Building An Extensible Operating System

A thesis presented

by

Christopher Allen Small

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts

October 1998

Copyright © 1998 by Christopher Allen Small

All Rights Reserved

Abstract

When designing an extensible operating system, a developer must ensure that the operating system is protected from misbehaved extensions. Two kinds of protection are needed: first, extensions should not violate the operating system's interface, and second, extensions should not be able to leave the operating system in an inconsistent state.

The major research contributions of this thesis include:

- The design and evaluation of MiSFIT, a software fault isolation tool for the x86 architecture that ensures that extensions do not violate the operating system's interface and incurs minimal overhead.
- The design and evaluation of VINO Lightweight Transactions, a low-overhead mechanism that allows the kernel to maintain its consistency in the face of ill-behaved extensions.
- Experiments that show the end-to-end overhead of MiSFIT and VLT protection is low, on the order of 1-2%, and the net performance gain possible from using application-specific extensions is significant, in some cases more than 20%.
- A cost-benefit framework for comparing extension technologies and an evaluation comparing commonly used extension technologies.

Acknowledgments

This thesis is dedicated to my wife, Carol Sandstrom, and my advisor, Margo Seltzer. It would not exist were it not for their help and encouragement.

While I was in grad school, Carol kept me going when the going was tough, let me go to work when she saw that I needed to work, made me come home when she saw that I needed to come home, and made sure to remind me regularly how much more fun I was having in school than I had had in a real job. And she brought us our daughter Harriet, who has been both my greatest joy and my greatest motivation for finishing.

Margo leads and teaches by example. She instilled in me a core set of values: do it right, don't cut corners, understand the numbers, and explain things clearly and honestly. What she taught me has had, and will continue to have, a significant, pervasive, and lasting effect on how I approach my work. It would be entirely appropriate for me to list Margo as a co-author on every paper I write for the rest of my career.

My committee (Margo Seltzer, Tom Cheatham, Mike Smith, and Jim Waldo), provided invaluable help and guidance, especially as I was finishing up. My thanks to them for their support.

Jim Waldo provided me with a very valuable perspective *vis a vis* completion: he told me (more than once) that any thesis that is done is infinitely better than any thesis that is not done. I believe that this document falls into the former category.

My thanks to Yaz Endo, for finding a VM bug in five minutes that I'd been trying to find for days; to Keith Smith, for retaining his good humor after being my office-mate for five years; to Dave Sullivan, for his proofreading and wordsmithing skills; and to Dave Holland, for the amazing feat of taking a file system, VM, and grafting, and creating a real operating system—one that *works*.

Pauline Mitchell, and her successor, Susan Wieczorek, made being a graduate student tractable, ensuring that paperwork got done, deadlines were met, and that the rules didn't get in the way of the underlying goals. Pauline is greatly missed, and Susan's efforts are greatly appreciated.

VINO, as described here, was designed by Yaz Endo, Dave Holland, Margo Seltzer, Keith Smith, and myself. The networking subsystem was designed by Aaron Brown, Narendra Ghosh, and Kostas Magoutis, and implemented by Narendra and Kostas. Large amounts of additional work was done by Robert Haas, David Krinsky, Stephen Manley, Hany Saleeb, and David Sullivan.

After five years, it's good to be done.

Introduction 1
1.1 Motivation 1
1.2 Thesis Statement 2
1.3 Outline 2
Related Work 3
2.1 Introduction 3
2.2 Resource Management and Arbitration 3
2.3 Isolation 5
2.3.1 Introduction 5
2.3.2 Hardware Isolation 6
2.3.3 Software Isolation 9
2.3.4 Summary of Isolation Techniques 13
2.4 Transactions 13
2.4.1 QuickSilver Operating System 15
2.4.2 Rio Vista and Non-Volatile RAM 15
2.4.3 Transaction Alternatives 16
2.5 Summary 16
The VINO Kernel Architecture 19
3.1 Introduction 19
3.2 Overall Architecture 19
3.3 Compatibility 20
3.4 Support For Extensibility 20
3.5 Events 21
3.6 Summary 22
Comparison of Extensibility Technologies 23
4.1 Introduction 23
4.2 Taxonomy 24
4.2.1 Prioritization Grafts 24
4.2.2 Stream Grafts 25
4.2.3 Computation Grafts 26
4.3 Performance Analysis 27
4.3.1 Hardware Platforms 27
4.3.2 Extension Technologies 27
4.3.3 Upcall Overhead 28
4.3.4 VM Page Eviction 29
4.3.5 MD5 Fingerprinting 32
4.3.6 Logical Disk 34
4.4 Updated Java Performance Results 35
4.5 Conclusions 37
Kernel Safety 39
5.1 Overview 39
5.2 SFI Is Not Enough 39
5.3 MiSFIT Design and Implementation 40
5.3.1 Indirect Loads and Stores 41
5.3.2 Indirect Jumps and Calls 42
5.3.3 Matching Pushes With Pops 43

5.3.4 Global Data and Virtual Function Tables 43
5.3.5 Block Instructions 44
5.3.6 Saved Registers and Returned Addresses 44
5.3.7 Dynamic Linking 45
5.3.8 Optimizations 45
5.3.9 Stubs 45
5.3.10 Code Signing 46
5.4 MiSFIT Overhead 46
5.4.1 Operating System Extension Benchmarks 46
5.4.2 SPECInt Benchmarks 48
5.5 Conclusions 49
Kernel Consistency Maintenance 51
6.1 Introduction 51
6.2 Atomic Interfaces and Transactional Interfaces 51
6.3 VINO Lightweight Transactions 54
6.3.1 Abort and Commit Logs 54
6.3.2 Nested Transactions 55
6.3.3 Graft Safe and Graft Callable Functions 55
6.4 Performance Overhead 56
6.5 Conclusions 56
Application Results 57
7.1 Overview 57
7.1.1 Database Application Performance 57
7.1.2 Events 58
7.1.3 File System Layout 58
7.2 Test Platform 58
7.3 Database Performance Improvements 58
7.3.1 The Wisconsin Benchmark 59
7.3.2 Baseline 59
7.3.3 Look Ahead Page Eviction 60
7.3.4 Random Access Read Ahead 61
7.4 File Unlink Events 65
7.4.1 Events vs. Polling 65
7.4.2 Asynchronous Write Graft 68
7.4.3 Synchronous Write Graft 69
7.5 File System Layout 70
7.5.1 Write Throughput 71
7.5.2 Read Throughput 72
7.5.3 File Layout Summary 72
7.6 Summary 73
7.7 Graft Code 73
7.7.1 Page Eviction Graft 74
7.7.2 Read Ahead Graft 74
7.7.3 Unprotected Asynchronous Write Graft 75
7.7.4 Asynchronous Write Graft 75
7.7.5 Unprotected Synchronous Write Graft 76

7.7.6 Synchronous Write Graft 76
7.7.7 File System Layout Graft 77
Discussion and Conclusions 79
8.1 Overview. 79
8.2 Results and Contributions. 79
8.3 Future Directions 80
8.4 Conclusion 81
Appendix 83
The Instruction Set Architecture of the Intel x86 Processor Family 83
Glossary 85
References 87

Chapter 1

Introduction

Applications rule.

