more effectively than a process that is unable to directly acquire locks. Similarly, if grafts can consume kernel resources, such as physical memory, and hold them over long intervals, resource contention and starvation can cause significant problems.

Consider this malicious code fragment:

```
lock(resourceA);
while (1);
```

If resourceA is highly contested, then we cannot let the graft hold it arbitrarily long. In fact, a graft cannot be allowed to hold any limited kernel resource for an arbitrarily long period of time.

We cannot rely on a static check to prevent grafts from hoarding resources. Therefore, we must be able to *preempt*, and, if necessary, *terminate* the thread from which a graft is called. If we terminate the thread, we undo any kernel state changes that have been made, releasing any resources held by the thread and allowing the rest of the system to make forward progress.

Through preemption and scheduling we can prevent a graft from taking more than its share of a resource. Even a graft with an infinite loop gets no more CPU time than would a user-level process with the same infinite loop.

1. Grafts must be preemptible (§2.2).
2. Grafts cannot hold kernel locks or limited kernel resources for excessive periods of time (§2.2).
3. Grafts cannot access memory to which they have not been granted permission (§2.1).
4. Grafts cannot call functions that alter or return data that the graft is not allowed to access (§2.3).
5. Grafts cannot replace restricted kernel functions (§2.3).
6. The kernel must not execute grafts that are not known to be safe (§2.1, §2.3).
7. Grafts must not call functions to which they have not been granted access (§2.1).
8. Malicious grafts can only affect applications that have agreed to use them (§2.4, §2.5).
9. The kernel must be able to make progress even with a faulty graft in its path (§2.2, §2.4, §2.5).

Table 1. Rules for Grafting. Based on the ways in which grafts might corrupt the kernel, we derive these rules for creating a safe, stable extensible kernel. We include the numbers of the sections that imply each rule.

2.3 Attempting to Use Incorrect Interfaces

There are certain parts of the kernel that cannot be extended for a single application. For example, a single application, running as a normal user, cannot be allowed to replace a global kernel policy. If it could, the application could take over the system by downloading a highly biased scheduler. Such global graft points must be accessible only to privileged users (users who, in a conventional system, would be allowed to halt the system, install new drivers, build a new kernel, etc.). Additionally, the kernel must somehow verify that the downloaded graft has been properly protected (e.g., compiled with the correct compiler). Finally, we must limit the functions that are callable from grafts. As pointed out in Section 2.1, grafts should not be allowed to call functions that return private data. Additionally, grafts should not be able to call functions that change kernel state in an unrecoverable fashion; a graft should not be able to call shutdown().

2.4 Antisocial Behavior

Our next category of graft misbehavior arises from grafts that simply do not do what they have agreed to do. Consider a scheduling graft used by a collection of clients and a server. Assume that the graft always selects the same process to run. This scheduling discipline penalizes the members of the scheduling group, but has no adverse effect on processes that use the normal scheduling discipline. We find this model of behavior acceptable, applying Cao's principle for defining an acceptable allocation policy: the selection of an application specific policy should not adversely affect other applications [5]. The task of the kernel is to prevent grafts from damaging the integrity of the kernel. We interpret this to mean crashing the kernel, corrupting data, or interfering with processes that have *not* agreed to use the graft.

2.5 Covert Denial of Service

A graft can attempt a denial of service attack, by taking advantage of the fact that the system is relying on its correct execution to make forward progress. A page replacement graft is such an example. When a page is about to be evicted, the page daemon calls the graft so that the graft can present an alternate page to replace. If the graft never returns, the page daemon cannot make forward progress causing the system to eventually run out of free pages. Therefore, it is essential to provide some mechanism by which we can detect such a throttling state and return the system to a state where it can make forward progress.

2.6 Summary

Table 1 summarizes the restrictions that we must impose on grafts if we are to protect the kernel. In the next section, we discuss how VINO addresses each of these restrictions.

3 The VINO Grafting Architecture

VINO is an extensible operating system designed to provide resource-intensive applications greater control over resource management. VINO supports the downloading of kernel extensions, which are written in C++ and protected using software fault isolation. To facilitate graceful recovery from an extension failure, VINO runs each invocation of an extension in the context of a *transaction*. If the invocation fails or must be aborted (e.g., because it is monopolizing resources), the transaction mechanism undoes all actions taken by the invocation of the extension.

The VINO kernel is constructed from a collection of objects and consists of an inner kernel and a set of resources. VINO provides two different modes of extensibility. First, a process can replace the implementation of a member function (method) on an object; this type of extension is used to override default policies, such as cache replacement or read-ahead.

Second, a process can register a handler for a given event in the kernel (e.g., the establishment of a connection on a particular TCP port). Extensions of this type are used to construct new kernel-based services such as HTTP and NFS servers.

VINO runs on Intel's x86 processors. The machine-independent parts of VINO consist of entirely new code. Following traditional engineering practices, we have encapsulated all of the machine dependent parts of the kernel behind a standardized interface. Beneath this interface we use the machine dependent portions of NetBSD 1.0—locore, the pmap module, and the device drivers. By using the machine dependent code from a pre-existing system, which itself runs on a variety of platforms, we hope to simplify the task of porting VINO to other architectures.

In this section, we discuss the VINO kernel transaction mechanism, which is used to allow the kernel to recover from misbehavior by extensions, and how the kernel determines when to abort an extension invocation. We then describe our software fault isolation tool and dynamic linker, and give examples of the two types of grafting.

3.1 Kernel Transaction Support

We encapsulate each graft invocation in a transaction to allow us to spontaneously abort a graft and clean up its state. When a function is grafted into the kernel a small wrapper function is interposed; the wrapper begins a transaction for the graft invocation and then calls the grafted function. When the grafted function returns, the wrapper commits the transaction.

The transaction support necessary for grafts is simpler than a conventional data manager's transaction mechanism. The single goal of graft transactions is to provide a means for backing out changes made by faulty grafts. Therefore, the transaction system does not need to handle permanent data, so its log need only be transient, and it never has to "redo" operations; it only has to undo them[1]. Therefore, of the four "ACID" properties typically associated with transactions (atomicity, consistency, isolation, and durability), we need only provide the first three. However, because graft functions may indirectly invoke other grafts, we found it necessary to include support for nested transactions. In this manner, any graft can abort without aborting its calling graft.