– Jim Kurose
[personal communication]

1.1 Motivation

Operating systems exist to run applications, and therefore operating systems should do their best to support these applications. However, when designing an operating system, it is often necessary to make compromises—design choices that are beneficial to one application will be detrimental to another, as not all choices can be in the best interests of all applications.

The primary way in which an operating system supports applications is by abstracting and sharing resources between them. It is in the policy choices made when abstracting and sharing resources that an operating system implicitly decides which applications it will serve well, and which it will serve poorly. For example, operating systems typically assume that resources used in the recent past are most likely to be used in the near future. This assumption drives how the operating system represents and caches resources. Fortunately, for most applications, this assumption is true. But an application whose behavior does not follow this pattern will be ill-served by the policy decisions driven by this assumption.

The goal of the VINO extensible operating system project is to give applications control over the resource management decisions, or policies, made by the kernel. The idea of giving applications control over some or all of the behavior of the operating system is not a new idea; for example, any MS-DOS application can patch (or overwrite) the kernel while the system is running, and in this way take control of the kernel's behavior. VINO is designed to permit *safe* extensibility—an untrusted application can modify the behavior of the kernel, but only in limited ways, without compromising the safety or stability of the kernel.

1.2 Thesis Statement

The goal of this thesis is to show that it is possible to construct an operating system that allows untrusted applications control over kernel policies through the use of safe extensibility, and that the cost of safety is low enough that it does not outweigh the performance advantages of the extensions added to the kernel.

Applications running on VINO are offered control over resource *use*; VINO retains control over resource *allocation*. For example, the kernel decides how many pages of physical memory an application may use, but the application decides which pages of virtual memory to map into those physical pages. In this way, the kernel is able to ensure that each application is given no more than its fair share of resources.[1]

VINO gives an application control over how its resources are used by permitting the application to extend the behavior of the kernel by loading code that implements the application's resource use policy or policies. Because the extension code comes from untrusted applications, and such applications can act in arbitrary and malicious ways, VINO must protect itself from such misbehavior.[2] VINO protects itself through the use of the Minimal i386 Software Fault Isolation Tool (MiSFIT) and VINO Lightweight Transactions (VLTs).

These protection techniques have associated costs. In this thesis I measure both the per-operation cost (using microbenchmarks) and the overhead on end-to-end application tests (by comparing the performance of the system with the protection mechanism enabled and with the protection mechanism disabled). In the end-to-end application tests I ran, the overhead of protecting the kernel from extensions is very low, less than 1%, and the end-to-end performance benefits to the application are significant, as much as 20-30%.

1.3 Outline

Chapter 2 discusses related work, and Chapter 3 introduces the overall architecture of the VINO kernel. Chapter 3 also describes how the kernel design abstracts out policies and separates resource allocation from resource use.

In Chapter 4, I present the results of a study that was done to help choose the safety technology for VINO, which isolates the VINO kernel from misbehaved extensions. The chosen technology, Software Fault Isolation (SFI), and my implementation of SFI for VINO (MiSFIT), are discussed in Chapter 5. Isolation provides safety, but does not ensure consistency; in Chapter 6 I discuss VINO Lightweight Transactions (VLTs), which are three orders of magnitude faster than conventional persistent transactions.

Chapter 7 contains two end-to-end application studies of VINO. The first study shows the benefit of giving a database system control over page replacement and file system read-ahead. The second study demonstrates the event-graft mechanism and the efficacy of the transaction mechanism.

I conclude in Chapter 8 with a summary of results and a discussion of ways in which this work can be applied and extended. Glossary and reference sections complete the document.

---

1. For this thesis I assume that leaving resource allocation decisions in the hands of the kernel permits fair resource allocation, but does not ensure it—that task is left up to the kernel developer, and is outside the scope of this work.
2. "Misbehavior" is defined in detail in Chapter 6 and 7.

Chapter 2

Related Work

Those who cannot remember the past are doomed to repeat it.

– George Santayana

2.1 Introduction

In this chapter I place the VINO extensible operating system and its contributions in the context of related work.

When compared with traditional operating systems, VINO allows applications more control over how resources are used. In the first section of this chapter, I discuss the problems of resource management and arbitration, and how global fairness can be ensured while giving applications control over resource use.

VINO allows untrusted applications to extend the behavior of the running kernel, and must ensure that these applications do not violate the safety of the kernel. The second section of this chapter discusses various isolation techniques, and how the techniques are used by other extensible systems.

The interface that VINO exports to extensions is structured using transactions, which are more commonly found in the database world. Each extension invocation is wrapped in a transaction; if the extension invocation fails, any changes the extension has made to the state of the kernel are undone. The final section of this chapter describes the transaction model, and discusses how both traditional and non-traditional transactions are used as a structuring mechanism by other systems.

2.2 Resource Management and Arbitration

Policy decisions can have a considerable impact on the performance of applications. For example, it is common to abstract away disk layout from applications—the operating system does not offer applications the opportunity to specify the absolute (or relative) location of blocks of a file on disk. For many applications, disk layout is not important, either because the application reads little data from the disk, the application's performance is not limited by disk performance, or the application's data set fits into main memory and the initial data set load time is unimportant. For some

applications, such as an application reading a video stream from the disk, disk layout can be critical—the time it takes for the disk to move the head from the location of frame *n* to the location of frame *n+1* must be less than the pause time between frames.

Kernel resource management policies are often adaptive, attempting to deal automatically with the changing needs of the running applications. For example, an application that generates a large number of page faults will be given more physical memory, in order to reduce the system paging load; a process that frequently interrupts itself by requesting I/O will be scheduled more often, with the expectation that it will not run for very long each time it is scheduled. In this way, the kernel attempts to arbitrate the implicitly defined needs of applications.

If an application is given too much control over how resources are used, the application may degrade overall system performance. For example, if the kernel uses the adaptive paging policy described above, and a process controls which pages are kept in physical memory, the process may make a bad decision and evict a page that will be needed in the near future. This will increase the paging rate of the process, causing the kernel to allocate the process more physical memory, taking pages from other, better-behaved processes.[1] Although it is not possible for the operating system to keep the process from making this type of mistake, the operating system should try to localize the damage caused by the behavior of the process.

An algorithm developed Cao et al. [Cao94] addresses this problem. The algorithm keeps track of both the kernel's paging decisions and the application's decisions. If the application makes a bad decision (the application's choice is paged in sooner than the kernel's choice would have been), the application is not rewarded by giving it more physical memory; it must suffer for its poor choice. In this way, applications that make foolish choices do not decrease the overall performance of the system. The VINO kernel uses Cao's strategy in the virtual memory subsystem to ensure fairness in physical memory allocation.

In the domain of CPU scheduling, it is often the case that the operating system allocates a measure of CPU time to each individual process; the more processes an application uses, the more CPU time it receives. Waldspurger et al. [Waldspurger94] propose a scheme where each application is given a percentage share of the total available CPU time. The application can use the share itself, or divide it among its subprocesses. The VINO scheduler uses an extension of this model, where a process may give its CPU time to any other process. Waldspurger's method can be applied to any time-multiplexed resource, such as network or disk bandwidth, although VINO currently does not attempt to fairly schedule disk or network access.

Patterson's Informed Prefetching and Caching mechanism [Patterson95] allows applications to provide explicit hints to the operating system that state which blocks of a file will be needed in the near future. These hints are used to assist in determining which blocks to evict and to prefetch. The benefits of this technique have been shown in certain cases to be as much as 30%. This technique is precisely the type of extensibility that is enabled by the VINO architecture; as I describe in Chapter 7, this functionality can be implemented with two simple kernel extensions (approximately thirty lines each). VINO applications can then take advantage of the research results of Patterson et al. and others working in this domain.

Mowry et al. have developed a method for automatically generating prefetching and caching hints [Mowry96]. This technique analyzes the behavior of numeric applications with working sets that are larger than main memory, and adds code to the application to generate prefetching and caching hints to the kernel. Work done by Kimbrel et al. unifies work on parallel prefetching and caching [Kimbrel96], which, in the past, were typically studied in isolation from each other. The results include a simple, unifying strategy that combines the benefits of previously presented

---

1. On a kernel with this type of paging strategy, any process can cause the kernel to allocate more memory to it by accessing (and faulting in) pages that are not resident. VINO does no better than any other system when faced with this behavior.

algorithms, and is applicable both in cases where performance is I/O bound (and aggressive prefetching is beneficial) and when performance is not I/O bound (and more conservative prefetching is appropriate). As with Patterson's work, both the work done by Mowry et al. and the work done by Kimbrel et al. would be well supported by the extensibility features offered by the VINO kernel, and, in fact, would use the same two prefetching and caching extensions mentioned above.

Although it is sensible for the operating system to use heuristics for managing resources in the common case, it is clear that in some cases application control over management of resources (such as page prefetching and caching) can improve application performance and reduce the overall burden on the system. For example, an application can reduce its paging burden, or asynchronously prefetch data when no other I/O is taking place in the system.

2.3 Isolation

An operating system must protect itself from untrusted applications and, if extensible, from untrusted extensions. Operating systems traditionally use virtual memory hardware to protect themselves from applications because VM hardware has characteristics that make it well-suited to this task. The needs of kernel extensions are not the same as those of applications, which implies that we should examine other mechanisms for protecting the kernel from extensions.

VINO protects itself in the traditional manner from misbehaved applications, through the use of virtual memory protection. However, the kernel loads extensions into its address space, using software fault isolation (SFI) to protect itself from misbehaved extensions.

2.3.1 Introduction

An *interface* is a collection of functions and data elements. An operating system includes both a public (exported) interface and a private (internal) interface; the public interface is for the use of clients (applications and/or extensions); a private interface is solely for the use of the operating system itself. A well-behaved application or extension communicates with the operating system through the use of its public interface. An ill-behaved application or extension may attempt to manipulate the operating system by invoking functions or manipulating the state of the private interface. An *isolation mechanism* ensures that private operations and state remain private. It must ensure not only that a client does not invoke a function that is not in the public interface; but also that the client reads and writes only the data elements in the public interface.

Isolation can be implemented in hardware or software. *Hardware techniques* include virtual memory protection (as in the case of user-level processes running on a conventional operating system), or segregating clients and servers to different machines (allowing communication only through the use of a remote procedure call mechanism).

Software techniques can be classified by how the code is made safe. *Safe languages*, such as Modula-3 and Java, do not allow illegal operations to be expressed in the source language. *Software Fault Isolation* transforms code generated by a compiler for an otherwise unsafe language, augmenting it with additional instructions to ensure its safety. *Interpreters* check the behavior of code as it runs, and do not allow the code to violate the kernel's internal interface.

When using a software technique, in order to protect itself, the system must verify that the software transformation has actually been applied. This can be done in one of two ways: the system can perform the transformation itself, when the application/extension is loaded, or the application/extension can be packaged with verification that the transformation has been applied. Such verification can take a variety of forms, from a cryptographic signature applied at transformation time (which is the technique used by VINO), to a formal proof of the safety of the

code that can be checked at load time (as exemplified by Necula's *proof-carrying code* [Necula96]).

When two pieces of code are fault-isolated from one another, they are described as being in different *domains*. The *domain crossing cost* of a fault isolation technique is the cost of moving from one domain to another (or, alternatively, the cost of communicating between one domain and another). Hardware protection techniques commonly have a higher domain crossing cost than software techniques (as discussed in Chapter 4).

Software techniques can incur a per-instruction overhead that is not present with hardware techniques. To ensure that dangling pointers are not created, safe languages must use some form of automatic storage reclamation (i.e., garbage collection), which can reduce performance. Additionally, a safe language must statically or dynamically verify that array accesses are in bounds and that pointer assignments are well typed; these checks cannot always be made at compile time. Software Fault Isolation inserts extra instructions into the code to ensure safety, instructions that take time to process. Interpreters (even those that use just-in-time compilation techniques) have both translation and run-time overhead, causing interpreted code to run more slowly than compiled code, by two to one hundred times [May87, Hölze94].

The decision to use a hardware technique or a software technique is, as a first approximation, driven by the amount of code run between domain crossings. If little code is run between domain crossings, a software technique (with its lower domain crossing cost) can be more efficient. If much code is run between crossings, a hardware technique (which can better amortize the domain crossing cost) may be the best choice. For some systems, the point may be moot: the per-invocation cost can be low enough to not matter, extensions may be invoked so infrequently that the cost of isolation is not significant, or the benefit of the extension is so great that the protection overhead is immaterial.

It is not uncommon for operating systems to allow applications to add code to the running system. Many operating systems allow device drivers to be loaded while the system is running (e.g., through the use of the *modload* facility of SVR4.2, implemented in Solaris [Vahalia96, pp. 541] and NetBSD 1.3,[2] the *sysconfig* utility of Digital Unix [DEC96, pp. 4–4], Linux *modules* [Beck98, ch. 9], and the dynamically loaded device drivers of Windows NT [Custer93, p. 272]). Few systems, however, work to ensure that the newly loaded code will not compromise safety or stability.

In addition, by ignoring the potential reliability problems, this solution has the smallest overhead. For example, microcomputer operating systems (e.g., MS-DOS [Dettman89] and the Macintosh OS [Apple94]) allow arbitrary code to be loaded into the kernel, providing no protection from misbehaved applications. On the other hand, because applications have complete control, it is much easier to manipulate and extend the kernel to provide for the needs of applications.