All graft transactions are managed by the default VINO transaction manager. When a transaction is initiated the manager allocates a transaction object that is associated with the thread that invoked the graft. The VINO transaction manager uses two-phase locking and an in-memory undo call stack. Because the kernel is preemptible, it must acquire locks on all resources being accessed or modified. In the non-transaction case these locks are released as soon as a thread is done manipulating the resource. When the currently running thread has a transaction associated with it, lock release is delayed until commit or abort.

Modifications to permanent kernel state are encapsulated in accessor functions (i.e. a grafted function cannot directly manipulate kernel data; it must go through data accessor functions). Each such accessor function that can be called from a grafted function has an associated undo function. Whenever an accessor function is called, if there is a transaction associated with the currently running thread, the corresponding undo operation is pushed onto the transaction's undo call stack[2]. If a transaction aborts, the transaction manager invokes each undo operation on the undo call stack, and returns a transaction abort error to the graft stub, which then calls the default function (i.e., the function that was replaced by the graft).

When a non-nested transaction commits, the locks are released, the undo call stack and transaction object are freed, and execution continues normally. When a nested transaction commits, its undo call stack and locks are merged with those of its parent. Although different in implementation, graft transactions are similar in concept to the volatile transactions used in the Quicksilver system [7].

3.2 When to Abort Graft Transactions

Transactions provide the mechanism by which we can abort resource intensive grafts, but we still need a policy to determine when to abort a graft. Grafts are allowed to run so long as they do not interfere with the behavior of other processes. For the purpose of discussing resource hoarding, we can divide the various system resources into two categories. For some resources, we are primarily concerned that a graft does not hold the resource for too long, thus becoming a bottleneck to all other threads that need the resource. We call these *time-constrained resources.* With other resources, such as memory, we wish to ensure that a graft does not use too much of the resource. We call these *quantity-constrained resources.* In VINO, we use a different technique to enforce limits on each of these types of resource.

---

1. Note that conventional transaction semantics can be provided by creating a new instance of our transaction manager and replacing the transient log manager with a permanent one.

2. Our current implementation requires that this code be added by hand. This could clearly be automated with a preprocessor and appropriate source-code decoration.

Consider the case of locks as an example of a time-constrained resource. If a graft holds a lock that no other thread requests, then continuing to hold that lock does not affect the rest of the system. Conversely, if other threads do request the lock, then the graft is potentially degrading system performance. Therefore, with every lockable resource, we associate a time-out value that indicates how long a lock can be held on that object during periods of contention. This time-out based locking also provides an implicit mechanism for breaking deadlocks. Because resource requirements vary tremendously, reasonable time-out intervals must be determined (experimentally) on a per-resource-type basis. For example, a page may be locked for tens of milliseconds during I/O while a free space bitmap should be locked for only a few hundreds of instructions while it is being traversed.

When a request for a lock blocks, the waiting thread schedules a time-out whose duration is based on the resource being requested. If the time-out on a lock expires, and the lock is held by a thread that is executing a transaction, we abort that transaction. Note that we abort the transaction even if the lock was acquired before the graft was invoked. In such a case, the graft will return to the invoking code which presumably will release the lock(s) in a timely manner.

To enforce limits on quantity-constrained resources, we use the same mechanisms for grafts that we use for user-level threads and processes. Each thread in VINO has a set of resource limits associated with it. These limits constrain the amounts of various resources (e.g., memory) that the thread may consume. When a graft is installed, it initially has limits of zero (i.e., it cannot allocate any resources). The installing thread may transfer arbitrary amounts from its own limits to the newly installed graft, or the thread can request that all of the graft's allocation requests be "billed" against the installing thread's own limits. If multiple processes wish to pool resources (e.g., a collection of database clients and servers may wish to pool their wired memory resources to create a shared buffer pool), they can each delegate their resource rights to the graft, in a manner analogous to ticket delegation in lottery scheduling [21].

When a thread invokes a grafted function in the kernel, the thread's resource limits are replaced by those associated with the graft. Thus, the same mechanisms that prevent processes from exceeding resource limits are automatically applied to grafts. When the process would normally be denied requests for new resources, the graft's requests also fail.

3.3 Graft Code Safety

As stated above, grafts are protected through the use of software fault isolation [20]. The overhead of software fault isolation has been shown to range from 5% to 200%, depending on the application. We developed an SFI tool, MiSFIT, for this purpose [17]. At compilation time MiSFIT inserts instructions to protect loads and stores. Code is added to force the target address to fall within the range of memory allocated to the graft. The cost of this protection is two to five cycles per load or store.

To protect function calls, VINO kernel developers maintain a list of *graft-callable* functions. Only functions on this list may be called from grafts. Direct function calls are checked when grafts are dynamically linked into the kernel; the function is looked up in the graft-callable list; if the target function is not on the list, the graft is not loaded into the system.

Indirect function calls (e.g., C++ virtual function calls) are checked at run-time by looking up the address of the target function in a hash table containing the addresses of all graft-callable functions. If the target function is not on the list, the graft's transaction is aborted. In general, the cost of probing a hash table depends on the contents of the table and the key being probed for. Through the use of a sparse open hash table we find our average cost is ten to fifteen cycles per indirect function call.

Graft-callable kernel routines must perform the same type of argument checking and verification that system calls do. A graft is run with the user identity of the process that installs it; graft-callable functions are responsible for checking that the user has been granted access to files, memory, and devices that the graft attempts to use. In this way the protection domain in which the graft runs is (at least in theory) the same as the protection domain of the process that installed the graft.

VINO must ensure that code loaded into the kernel has been processed by MiSFIT. MiSFIT computes a cryptographic digital signature of the graft and stores it with the compiled code. When VINO loads a graft it recomputes the checksum and compares it with the saved copy. If the two do not match the graft is not loaded. Tools that perform this type of code signing are commercially available [10].

3.4 Function Graft Example

Once a graft has been compiled, processed by MiSFIT, and assembled, it is ready to be grafted into the running system. To install a graft, an application must first obtain a handle for the *graft point.* This is accomplished by looking up the graft point in a kernel-maintained graft namespace. The name is composed of the object to

```
file_o *db;
graftpoint_handle_o *gp;

db = file_o::open("db", "r");
gp = graft_namespaces->lookup(db,
                              "readahead");
gp->replace("my_readahead.o");
```

Figure 1. Function graft example. The database server is replacing the kernel's default file read-ahead function with an application-specific version. The server looks up a handle for the read-ahead function of the database in the graft namespace. It then installs the new read-ahead function. Although the function calls shown here are to C++ member functions, they invoke VINO system calls.

be grafted (e.g., the open file) and the name of the function to be replaced (e.g., "read-ahead"). The graft point handle provides a *replace* method that is used to instruct the kernel to replace the function at the graft point with the new function. Figure 1 shows an example.