VINO uses Software Fault Isolation, which has both a low cross-domain cost and a low per-instruction cost. The VINO system is designed to use cryptographic signatures to ensure that dynamically loaded code was processed by the VINO software fault isolation tool, MiSFIT.[3]

2.3.2 Hardware Isolation

The Mach microkernel [Accetta86] was designed to allow kernel functionality to be moved out of the kernel address space into user-level external servers, with the goal of increasing safety, robustness, and flexibility. These servers can be modified or replaced without modifying the kernel, or multiple servers, implementing different policies, can be run concurrently.

---

2. Documented in the *modload(8)* on-line manual page of NetBSD 1.3.

3. Both MiSFIT and software fault isolation are described in detail in Chapter 5.

The high cost of switching between protection domains motivated the current generation of microkernels, which link server code directly into the kernel address space (e.g., Chorus [Guillemont91], Windows/NT [Microsoft95]), which eliminates the safety offered by isolation. The L4 microkernel [Leidke97] addresses the context switching cost directly, greatly reducing the work done by, and cost of, upcalls. (L4 is discussed in more detail below.)

The Exokernel architecture [Engler95] follows a different path: although, as with conventional microkernels, most operating system work is done at user level, instead of decomposing the system into user-level servers, an exokernel only controls access to physical devices. Applications link with a "library operating system" (libOS), which implements the abstractions typically provided by an operating system. The argument for libOSes is that much of operating system complexity (and performance overhead) is caused by the requirement of supporting multiple concurrent processes and user identities; a libOS, which supports only a single process and user, can be much simpler and more efficient than a traditional operating system. The exokernel is responsible for ensuring that one application does not trample on the resources of another, but otherwise allows applications to control how resources are allocated and used.

The Open Systems Toolkit of the DOMAIN file system [Rees86], predating the Exokernel by a decade, provided a similar service, albeit with better application support, on a conventional (POSIX-like) operating system. Associated with each object in the file system was a set of *traits*, that enumerated the operations that could be performed on the object. Traits were abstract, in that they specified an interface, not an implementation—an object could support, say, the *read/write/seek* trait, but provide its own implementation that had nothing to do with the standard implementation of the *read/write/seek* trait. Trait implementations ran in user space, and were automatically and dynamically linked to an application as they accessed file system objects. In this way, the (apparent) behavior of file system objects could be extended, entirely at user level.

For example, an auto-compressing implementation of the *read/write/seek* trait would compress data as it was written to the file and uncompress data as the file was read. Files that used this trait implementation would appear, to applications, as if they were standard files, but the auto-compression implementations of read and write would be automatically loaded (and transparently run) by any application that used an auto-compressing file.

2.3.2.1 Minimizing Domain Crossing Cost

There are two components that contribute to the cost of hardware context switching. In addition to moving the flow of control between fault domains, if the originating domain wishes to communicate with the receiving domain, it is necessary to move data from the former to the latter. In this subsection I discuss work done to reduce both costs.

Hardware protection has the advantage that, because it is built into the hardware, overhead is incurred (above and beyond the intrinsic cost of running the code) only when the system changes fault domains. Unlike with software techniques, there is no per-instruction overhead associated with hardware protection. The downside to hardware protection schemes is that the cost to cross fault domains is typically much higher than with software techniques.[4]

Changing hardware fault domains (also known as a hardware context switch) is expensive for three reasons: hardware state must be updated, the contents of the hardware caches may be invalidated or overwritten, and the system often performs bookkeeping when it switches contexts.

---

4. I assume here that the same hardware is being used for the hardware-protection and software-protection cases being compared. Some hardware protection techniques may be more expensive (or impossible) on certain types of hardware; some kinds of software protection are less expensive to use on RISC machines than on other machines. My statement here assumes that the hardware platform is fixed, not malleable.

There is hardware state associated with each context, and this state must be updated when contexts are changed. The trade-off here is that state cached in hardware is accessed more quickly than state kept in memory—but that state must be saved and restored on context switches. For example, the memory management unit (usually) needs to be informed that a different process is in context, which can entail switching page tables and flushing the memory management unit translation lookaside buffer (TLB). At the other end of the cost spectrum, the contents of the machine registers need to be saved and reloaded (a cost incurred by both software and hardware techniques).

Most modern hardware uses high-speed (often on-CPU) memory caches in order to reduce the cost associated with accessing main memory. But such caches tend to be small, and their contents are often overwritten when switching contexts. This is, in part, due to the amount of (kernel) code run when switching contexts; by reducing the amount of code run on a context switch, less of the cache is overwritten, and more code is retained.

Also, operating system kernels typically keep track of statistics about each running process, and these statistics and other bookkeeping data need to be updated when the system switches contexts.

These factors combine to cause context switches to take thousands or tens of thousands of machine cycles [McVoy96]. However, there has been considerable progress made in bringing domain crossing costs down, primarily concentrating on reducing the amount of hardware and system state changed on a context switch. Perhaps the most technically impressive work has been done on Leidke's L4 microkernel [Leidke97]. By carefully laying out kernel code so that it remains in the level one cache and minimizing the work done by the kernel when changing fault domains, L4 performs hardware fault domain (address space) switches in 121 machine cycles (0.73µs on a 166MHz Pentium). Leidke justifies his minimalist approach to context switching by arguing that *any* work done in the kernel that does not directly contribute to changing contexts need not, and should not, be done in the kernel as part of a context switch. When this code is added back in, context switch time increases to that of comparable systems, as it was in later work layering Linux on top of L4 [Härtig97]. In this work, a context switch took 17.1µs, approximately twice as long as a context switch in Linux running on the native hardware (8.05µs).

Exception handling is a special case of context switching. Typically, when an exceptional condition (e.g., a page fault) is raised by a process, the process is suspended and the kernel handles the condition. In some cases a process may wish to handle the exceptional condition itself (e.g., implement its own page fault handler at user level). In this situation, when an exception occurs, the hardware "switches" from the address space of the process, through the kernel, and back to the originating address space (user level $\longrightarrow$ kernel level $\longrightarrow$ user level). Because the final address space is the same as the initial address space, in some cases it is possible to reduce the work done.

For example, Thekkath's work on fast exception handling [Thekkath94] showed an exception delivery round-trip time of 8µs on a 25MHz MIPS R3000 (not directly comparable to Leidke's L4 result, but in the same ballpark). Like L4, Thekkath's system minimizes the amount of work done when changing fault domains. In addition, exceptions are delivered to the same process that initiated the transfer; as a result, the kernel has less internal state to update, and hence less work to do.

2.3.2.2 *Communication Between Fault Domains*

The most general approach to communicating between fault domains involves marshalling[5] the arguments, copying the data from the originating domain into the kernel, then out again to the receiving domain, and finally unmarshalling the arguments. Bershad et al. [Bershad89] found that this level of complexity and generality was not necessary in the common case. In a study of 1.5 million cross-domain calls, none required the generality of marshalling, and sixty percent of the calls transferred 32 or fewer bytes. They found that the size of the message sent between domains had a bimodal distribution – either a small number of bytes, or a large number of bytes (where a large block of data, e.g., a disk block, is being sent from the originating domain to the receiving domain). In the former case, it is possible to reduce the communication overhead by simply loading the arguments into registers in the originating domain before the call is made. In the latter case, the kernel can perform a block copy of the data without interpreting it (saving marshalling time), or the domains can use virtual memory hardware to share a region of memory between them, and pass the message via this shared region without involving the kernel.

2.3.2.3 *Summary*

By reducing the amount of kernel and hardware state modified when a context switch takes place and taking advantage of techniques that reduce communication costs in the common case, the overhead of hardware context switches can be reduced considerably. However, the cost will still be higher than that of a function call, which is the theoretical lower bound on software context switching times.

2.3.3 Software Isolation

There are three basic software protection schemes: safe programming languages, software fault isolation (SFI), and interpretation. The three can be categorized by the use of static protection techniques (safe programming languages) vs. the use of run-time protection techniques (SFI and interpretation). In the former case, most or all checking is done at compile time; in the latter, an intrinsic run-time overhead is paid to ensure safety.

2.3.3.1 *Safe Languages*

A safe language is one in which it can be guaranteed that a pointer to an object of type T will always point to an instance of type T. In order to ensure this, the language must do several things, including: supporting automatic free store management (garbage collection) so that dangling references cannot be created; checking array bounds on accesses (so that a program never indexes outside the bounds of an array); and ensuring (either at compile-time or at run-time) that, when assigning a value to a variable, the value is of an appropriate type. Examples of safe languages that have been used for systems programming are Modula-3 [Nelson91], Oberon [Mössenböck94], and Java [Gosling96].

Even with all of these additional features, safe languages can perform as well, or nearly as well, as unsafe languages such as C; for example, the performance overhead of Modula-3 relative to compiled C or C++ is negligible [Sirer96]. Many required checks can be made statically and compiled out, and the restrictions imposed by a safe language can enable more aggressive

---

5. Marshalling is the process of serializing (writing out in a linear form) a data structure that may not be stored linearly in memory. Additionally, marshalled representations are often self-describing (include meta-information about the data) and machine independent (so that they can be easily used by a machine with a different byte order).

optimizations than are possible with less safe languages (e.g., some pointer aliasing problems are greatly reduced). The cost of automatic free store management in "typical" cases is 10-30%, but with appropriately chosen garbage collectors, the overhead can be reduced below this level [Jones96]. Arguments have been made that some automatic free store management algorithms can, in fact, *improve* performance, because they can reduce memory fragmentation (reducing paging and cache conflicts) and concentrate the most frequently used objects into a small number of pages [Seidl97].

Some database servers allow clients to load query- or data-type-specific code into the server to improve performance. The Thor database server uses a typesafe language, Theta, for writing these extensions [Liskov95]. The SPIN operating system [Bershad95] is written in Modula-3 and uses it as its extension language as well.

A disadvantage of requiring that extensions be written in a safe language is that the extensible system either needs to be written entirely in the language used for writing extensions, or the system needs to deal with the complexity of communicating between the safe environment (where extensions run) and the unsafe environment (where the rest of the code lives). Since these two environments are likely have different data models, any data passed between the two environments may need to be reformatted (in essence, marshalled and unmarshalled), and the garbage collector needs to be made aware of any data that is shared between the safe and unsafe environment. [6]

When choosing the fault isolation technology for VINO, we decided that the complexity of communicating between a safe language and an unsafe language was a larger burden than we were prepared to take on. We were not in a position to write the entire kernel in a safe language, as we reused portions of the NetBSD kernel, which was written in C.

2.3.3.2 Software Fault Isolation

The term Software Fault Isolation (SFI) is, at times, used as a general term for any type of software isolation. The term "sandboxing" is then used to describe the particular software fault isolation technique I discuss in this thesis. Confusingly, the term "sandboxing" is used at other times as a generic descriptive term, and "Software Fault Isolation" as a term for the specific technique I discuss here.

In this thesis I use both "Software Fault Isolation" and "sandboxing" to describe a particular technique for software protection that was introduced by Wahbe et al. [Wahbe93]. This technique assumes that an extension is given a region of memory for it to use (its "sandbox"). The crucial observation made by Wahbe et al. is that it can be substantially less expensive (i.e., can take fewer machine cycles) to restrict a memory reference to the sandbox than it is to verify that a memory reference falls within the sandbox.[7] The instructions that restrict memory references can be added by the compiler, by a post-processing tool, or at extension load time. The results of Wahbe et al. show a run-time overhead of five to ten percent.

A follow-on to this work was the Omniware Portable Code system [Adl-Tabatabai96]. The Omniware compiler generates portable code for an abstract virtual machine (OmniVM), which is translated to native fault-isolated code at runtime. Along with the source language independence provided by software fault isolation techniques, the Omniware system also offers target-independent portable code. Unlike the Java byte code intermediate form (see Section 2.3.3.4, below), the OmniVM intermediate form made no claims to safety; it used a standard RISC-like

6. If the safe and unsafe environments have the same data model, as is the case with VINO, marshalling costs can be greatly reduced or eliminated.
7. The former can be done by setting the high bits in the memory reference to match those of the sandbox (with two instructions: an "and" and an "or"); the latter requires a comparison and a conditional jump.

instruction set. At run-time the intermediate form was interpreted or compiled to native code, with SFI instructions added to ensure the safety of the code.

Silver developed a version of the Gnu C compiler, which generates software fault isolated code for the DEC Alpha processor [Silver96]. Most of the modifications to gcc were made in the machine-independent portion of the compiler, although some changes were needed in the machine dependent portion of the code. The implementation takes advantage of the large number of registers available on the Alpha processor. The author states that porting the modifications to the x86, which has a severely limited register set, would be difficult.

Several other researchers in the area of extensible operating systems have developed one-off software fault isolation tools, including Banerji [Banerji96] and Engler [Engler95]. Unfortunately these tools suffer from working on less widely used platforms, working only with domain-specific languages, or not being publicly available.

VINO uses MiSFIT, the Minimal i386 Software Fault Isolation Tool, to protect itself from misbehaved extensions. MiSFIT is discussed in detail in Chapter 5.