This interface enables the replacement of a single member function for a given object. The list of functions that can be grafted on each class is specified by the class designer; some classes may not allow any of their functions to be grafted; others may allow all functions to be grafted.

3.5 Event Graft Example

The interface discussed above is suitable for modifying the behavior of a single object. However, an application may want to drop an entire service into the kernel, such as an HTTP server [4], an NFS server, or a database server. Our *event graft* model is based on the idea that these services are typically, if not always, designed to respond to a stream of incoming external events. Each of these servers receives a request, processes it, and sends a response. We model servers as handlers for events, where each request is viewed as an event. We extend our definition of function graft points, introduced above, to encompass these events: event graft points correspond to the external events to which a service responds.

Along with the replacement of a graft function shown above, we also permit the *addition* of a new graft function to a graft point. Rather than replace an existing function, the grafted function will be called in addition to any other functions added to the graft point. We provide an interface for applications to specify the order in which grafted functions are called.

When an event occurs in the kernel (e.g., a new connection is established on the TCP port dedicated to HTTP, or a packet is received on the UDP port for NFS), VINO spawns a worker thread and begins a transaction. It then invokes the grafted function (passing it a file descriptor or other data required to process the event).

```
// http server installation,
// invoked at user level
    graftpoint_handle_o *gp;

    gp = graft_namespace->lookup("tcp/80");
    gp->add("http_server.o");
```

```
// http server code, run as graft in kernel.
http_server(file_o *fd)
{
    char buf[256];
    fd->read(buf, sizeof(buf));
    // process http request...
    ...
}
```

Figure 2. Event grafting example. The first code fragment installs the server code on TCP port 80; the second code fragment represents the server itself. Each time a new connection is accepted on TCP port 80 a worker thread starts a new transaction and invokes the http_server() function, passing it the file descriptor from which http_server() can read the http request. When http_server() finishes handling the connection, it returns, and the worker thread commits the transaction and closes the connection.

When the grafted function returns, the worker thread commits the transaction and exits. An example of adding a function to an event graft point, and the outline of an HTTP server graft, are shown in Figure 2.

3.6 Summary

Returning to Table 1, we can now identify how VINO copes with the various classes of misbehavior. By design, our kernel is preemptible. Therefore, any thread, including any thread that called a graft, is preemptible (Rule 1). The combination of transaction abort and resource accounting protects against resource hoarding (Rule 2). If a graft consumes too many resources or runs for too long a period of time while holding a high-contention lock, its transaction is aborted. When a graft transaction is aborted, the graft is forcibly removed from the kernel, so that new invocations of the call use normal kernel code and not the misbehaving graft code. Our SFI compiler generates instructions to prevent grafts from accessing memory to which they are not entitled (Rule 3) and from executing functions to which the graft does not have access.

Rules 4 and 7 are provided for by a combination of static and dynamic methods. When constructing the list of graft callable functions, we must exclude those functions that return data without checking for appropriate permissions (Rule 4). MiSFIT and the dynamic linker ensure that only functions on the graft callable list are invoked by grafts (Rule 7).

Rules 5 and 6 are enforced statically through our downloading mechanism. In addition to verifying that a graft does not call inappropriate functions, the dynamic loader prohibits grafting onto restricted kernel entry points, such as the security enforcement modules (Rule 5). The digital signature scheme described in Section 3.3 ensures that the kernel does not execute any grafts that are not known to be safe (Rule 6).

We believe that the combination of resource accounting, the downloading mechanism, and the separation of global and local policy decisions limits the applications affected by malicious grafts to only those applications that use those grafts and ensures that the kernel can make forward process, even in the presence of a malicious graft (Rules 8 and 9).

4 The Cost of Graft Protection

In previous work we presented a taxonomy of types of kernel extensions [16], and we use that taxonomy here to evaluate the overhead of graft maintenance in VINO. We identified three basic graft structures, each of which encompasses a broad class of kernel graft points. *Stream Grafts* act much like UNIX filters, accepting data, transforming or manipulating the data, and producing either a new data stream or result. Some examples of Stream Grafts are encryption, compression, and checksum calculation. A *Prioritization Graft* chooses a candidate from a set such as selecting a process to schedule, a page to evict, or a buffer to flush. A *Black Box Graft* is more general than Prioritization and Stream grafts; a Black Box graft has some number of inputs, some state, and a single output. From outside the graft, it appears as a "black box" function, producing a single output value. File system read-ahead, access control checking, and name resolution are examples of Black Box grafts.

In this section, we present sample grafts from each class and quantify the overhead associated with making the graft safe. The VINO system is still in its infancy, so we cannot run large, complex applications. For this reason, we perform the analysis at the graft level, as opposed to the application level. This allows us to perform fine grain measurements and also makes our measured overheads as conservative as possible. For example, if our protection mechanisms impose a 25% penalty on the graft in isolation, the observed penalty in a complete application can only be smaller.

Table 2 outlines our measurement methodology, identifying how we decompose each graft to isolate individual overhead components. Figure 3 depicts the code paths and general structure of our grafts, highlighting the typical paths that we measure. In an effort to encapsulate the full cost of extensibility, we measure our *base path* by removing any levels of indirection and results checking that we introduced to facilitate grafting. The *VINO path* measures our normal kernel paths; it includes any extra levels of indirection we impose, but no transaction overhead. The *null path* includes full support for grafting, including transaction begin and end, but does the minimal amount of work possible for each example graft. The *safe* and *unsafe paths* include the full graft path, with transactions, and quantify the MiSFIT overhead in the difference between the two paths. Finally, the *abort path* results from a transaction abort at the end of the graft execution in the *safe path*. As we measure the cost of these increasingly complex execution paths, we report both the total execution time of each path and the incremental overhead between successively complex paths.

Measurement	Explanation
Base path	Kernel code path with all extra indirection and graft-support removed.
VINO path	Normal VINO kernel path, with indirection for graft support and return-value verification.
Null path	Includes graft stubs, transaction begin and commit, and minimal (null) graft.
Unsafe path	Includes full graft code and lock overhead
Safe path	Includes code protected with MiSFIT.
Abort path	Complete safe path with transaction abort instead of commit.

Table 2. Measurement Methodology. Each graft benchmark will decompose the graft cost into the components described here.