2.3.3.3 Interpreters

Instead of loading compiled, native code directly into the extensible system, an interpreter takes source code or code in some intermediate form and processes it directly. The code given to the interpreter can be thought of as instructions for an abstract virtual machine, and the interpreter implements that machine. The advantage is that the interpreter can ensure the safety of each command or instruction as it is processed; the disadvantage is that, because each command or instruction is processed by the interpreter, there is typically a much greater overhead associated with interpretation than with compiled techniques. Intuitively, the interpreter can be thought of doing the work of the compiler (converting the source or intermediate form into an executable form), the work of the safety mechanism (by ensuring the safety of each instruction as it is interpreted), and the work of the hardware itself (by evaluating the instructions). The fastest interpreters are ten to one hundred times slower than compiled code [May87]. For example, Java byte codes can be interpreted directly; in Chapter 4, I measure directly interpreted Java running ten to seventy times slower than compiled C.

Applications can also often be extended using simple interpreted languages. For example, the functionality of web browsers can be extended using JavaScript, a simple scripting language.[8] Many Microsoft applications can be extended using Visual Basic for Applications (VBA), a variant of the Visual Basic programming language, and many applications from Lotus Development Corporation can be extended using Lotus Script, a BASIC-like language.

The HiPEC system [Lee94] allows applications to control VM caching policy using programs written in a simple, assembler-like, interpreted language designed specifically for the task of managing a queue of VM pages. The performance impact of executing a program in this language is low, but the expressiveness of the HiPEC language is limited (it has only 20 basic instructions). Although this lack of expressivity has the advantage of ensuring safety, it is not suitable for general-purpose extensions.

Packet filters are used to demultiplex a stream of network packets by examining the contents of each packet header. Packet filters are often implemented in a simple interpreted language [Mogul87]; Berkeley Packet Filters are one example [McCanne93]. Such languages, designed to efficiently describe packet headers, are usually limited to commands that match sections of a packet (to specify which packets to give to the filter) and simple transformations on the contents of the filter. Like HiPEC, the expressivity of packet filter extensions is severely limited (although sufficiently rich for their intended use).

---

8. Other than the name, JavaScript has little to do with the Java programming language.

Campbell et al. proposed extending the μChoices operating system with "a simple flexible scripting language similar to Tcl" to aggregate multiple kernel calls or remove control traffic between user-level and kernel-level [Campbell95]. (Unfortunately, nothing more has been published on this work.)

The Inferno kernel is protected from misbehaved end-user code by requiring that extensions be written in Limbo, an interpreted safe language [Pike97]. Limbo, like Java, is compiled to an intermediate form that can be directly interpreted or translated to native code at load time or run time.

Unlike HiPEC and packet filters, the VBA, Lotus Script, Tcl, and Limbo languages are "real" programming languages.[9] They include control structures, complex data types, and can compute arbitrary functions. This expressivity is the source of their value as extension languages, but also requires a larger, more complex interpreter and more run-time support. Additionally, because each provides its own execution environment (as with safe languages), communication between the rest of the system (presumably written in an unsafe language) and the extension environment can be complex and slow.

In order to reduce the overhead of interpretation, an interpreter can generate native code, either as the extension is loaded or as it is run. A "just-in-time" compiler (JIT) is an interpreter that generates native code rather than interpreting code directly. Code compiled by a JIT has been shown to be much faster than interpreted code; for example, Hölze et al. showed that a JIT for the Self programming language[10] generated code that ran only two to ten times slower than conventionally compiled code [Hölze94] (as compared with interpreted code, which runs ten to one hundred times slower than conventionally compiled code). A JIT has several disadvantages over a conventional compiler, however. First, it is usually working from an intermediate form that does not have as much high-level information about the code as would a conventional compiler, and thus may not be able to perform sophisticated procedure-level optimizations.[11] Second, because a JIT typically translates one function (or a small number of functions) at a time, it cannot perform more powerful inter-procedural optimizations available to a conventional compiler. Third, because a JIT operates in a dynamic environment, it cannot assume that it has complete information about the type hierarchy used by the program. When a program is compiled into a monolithic executable, the compiler can examine the complete type graph and convert what would normally be late-bound (dynamically dispatched) calls into direct calls, thus improving overall performance; such conversions are not possible with a JIT. Fourth, and perhaps most important, because a JIT does its work at run-time, it must weigh the time it spends generating code and performing optimizations against time that might be better spent running the generated code. If a dynamically loaded function is run only once, it may make little sense to spend a lot of effort squeezing the last bit of performance out of it; we should be concerned about the total time required to load, compile, and run the code, not just the time required to run the code. A conventional compiler has the luxury to take more time and do a good job generating code.

2.3.3.4 Verification of Software Protection

Software isolation techniques require that the system running the application/extensions can verify that protection has been applied. The original software fault isolation system [Wahbe93] applies the SFI transformation at extension load time, requiring no other verification. The VINO system

---

9. In the sense that they are Turing-complete.

10. Although linguistically different, the work involved in compiling Self is comparable to that of Smalltalk-80 or Java.

11. This problem can be solved by bundling high-level information with the intermediate form, analogously to how Necula's PCC bundles code safety information with the compiled code.

uses a cryptographic digital signature [Schneier96]. When loading extension code, VINO recomputes the code's signature; if the computed signature matches the signature stored in the extension, VINO is assured that the code was processed by the MiSFIT software fault isolation tool.

A new technique, *proof-carrying-code* (PCC), pairs the extension with a proof of its safety [Necula96]. The proof is generated as a side-effect of the compilation process, and is used as a guide by the system to verify that the code does nothing unsafe. Although techniques for generating formal proofs given a set of axioms are well-known, proof generation is usually computationally expensive. Necula's work puts the burden of proof on the compiler, with the dual observations that much of the work necessary for generating the proof is available to the compiler as part of the compilation process (essentially, as a side-effect of type checking), and that it is often much easier to check that a proof is correct than it is to generate a proof *ex nihilo*.

Instead of using cryptographic signatures, VINO could use a simple variant of proof-carrying code. At extension load time VINO could scan the extension code and verify that no illegal loads or stores were performed. There are a small number of transformations performed by MiSFIT that would require some complexity on the part of the load-time checker; this complexity would be reduced by having MiSFIT generate simple annotations (a restricted form of proof) to guide the loader in its verification of the safety of the loaded code.

It is possible to *partially* decouple safe compilation from extension loading by compiling the extension into a safe intermediate form, as is done (more or less) with the Java programming language [Gosling96, Lindholm97]. Java is compiled to Java byte code, which code can be directly interpreted (where the interpreter can guarantee safety), or translated at run time to native code. If it is translated at run time, a simple verifier can be run to ensure that the byte code is type safe (e.g., it does not push an integer onto the stack and then pop an object reference). In this way, Java byte code is like proof-carrying code; although Java byte code does not travel with a reified proof, the byte code is sufficiently restrictive in its expressivity that verification of the safety of a piece of code can be done efficiently and quickly at program load time.

2.3.4 Summary of Isolation Techniques

An *isolation mechanism* ensures that private operations and state remain private: a client can invoke only the functions and read and write the data elements that are in the public interface. Isolation mechanisms can be implemented in hardware or software; software mechanisms include safe languages, software fault isolation, and interpreters.

As a rule of thumb, hardware techniques have a higher domain crossing cost and a lower per-instruction cost, making them most appropriate when domain crossings are few; software techniques have a lower domain crossing cost and a higher per-instruction cost, making them most appropriate when domain crossings are common and little code is run in the isolated domain. This is at best a guideline: some hardware techniques (e.g., the L4 IPC) have a low domain crossing cost, and some software techniques (such as safe compiled languages) have a low per-instruction cost. The particular choice of technology depends on the specific characteristics of the technologies being considered and the behavior and requirements of the application.

2.4 Transactions

Each time VINO invokes an extension, it wraps the invocation in a transaction. If the invocation fails, any changes the extension has made to the state of the kernel are undone.

The transaction model was developed in the context of database systems. Classical database-style transactions [Gray93] provide the *ACID* properties of atomicity, consistency, isolation, and durability. These properties allow a database server to guarantee that the integrity (internal

consistency) of the system will be maintained, and that the results of a transaction, once committed, will not be lost.

VINO allows extensions to use a subset of its internal kernel interfaces, and to modify kernel state directly. It is possible for an extension to move the kernel into an inconsistent state, and then to fail; if this happens, VINO must put itself back together and return to a consistent state. As an example, an extension may have pinned a page in physical memory: if the extension fails, VINO must ensure that the pinned page is released.VINO uses *VINO Lightweight Transactions (VLTs)*, which provide a three of the four ACID properties of traditional database transactions, to ensure that the kernel is left in a consistent state.

The ACID properties are:

- *atomicity:* An atomic action succeeds or fails completely, as a single unit. If an action commits, all of the changes requested as part of the action take place; if the action aborts, none of the changes take place. This usually refers to failure atomicity, where if a database server fails, the transaction either succeeds or fails as a whole; it is not possible for partial results to be recorded in (exported by) the database server. In terms of extensions and kernels, when an extension commits a transaction, either all or none of the operations requested by the extension as part of that transaction are applied to the kernel state.
- *consistency:* Each atomic action moves the server from one consistent state to another. At the completion of a transaction, any server consistency requirements are met.For example, in the case of an operating system, at the end of a system call, all resources used by the operation are relinquished. Any transient inconsistencies in the state of the kernel are cleared up when the transaction completes. Temporary buffers allocated for use during the transaction are reclaimed, and all other resources needed solely during the transaction are released.
- *isolation:* Clients outside a transaction are isolated from the intermediate changes made in the context of the transaction. For example, if there are two concurrent transactions, A and B, a client running in the context of transaction B will see either the state of the system before A started or the state of the system after A ends. Note how isolation differs from atomicity: atomicity ensures that transactions complete or fail as a whole; isolation ensures that *while running,* changes made within transaction A are not visible by clients outside transaction A.
- *durability:* Once a transaction commits, the results of the transaction can be recovered if the system crashes.

In order to ensure durability, transaction results must be written to stable store, typically a disk or tape. As the cost of a synchronous disk write is typically five to ten milliseconds, techniques have been developed to reduce this cost, such as group commit [Gawlick85], which batches a group of commit records together into a single write;[12] the use of non-volatile RAM, as exemplified by the Rio Vista system, described below; and the use of a looser definition of consistency, as with log structured file systems [Seltzer93], which ensure recovery of a consistent state, although it may not be the final state computed before failure.

Because VINO's kernel state is itself ephemeral, and not retained across a reboot, it is not necessary that VINO transactions provide durability. By eschewing a write to stable store, VINO Lightweight Transactions can commit a transaction in roughly five to ten microseconds, three orders of magnitude faster than a traditional implementation of standard transactions.

These are by no means the only possible variants of the standard transaction model. For example, nested transactions [Moss85] allow a transaction to create sub-transactions whose success or failure does not necessitate the success or failure of the enclosing transaction. (VINO nests transactions when one extension invokes another.) Sagas [Garcia-Molina87] break a long-running transactional process (saga) into a series of shorter transactions that can be interleaved

---

12. First invented and implemented by Gawlick in 1976, later independently developed by DeWitt et al. [DeWitt84].

with transactions of other sagas; each transaction is paired with a *compensating transaction* that can be used to undo the transaction's effects, if the saga is aborted.

As a generalization of the work done with extended transaction models, Chrysanthis and Ramamritham [Chrysanthis90] developed the ACTA framework, a general-purpose model for describing a variety of transaction models. ACTA can be used to categorize and compare different transaction models, and determine the intrinsic differences between them.

Although transactions are traditionally used in the context of a database management system, VINO is not the only non-database system to find it advantageous to use transactions. In the following subsections I discuss the QuickSilver operating system, which used transactions pervasively, and the Rio Vista recoverable virtual memory system, which, through the use of special hardware, offers durable transactions with the performance of VINO's ephemeral transactions.

2.4.1 QuickSilver Operating System

The QuickSilver operating system added transaction protection to the operating system kernel. Each system call ran in a transaction, but there was no intra-system call transaction protection [Schmuck91]. If a system call failed, the kernel was able to return itself to a consistent state by aborting the enclosing transaction, rolling back any changes made by the system call to the kernel's state. Pervasive kernel support for transactions simplified many tasks that are typically handled in an ad-hoc fashion. For example, when a text editor saved the new contents of a file, instead of writing to a temporary file and then removing the original file, the text editor could wrap the file write in a transaction. If writing the file failed, the old contents still existed on disk; if it succeeded, the old contents were eliminated. QuickSilver did not provide a nested transaction mechanism, but the system designers believe, in retrospect, that the presence of such a facility would have been useful.

Where QuickSilver used the transaction system to structure the kernel, exported the transaction system's functionality to user level, and provided transactions for durable operations (e.g., writing a file to disk), the VINO operating system uses the transaction system only to protect itself from misbehaved extensions. It does not (currently) provide transaction support to applications, and does not attempt to support durable operations. In this way, the goals of VINO are less aggressive than those of QuickSilver. However, by making transactions less pervasive, it may have been easier for VINO to support standard POSIX system call semantics; the designers of QuickSilver found that the transaction model at times complicated this task.

2.4.2 Rio Vista and Non-Volatile RAM

Lowell and Chen have developed what they term "free transactions" in the Rio Vista recoverable virtual memory system [Lowell97]. Rio Vista transactions provide the uncommon combination of durability and extremely high performance (5-10 µs) by using non-volatile RAM (NVRAM) for stable storage. NVRAM has the performance characteristics of RAM, but retains its contents in the face of system failure and power loss.