Our test platform consists of an Intel Endeavor motherboard with a 120 MHz Pentium processor, a 512 KB pipeline burst L2 Cache, and 32 MB of 60ns EDO DRAM. We use a single 5400 RPM Fujitsu M2694ESA disk with a SCSI interface, a formatted capacity of 1080MB, an average seek time of 9.5 ms, and a 64KB buffer. As our tests were performed on a Pentium, we were able to take advantage of the hardware cycle counter on the CPU. We computed the number of cycles for each test, and then using the clock speed of the processor, converted from cycles to microseconds. To reduce the sensitivity of our results to cache effects, we drop outliers by eliminating the top 10% and bottom 10% of the measurements before computing the means and standard deviations. (We ran each test between 300 and 3000 times depending on the test.) In most cases the standard deviations were negligible (less than 2.5% of the mean). We observed higher standard deviations for very short duration events, because an individual cache miss can account for a significant fraction of the measurement. In a few of the tests, we still find differences in cache behavior between test cases; in these cases, we explicitly measure and report the additional cache overheads.

For each of our sample grafts, we perform a simple cost-benefit analysis. In each case, the cost of the graft is the time to execute the grafted function, along with the general overhead of executing a graft (transaction protection, MiSFIT overhead, etc.). The difference between the *safe path* and the *VINO path* provides the total cost of a graft in terms of the overhead that it adds to the system. The difference between each successive pair of measurements in Table 2 corresponds to one part of the overhead. The *null path* adds the cost of transaction protecting a graft function to the *VINO path*. The cost of the graft function itself can be determined by comparing the *null path* with the *unsafe path* (where we add the graft function without any MiSFIT protection). Finally, we compute the MiSFIT overhead by comparing the *unsafe path* and the *safe path* measurements.

The benefits associated with each graft depend on the specific functionality being grafted into the kernel. We estimate the benefit derived from each of our test grafts and compare this to the cost of the graft in order

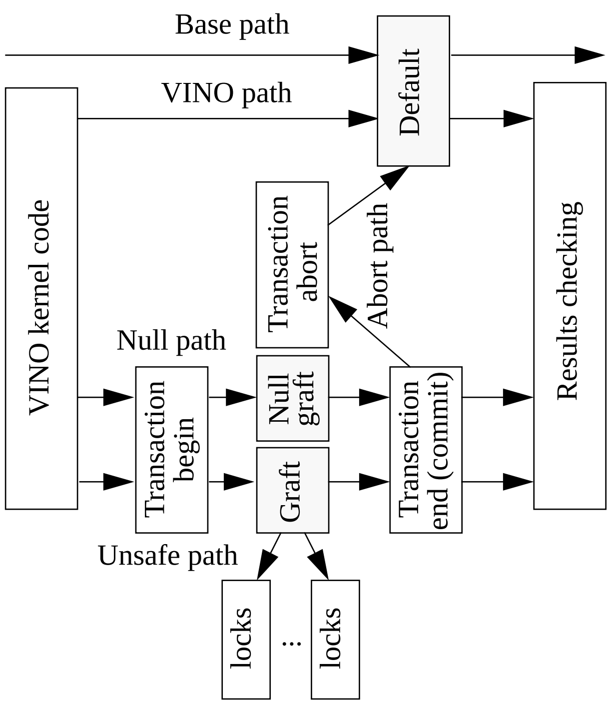

Figure 3. Graft Evaluation Model. The *base path* represents the cost of kernel functionality without the extra indirection required to support grafting. The *VINO path* includes the cost of indirection. The *null path* includes the graft overhead and any additional overhead required to check the graft's return value(s), but not the actual graft code. The *unsafe path* includes the actual graft, but does not include MiSFIT overhead, and the *abort path* is the complete, *unsafe path* with transaction commit replaced by a transaction abort. The shaded boxes represent points where policy decisions are made.

to determine the suitability of the VINO extension architecture for the different types of grafts that we have measured.

4.1 A Black Box Graft: Read-Ahead

File read-ahead is an example of a policy for which traditional operating systems implement a general algorithm that is good for most applications, but not necessarily optimal for all. A typical file read-ahead policy operates on the assumption that most applications perform sequential I/O. When the system detects sequential access to a file, it asynchronously prefetches some additional amount of file data with each read request. Because most file accesses are sequential [3], this policy usually improves performance. There are several cases, however, where this general policy does not improve (and can even degrade) application performance [12].

At first glance, it may seem that user-level threads are the simple and obvious solution to application-directed prefetching. However, without kernel support, a strictly user-level prefetching implementation is unable to exploit kernel-level information about the on-disk layout of the file data. Similarly, without explicit kernel support, an application that performs many short sequential reads to different offsets in a large file may incur the overhead of having the kernel prefetch unneeded parts of the file. Finally, thread-level prefetching is indistinguishable from normal user I/O and the kernel is unable to assign accurate priorities to pending I/O requests.

4.1.1 Cost-benefit Analysis of Read-ahead

File read-ahead is one of the most appealing kernel policies to graft, because the potential gains are large. For the remainder of this discussion, we will consider read-ahead in the context of a specific hypothetical application, modeled on a random access workload such as might be generated by a database server. The core of this application is a loop that reads a block of data, and then performs some computation on it. We assume that the application reads the blocks of data in a non-sequential order, but has advance knowledge of what blocks it will need.

Because the default kernel read-ahead policy only supports sequential access, no prefetching would normally be done. Thus, each time the application issued a non-sequential read request, it would block until the data was fetched from disk. If the application can graft a new read-ahead function onto the file, however, then each time it reads one block, it can also prefetch the next block. Thus, we can imagine the application reading block A and prefetching block B, computing on block A, then reading block B and prefetching block C, and computing on block B.

What are the costs and benefits associated with this scheme? Consider the request to read block B. The cost for the application is the overhead associated with prefetching the block as described above. The benefit to the application is that the amount of time that the application is suspended waiting for block B is reduced by the amount of time since that prefetch request. This is the amount of time that the application spent computing between read requests. Thus, the application will win if the cost of the read-ahead graft is less than the time the application spends between read requests.

4.1.2 Implementing the Read-ahead Graft

In VINO, application level file descriptors are handles for kernel level *open-file* objects. Traditional file-related system calls (read, seek, etc.) are translated to method invocations on the appropriate open-file. Whenever a user issues a read request, the corresponding method on the open-file handles the read, and then calls its *compute-ra* method to determine which (if any) additional file blocks should be prefetched. This function is passed a descriptor describing the offset and size of the current read request, and is allowed to provide a list of additional file extents that should be prefetched. These prefetch requests are passed to the underlying file system where they are added to a per-file prefetch queue. The file system removes prefetch requests from this queue and issues them to the I/O system as memory becomes available for read-ahead. (In this manner, if a graft of the *compute-ra* function asks for 100MB to be prefetched, it will not steal all of the system's memory pages. Instead, the 100MB will be prefetched in order, as pages become available.) The allocation of memory buffers to satisfy read-ahead requests is determined by a global policy that cannot be grafted by users with normal privileges.

The default read-ahead policy used by VINO only prefetches when the user accesses a file sequentially. Applications that wish to specify an alternate prefetching policy do so by grafting a new *compute-ra* function onto the appropriate open-file object.

As described above, our hypothetical application would benefit from a read-ahead policy that permitted it to specify the blocks to be prefetched. To this end, we implemented a graftable read-ahead policy for non-sequential access. A memory buffer is shared between the application and the read-ahead graft, allowing the application to specify its anticipated file access pattern. The graft version of the *compute-ra* function uses the data in this shared buffer to issue read-ahead requests.

4.1.3 Measuring Read-ahead Graft Overhead

We tested the read-ahead graft by reading three thousand four kilobyte blocks in a random order from a twelve megabyte file. Each time the application code issued a read request to the open file object, it also placed the location and size of its subsequent read in the shared buffer so that it could be prefetched. Table 3 shows the overhead for the read-ahead graft.

	Overhead (µs)	**Elapsed time (µs)**
Base path		0.5
Indirection cost	*1.0*	
VINO path		1.5
Transaction begin	*36.0*	
Null graft cost	*1.5*	
Transaction commit	*+ 28.0*	
Incremental overhead	*65.5*	
Null path		67.0
Lock overhead	*33.0*	
Graft function	*2.0*	
L1 cache miss time	*+ 2.0*	
Incremental overhead	*37.0*	
Unsafe path		104.0
MiSFIT overhead	*3.0*	
Safe path		107.0
Abort cost (additional, above commit time)	*1.0*	
Abort path		108.0

Table 3. Read-ahead Graft Overhead. The read-ahead graft decides which page should be prefetched with each read request. The *base path* measures the time to select the next (i.e., sequential) block. The graft function allows the user to specify an access pattern. This simple function has a short execution time, yielding a disproportionately large graft overhead.

From these measurements, we see that the cost of executing the grafted read-ahead function (the difference between the *unsafe path* and the *null path*) is 37 µs. Most of this cost is the overhead of acquiring a lock before accessing the shared memory buffer. The other cost of executing the grafted read-ahead function is the grafting overhead—the cost of running the graft function in a transaction, the overhead of applying MiSFIT to the graft function, and the extra checking required to validate the values returned by the graft function. Table 3 shows that the total cost of starting and committing a transaction is 64 µs. The overhead imposed by MiSFIT, derived by comparing the measurements of the safe path and the *unsafe path*, is another three microseconds. Thus the total grafting overhead for this function is 65.5 µs.

Returning to our cost benefit analysis, the total cost of executing the grafted read-ahead function (the *safe path* from Table 3) is 107µs. Thus, our application would benefit from using this graft assuming that it spends at least 107 µs between read requests. For comparison, it takes 137 µs to sum a four kilobyte array of integers on our test machine. (4KB is our file system block size.)

4.2 A Prioritization Graft: Page Eviction

Virtual memory page eviction is another example of a policy for which traditional operating systems implement a general algorithm (e.g., some variant of the clock algorithm) that is good for most applications, but not all. Applications for which LRU is the right paging strategy will enjoy fine performance under this algorithm, but there are cited cases where such an algorithm is suboptimal [2,12,18].

The key challenge in supporting application-provided page out selection is to do so in a manner that does not compromise the integrity of the virtual memory system. There are three requirements necessary to enforce this. First, the page eviction decision must be made in a timely fashion, because poor VM performance can slow the entire system. Second, the value returned by the graft must be valid (or detectably invalid). Third, the graft cannot permit the application to use more physical memory than would be allowed if the same application ran without a page eviction graft.

4.2.1 VINO VM Page Eviction

The VINO virtual memory system is based loosely on the Mach VM system [14]. A virtual address space (VAS) consists of a collection of memory objects mapped to virtual address ranges. A memory object represents a contiguous piece of data that may be backed by a variety of objects such as a device, a network connection, or a file. Once a memory object is associated with a particular object, the object becomes responsible for handling page faults to the memory object in a manner appropriate for the materialized item (e.g., read a file from disk, read data from a network connection).

Virtual memory page eviction is implemented by a two-level eviction algorithm. A global page eviction algorithm selects a victim page. Then, if the owning VAS has installed a page eviction graft, it invokes the graft passing it the victim page and a list of all other pages that the virtual memory system currently assigns to the particular VAS. The VAS-specific function can accept the victim page or suggest another page as a replacement (similar to Cao's replacement strategy [5]).

The global algorithm then verifies that the selected page belongs to the specific VAS and is not wired. If either of these checks fails the system ignores the request and evicts the original victim. When an acceptable choice is returned, we use Cao's approach and place the original victim into the global LRU queue in the spot occupied by the replacement specified by the graft.

	Overhead (µs)	**Elapsed time (µs)**
Base path		39
Indirection cost	*1*	
VINO path		40
Transaction begin	*52*	
Null graft cost	*2*	
Transaction commit	*34*	
Results checking	*+2*	
Incremental overhead	*90*	
Null path		130
Lock overhead	*34*	
Graft function	*160*	
Results checking	*+5*	
Incremental overhead	*199*	
Unsafe path		329
MiSFIT overhead	*26*	
Safe path		355
Abort cost (additional, above commit time)	*–7*	
Abort path		348

Table 4. Page Eviction Graft Overhead. The grafted code runs in addition to the default code. Because the pagedaemon runs as a background thread, its behavior is not completely deterministic, and we observe high standard deviations when calculating incremental overheads. While the measurements with over 329µs duration had less than 3% standard deviation, short-duration measurements had high standard deviations (the highest was 16% for the *VINO path*). For both *unsafe* and *safe paths*, the graft overrules the default victim selection. The total cost of the *abort path* is lower than the *safe path*, because results checking and list manipulation are simplified.

4.2.2 Measuring Page Eviction Graft Overhead

We tested our sample page eviction graft with an application that has a 2MB data footprint of which a few pages are performance critical. The application and graft share a region of memory in which the application places the page numbers of those pages it wishes to retain in memory. During page out, the graft checks the globally selected victim to ensure that it is not one of the pages listed by the application. If it is, the graft scans the list of pages that it is allowed to evict, returning the first page it finds that is not on its list of important pages. Table 4 shows the different measurement paths for this graft.

When a graft disagrees with the default victim selection, the cost of victim selection increases by an order of magnitude, but is quite reasonable compared to the cost of the I/O operation that might be saved. If we apply a cost-benefit analysis, the cost of adding the graft is 316 µs, while the benefit of avoiding a page fault is approximately 18 ms in our system. The graft can disagree with the victim selection approximately 57 times for each I/O that we save. In addition, the cost is reduced to 159 µs when the graft agrees with the default victim selection, and because the pageout daemon runs asynchronously, the increased cost of victim page selection is unlikely to reduce application performance.

4.3 A Prioritization Graft: Scheduling

It is often the case that a group of threads or processes work in concert, and should be scheduled as a group. For example, a database server process and its clients can be thought of as a single application; when there are no outstanding server requests, the server process should not be scheduled, but when several clients are blocked on requests to the server, the server process should be given a proportionally larger share of the total CPU in order to more quickly reply to the outstanding requests.

Each user-level process has associated with it a kernel-level thread. When the kernel thread is chosen to be run next, its *schedule-delegate* function is run. The default version of this function returns the identity of the thread itself (i.e., instructions to run the selected thread). The schedule-delegate function can be replaced by grafting a process-specific function that, in the example above, would have the client return the identity of the server process when the client was waiting for the server to reply to a request.

Our example schedule-delegate graft scans a process list of 64 entries, examines each (to determine if one of the other processes should be run instead) and then returns its own ID.

The *base path* measurement is the cost of switching processes on our system, the primary costs of which are choosing which thread to run next, switching kernel threads, and switching VM contexts. In this case, the *VINO path* differs from the *base path* only in a call to a function that returns the new threads's ID and the code to verify that the returned ID is that of a valid thread (which is accomplished by probing a hash table containing the valid thread IDs). The *null path* adds transaction support around an invocation to this trivial function. The *unsafe path* invokes the graft described above without SFI protection, and the *safe path* includes the cost of SFI protection. The results are presented in Table 6.

	Overhead (µs)	**Elapsed Time (µs)**
Base path (two switches)		54
Indirection cost	*1*	
VINO path		55
Transaction begin	*38*	
Null graft cost	*2*	
Transaction commit	*30*	
L1 cache miss time	*+6*	
Incremental Overhead	*76*	
Null path		131
Lock overhead	*33*	
Graft function	*35*	
Result checking	*+4*	
Incremental Overhead	*72*	
Unsafe path		203
MiSFIT overhead	*5*	
Safe path		208
Abort cost (additional, above commit time)	*3*	
Abort path		211

Table 5. Scheduling Graft Overhead. The *base path* measurement includes the time to select the next process to run, switch to it, and switch back (including switching VM contexts twice). The *null path* includes a call to a function that returns its argument (the candidate thread). The *unsafe path* adds the invocation of a non-trivial function that locks and searches the process list. The largest increase in overhead comes from the transaction and lock costs, which sum to twice the process switch cost.

Because this graft walks the process list, it must acquire a lock for the list. The *unsafe* and *safe paths* add this cost to the cost of the code that walks the process list; the *safe path* adds the cost of MiSFIT protection. Each iteration of the loop that walks the 64-element process list takes about 0.5 µs, primarily because our collection class implementation is not well-optimized.

The cost for this graft, starting with the fixed transaction begin/commit cost, is higher than the *base path* cost (for switching processes twice). Although twice the cost of a process switch, it is still roughly 2% of a typical timeslice of 10 ms (as opposed to 0.5% for the *base path*).

The benefit of permitting processes to control scheduling is difficult to quantify; however, the benefit of being able to control which process runs next can be considerable. Multimedia applications are often structured as several cooperating processes or threads. In a conventional system, if the user interface thread is scheduled when it comes time for the application to display the next video frame, the best the UI thread can do is yield, and hope that the video thread is scheduled soon. With the ability to delegate a timeslice in the manner discussed here, the UI thread could hand off directly to the video thread, with the goal of better meeting the scheduling deadlines of the application.

An operating system with support for real-time scheduling and service guarantees might better meet the needs of this particular application; however, we do not believe that we can *a priori* determine all desirable scheduling policies and hard-code them into the kernel.

4.4 A Stream Graft: Encryption/Decryption

A stream graft is used to transform a data stream as it passes through the kernel. Examples of stream grafts are compression (and decompression), logging, mirroring, and encryption (and decryption).

Our graft performs a trivial (xor-style) encryption of data as it is copied to user level, and symmetrical decryption as it is brought into the kernel from user level. The encryption algorithm used is not computationally intensive, which is a conservative position to take. The primary cost imposed by our software fault isolation tool is protection against errant loads and stores, so the higher the ratio of memory accesses to other instructions, the higher the SFI overhead. The most trivial stream graft just copies data from input to output without transforming it; this graft has the highest ratio of stores to other instructions. Therefore, the simpler the transformation the graft performs, the more conservative the overhead estimate. For example, the cost of a computationally intensive encryption scheme (e.g., DES) would dwarf the overhead associated with software fault isolation.

Our sample graft is passed an 8KB input data buffer block and an 8KB output buffer. The graft encrypts the data into the output buffer and returns. This graft is particularly interesting in that it requires no synchronization overhead (the input and output buffers have been obtained in the caller), but offers nearly the worst case of software fault isolation overhead, because it consists almost entirely of load and store instructions. Table 6 shows the overhead for the encryption graft.

For the *base path* measurement we use the in-kernel bcopy function to copy an 8KB buffer (105 µs). The *VINO path* adds a function indirection that is sufficiently fast to be undetectable, and, as above, the *null path* adds transaction begin and commit (64 µs). The *base* and *VINO path* measurements are artificially low because they call bcopy in a tight loop. Using the Pentium on-chip counters, we measured an additional 24 µs spent servicing L1 cache misses in the *null path* case, for a bcopy time of 193 µs.

	Overhead (µs)	**Elapsed time** (µs)
Base path		105
VINO path		105
Transaction begin	*32*	
Transaction commit	*32*	
L1 cache miss time	*+24*	
Incremental overhead	*88*	
Null path		193
Graft function	*166*	
Unsafe path		359
MiSFIT overhead	*187*	
Safe path		546
Abort cost (additional, above commit time)	*4*	
Abort path		550

Table 6. Encryption Graft Overhead. As expected, this graft is a worst-case scenario for software fault isolation, imposing more than 100% overhead on the graft function.

The *unsafe path* encrypts the data as it copies it from input to output, adding another 166 µs over and above the cost of the bcopy, for a total of 359 µs. The encryption takes 3.4 times that of a straight bcopy (which is implemented using a hardware copy instruction that has a cost of only one cycle per word copied).

The cost of MiSFIT protection on the *safe path* adds 187 µs, for a total of 546 µs, or 5.2 times a straight bcopy. This overhead is not surprising, given the lack of optimization in our software fault isolation tool. Our tool protects each indirect memory access; since the graft consists primarily of memory accesses, we see a protection overhead between two and three times the cost of the function itself.

4.5 Transaction Failure Overhead

In each of our sample grafts, we measured the time required to abort the graft. This cost is a function of the number and complexity of the undo functions, the number of locks to release, and the constant overhead associated with ending a transaction. This cost varies dramatically, depending on the complexity of the graft.

For each of the grafts described above, we measured the cost of aborting the *null path* as well as the full grafted path. These measurements are shown in Table 7.

Our sample grafts have sufficiently little state that the full abort cost is only 0% to 40% more than the null abort cost. Most of these grafts have little *undo* work and few locks. While we believe that these grafts are representative of the fine-grain grafts that VINO allows, more complex grafts will have higher abort costs. The total abort time is represented by the equation: abort overhead + unlock cost + undo cost. The abort overheads we measured ranged from 32–38µs, and we measured the cost of releasing a lock at 10 µs per lock. The undo cost should be somewhat less than the actual cost of running the graft. Therefore, the abort cost equation becomes: $35\mu s + 10L + cG$, where L is the number of locks to be released, G is the cost of the graft, and c is a constant less than one.

	Null Abort (µs)	Full Abort (µs)
Read-Ahead	32	45
Page Eviction	38	50
Scheduling	33	45
Encryption	36	36

Table 7. Graft Abort Costs. For each of our sample grafts, the difference between the two columns is a function of the number and complexity of the undo functions and the number of locks that must be released.

The most significant variable in aborting a transaction occurs when the graft hoards resources and must be timed out. We currently schedule time-outs on system-clock boundaries, which occur every 10 ms. Therefore, the delay for timing out a transaction will be between 10 and 20 ms. This is obviously too coarse grain for some resources, and we expect to experimentally determine a more appropriate timing as the system matures.

4.6 Summary

The overhead associated with using transactions and software fault isolation to protect kernel integrity from misbehaving grafts varies according the type of actions performed by the graft. As the encryption graft demonstrated, MiSFIT can increase the execution time of graft code by nearly 200%. For less data intensive grafts, such as the file read-ahead graft, the MiSFIT overhead, while large relative to the cost of the graft itself, is only a few microseconds. Transaction costs are relatively stable across all grafts, increasing in proportion to the number of locks acquired on a graft's behalf. Each use of a transaction lock instead of a conventional kernel mutex lock adds approximately 19 µs to the graft's execution time and 14 µs to the abort cost.

The measured cost of running a graft in the context of a transaction can be substantial, adding as much as 200 µs to the execution time of the graft code. The true cost of downloading user code into the kernel, however, must be measured in terms of the performance and functionality gained by allowing applications to modify the kernel. Grafts may eliminate or hide disk accesses, avoid context switches, or eliminate programmer labor by allowing the reuse of kernel functionality. In many cases, the time gained more than compensates for the overhead of the grafting mechanisms. In other cases, the gains in flexibility and savings in labor will be sufficient compensation, and in some cases, the cost will outweigh any potential benefits, and we must explore other alternatives for kernel extensibility.

5 Related Work

VINO is one of many new operating systems that belongs to the class of extensible systems. It is most similar to the SPIN system [4]. In SPIN, extensions are written in a typesafe language (Modula-3) and downloaded into the kernel where they initiate a thread. Once installed, the thread can install handlers for any kernel events for which it has appropriate permission and in which it is interested. The use of a typesafe language simplifies some of the safety issues involved in building an extensible system, because the extensions cannot reference disallowed interfaces or data. Cleaning up after errant extensions is also simplified in SPIN, because the Modula-3 garbage collector can clean up state when a graft terminates. However, the areas we found most challenging, such as detecting and dealing with resource hoarding, identifying malicious extensions, and identifying the set of graft-callable and graft-replaceable interfaces, are also challenges for SPIN.

Our event graft model is similar to the event model of SPIN. Where the function graft model (discussed in section 3.4) is appropriate for simple, fine-grained graft points that correspond to single functions, event graft points provide better support for the addition of new services to the kernel.

The Exokernel project [6] is an extreme example of an extensible system. The goal of the Exokernel project is to remove abstractions from the kernel and export a low-level machine interface directly to applications. User-level libraries implement most of the abstractions traditionally implemented by the kernel, and the kernel implements the bare-minimum functionality required to export the hardware interface to applications safely. There are two ways to extend the Exokernel. The first is to modify the user-level libraries that implement the kernel abstractions. Because VINO is a conventional kernel architecture, there is no analogy in VINO. The second method of extending Exokernel is to download code into the kernel and use software fault isolation, as is done in VINO, to ensure safety.

Another approach to extensibility is to provide an interpretive environment in the kernel in which kernel extensions can be run. The interpreter can ensure safety by preventing extensions from wreaking havoc in the main kernel, but often incurs a significant runtime overhead [16].

The adaptable systems, such as Synthetix [19], take a different approach from the extensible systems. Rather than having applications explicitly modify the kernel's behavior, Synthetix is designed so that commonly executed paths through the operating system can be specialized. For example, the common path through the file system accesses the same file descriptor and the same or sequentially next block in the designated file. By providing a specialized component that removes branches and the normal code to map file descriptors to kernel structures, the performance of the normal case can be greatly improved [13]. The only additional cost comes in the form of checks that distinguish between the normal path and the specialized path and allow the system to execute the correct one at the correct time. This approach is sufficient for improving performance for paths that already exist in the kernel, but does not address functionality that is not present in the kernel initially.

6 Lessons Learned

C++ bought us some headaches that a safe language such as Modula-3 would have avoided. In particular, garbage collection would have simplified our task of cleaning up after a misbehaved graft aborts. We could also have avoided work-arounds such as delaying deletes until transaction abort. Finally, the SFI overhead for data intensive grafts, such as encryption, is irritating; a more constrained language would have provided protection at the compiler level. While we have succeeded in making our kernel robust against malicious grafts, it has been painful at times.

By far, the most challenging issues were not language issues; they were system design issues. Repeatedly, we found ourselves making trade-offs between restricting the graft interface and adding overhead to handle all the types of malice we foresaw with the broadened interface we provided. In general, we strove to make grafts as flexible as possible, even when it bought us extra complexity in the design or extra overhead in the protection mechanisms. The system is still too young for us to determine whether these trade-offs were justified. As we gain more experience with sophisticated applications, we can reevaluate this design decision.

```
get_lock(requested_lock) {
    for (lock=head(holders_list);
        lock != NULL;
        lock=lock->next)
        if (is_conflict(lock,
                        requested_lock)) {
            append(lock, waiters_list);
            break;
        }
    if (lock != NULL)
        append(lock, holders_list);
}
```

Figure 4. Conventional lock algorithm.

We also discovered that we had to think very differently when designing a system for fine grain extensibility. Every decision that might conceivably be extended had to be encapsulated in an interface. This encourages extreme modularity. (At this point, we have not been able to completely quantify this cost.) For example, a conventional lock manager might implement the `get_lock` request as shown in Figure 4. Unfortunately, this code encapsulates at least two policy decisions. First, it assumes that any incoming lock request can be granted if it does not conflict with any holders, ignoring the locks on the wait list (e.g., it implements a reader priority locking protocol). Second, it assumes that locks should be appended to the waiters list, implying an ordering. A more general implementation of `get_lock` is shown in Figure 5. This implementation encapsulates each policy decision at the cost of a level of indirection at each decision point. On our system, function calls typically cost approximately 35 cycles at 8.3 ns/cycle; these add up remarkably quickly.

Perhaps the most daunting design issue that confronted us was selecting the right abstraction for grafts. Are they threads? Are they simple functions? We revisit this question regularly, but our current position is that grafts are effectively user-level processes that happen to run in the kernel's address space. As processes are isolated from the kernel by address space boundaries, grafts are isolated from the kernel by software fault isolation. Grafts interact with the kernel

```
get_lock(requested_lock) {
    if (can_grant = grantable(lock))
        lock_add(lock, holders_list);
    else
        lock_add(lock, waiters_list);
}
```

Figure 5. Encapsulated lock algorithm.

through a selected set of interfaces, but these interfaces are much lower level and functionally richer than the processes' system call interface. In an ideal world, grafts should look just like other kernel code, and for the most part, they do.

7 Conclusion

Two simple mechanisms, software fault isolation and transactions, protect our kernel from mischievous extensions imposing penalties ranging from 104 to 270 µs. In all of our test cases, these costs are outweighed by the potential benefits of the grafts. Because grafts either provide functionality not present in the system or significantly improve performance, we believe that such overhead is acceptable for most scenarios. It is certainly possible that we have overlooked classes of misbehavior that we cannot detect and/or handle, but our mechanisms are applicable across a wide range of extensions.

Acknowledgments

We would like to thank the program committee and reviewers for their helpful comments, and especially our shepherd, Jim Gray, for his insight and assistance in improving the paper and its presentation. We would also like to thank the members of the VINO group, especially Aaron Brown and David Holland, for their help.

References

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A New Kernel Foundation for UNIX Development," *Proc. Summer 1986 USENIX Conf.,* Atlanta, GA, July 1986, 93–112.

[2] Appel, A., Li, K., "Virtual Memory Primitives for User Programs," *Proc. ASPLOS IV,* Santa Clara, CA, April 1991, 96–107.

[3] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System," *Proc. 13th SOSP*, Pacific Grove, CA, Oct. 1991, 198–212.

[4] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. 15th SOSP,* Copper Mountain, CO, Dec. 1995, 267–284.

[5] Cao, P., Felten, E., and Li, K., "Application-Controlled File Caching Policies", *Proc. 1994 Summer USENIX Conf.,* Boston, MA, June 1994, 171–182.

[6] Engler, D., Kaashoek, F., and O'Toole, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th SOSP,* Copper Mountain, CO, Dec. 1995, 251–266.

[7] Haskin, R., Malachi, Y., Sawdon, W., and Chan, G., "Recovery Management in QuickSilver," *ACM TOCS 6*, 1, Feb. 1988, 82–108.

[8] Illustra Information Technologies, "Introduction to Illustra," Part No. ILL0795-01Ill, *Illustra Web DataBlade User's Guide*, Release 2.1 Beta. Sep. 1995. Part No. WEB-00-12-UG.

[9] Montz, A., Mosberger, D., O'Malley, S., Peterson, L., Proebsting, T., Hartman, J., "Scout: A Communications-Oriented Operating System," Department of Computer Science, University of Arizona, Technical Report 94-20, June 1994.

[10] Microsoft Corp., "How Software Publishers Can Use Authenticode Technology," http://www.microsoft.com/intdev/signcode.

[11] Nelson, G., *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[12] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J., "Informed Prefetching and Caching," *Proc. 15th SOSP,* Copper Mountain, CO, Dec. 1995, 79–91.

[13] Pu, C., Autrey. T., Black. A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K., "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proc. 15th SOSP,* Copper Mountain, CO, Dec. 1995, 314–324.

[14] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *Proc. ASPLOS II,* Palo Alto CA, Oct. 1987, 31–39.

[15] Seltzer, M., Endo, Y., Small, C., Smith, K., "An Introduction to the Architecture of the VINO Kernel," Harvard University Computer Science Technical Report 34-94, 1994.

[16] Small, C., Seltzer, M., "A Comparison of OS Extension Technologies," *Proc. 1996 USENIX Conf.,* San Diego, CA, Jan. 1996, 41–54.

[17] Small, C., "MiSFIT: A Minimal i386 Software Fault Isolation Tool,", Harvard University Computer Science Technical Report TR-07-96, 1996.

[18] Stonebraker, M., "Operating Support for Database Management," *CACM 24*, 7, July 1981, 412–418.

[19] Volanschi, E., Muller, G., Consel, C., "Safe Operating System Specialization: the RPC Case Study", *Proc. 1st Workshop on Compiler Support for System Software*, Tuscon, AZ, Feb. 1996.

[20] Wahbe, R., Lucco, S., Anderson, T., Graham, S., "Efficient Software-Based Fault Isolation," *Proc. 14th SOSP,* Asheville, NC, Dec. 1993, 175–188.

[21] Waldspurger, C., Weihl, W., "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. 1st OSDI,* Monterey, CA, Nov. 1994, 1–11.