Unlike VINO lightweight transactions, Rio Vista transactions do not provide for any semantics beyond simple undo. Rio Vista uses *physical logging*, which saves the original contents of a region of memory before it is modified in the context of a the transaction. If the transaction aborts, the original contents of the memory region are restored. VINO transactions support the more general technique of *logical logging*, where instead of just logging the original contents of a memory region, the transaction mechanism writes instructions for invoking an undo function to the log. If the transaction aborts, the undo function is called. Logical logging subsumes physical logging, as the undo function could simply copy back the original contents of the memory region. And logical

logging has the advantage of being more flexible; for example, it may be more efficient to recreate the original state of the modified object than to copy the state to the log and restore it on an abort. Since transaction abort is typically uncommon, a transaction system may want to code for the common case, inducing a cost bias, in which less effort is spent logging and more is time spent in undo if the transaction aborts. This model is much more easily supported with a logical logging scheme such as the one used by VINO, than with a physical logging scheme, like the one used by Rio Vista.

Rio Vista is by no means the first system to use NVRAM for the purpose of hiding or eliminating disk latency. For example, the Harp file system [Liskov91] depended on the presence of short-term uninterruptable power supplies on servers, to ensure that servers had time to write committed data from main memory to stable storage when power failed. The Sprite operating system used a region of NVRAM to store critical kernel data structures to simplify and speed system recovery after a crash [Baker92]. A Network Appliance file server [Hitz94] stores its write-back cache in NVRAM, to allow the server to accept synchronous file writes without requiring synchronous disk writes. If the Network Appliance server crashes, unsaved data is recovered from NVRAM.

The advantages of NVRAM could be exploited by VINO's transactions, by allocating VINO's logs in NVRAM. This would allow VINO's transactions to support durability without increasing the overhead of the transaction system. However, as stated above, such durability is not important because the data protected by VINO Lightweight Transactions is itself ephemeral.

2.4.3 Transaction Alternatives

Instead of a transaction-based structuring mechanism, a system can offer an interface consisting of *atomic calls*. An atomic call is executed as atomic, isolated operation and moves the system from one consistent state to another.

System calls are typically implemented as synchronous operations (e.g., disk I/O), where the calling process blocks until the operation is complete, possibly holding kernel resources. In the Fluke operating system kernel [Ford98] such blocking operations have been divided into a sequence of short atomic system calls that do not block for a "long" period of time.[13] User-level applications perform the traditional operations by invoking some number of the short atomic system calls in sequence. With this structure, each short atomic call is an atomic and isolated unit that does not holding kernel resources for a "long" period of time.

For example, on a traditional system, a disk write() system call might consist of two parts, each of which might block for a "long" period of time: the first would ensure that the data to be written was in memory (blocking until it was faulted in, if ncessesary), and the second would write the data out to disk (blocking until it was written to disk). With the Fluke model, there would be five shorter system calls invoked: (1) fault in the data buffer's page and pin it there, (2) wait until the data was in memory,[14] (3) write the data buffer to disk, (4) wait until the data was written, and (5) release the data buffer's pages.

2.5 Summary

There are three problems to face when building an operating system that gives control over resources to applications: ensuring fairness, ensuring safety, and ensuring consistency. The techniques that VINO uses to ensure fairness were discussed above, in Section 2.2. VINO maintains its

13. The authors do not precisely define "long."

14. Although this would cause the process to block in the kernel, it would not require that the process hold any substantial kernel resources.

safety in the face of ill-behaved applications through the use of software fault isolation; my software fault isolation tool, MiSFIT, is discussed in Chapter 5. VINO ensures its consistency in the face of extension failure through the use of a lightweight (ephemeral) transaction mechanism, discussed more fully in Chapter 6.

In the next chapter, I present an overview of the architecture of the VINO kernel.

Chapter 3

The VINO Kernel Architecture

This is a really bad idea quite well implemented and presented.

– Anonymous reviewer of [Seltzer96]

3.1 Introduction

This chapter describes the overall architecture of the VINO kernel, which was designed from the bottom up to support extensibility. Although the techniques used by VINO to support extensibility could be applied to any operating system kernel, there are advantages to designing with extensibility in mind; these advantages are discussed below.

3.2 Overall Architecture

When the extensibility features of VINO are ignored, what is left is a (more or less) conventional, multithreaded, monolithic kernel. Much of the machine-dependent kernel code (boot code, locore, and device drivers) was adapted from NetBSD with few changes. However, unlike NetBSD, VINO is a multithreaded kernel; some effort was expended convincing the machine-dependent portions of NetBSD (which were designed to run in a single threaded kernel) to work with the multithreaded VINO kernel.[1]

The basic process and interrupt models come from Unix. The system consists of a supervisor mode kernel, and a collection of user mode processes. The kernel is divided into a "top half" (which implements most operating system functionality) and a "bottom half" (consisting of interrupt handlers). Code in the "top half" of the kernel is invoked synchronously by user-level processes (via system calls and other traps) or by in-kernel threads. The "bottom half" is invoked asynchronously when an interrupt is received from a device.

The kernel consists of a collection of concurrently running threads. A thread can reside entirely in the kernel, or have a user-level process associated with it. When a thread is run, the kernel switches to that thread's context. If the thread has no associated user-level state, switching kernel threads is light-weight (about as expensive as a function call). If the thread has an

1. Multithreading was introduced to support extensibility, and is discussed in more detail below.

associated process, the context switch is more expensive (as the kernel has to switch page tables, causing a TLB flush). Once running, such a thread can switch freely between supervisor mode (say, if it is processing a system call) and user mode (if it is running user-level code).

3.3 Compatibility

VINO implements the POSIX system call interface, and POSIX applications can be recompiled and run on VINO without modification.[2] An implementation of the Berkeley networking interface (sockets, TCP, and UDP) is underway as of this writing. The currently implemented VINO file system uses the same on-disk format as, and is compatible with, the BSD Fast Filesystem [McKusick96].

VINO currently runs only on x86-based PC-class computers. As VINO uses minimally modified (often unmodified) NetBSD device drivers, VINO supports most common devices found on the PC platform, including IDE and SCSI disk drives, network boards, and video boards.

There were two primary benefits of implementing a POSIX-compatible system call interface. First, we were able to "port" many standard utility programs to VINO simply by recompiling them. Perhaps more important, because we had a ready-made set of applications that used the system in strange and twisted ways that we would not have thought of, we found (and fixed) many bugs that would have otherwise slipped through.

3.4 Support For Extensibility

The VINO kernel is written in C++ and consists of a group of fairly coarse-grained C++ objects, such as threads, virtual address spaces, and open files. Some objects are owned by a single thread (e.g., the saved thread context is used only by the owning thread), others can be shared by a number of threads (e.g., a virtual memory object is shared by all threads that have the object mapped into their address space; an open file object is shared by all processes that have the file open), and some are shared by all threads in the system (e.g., the list of open files and the list of active threads).

The VINO model of extensibility is based on replacing kernel code at the level of a function, and being able to replace this code on a per-process basis. This model affected the kernel implementation in two ways. First, when writing VINO, we factored out into separate functions any kernel code that implemented a policy decision. Second, each function was made a member function of an object.

For example, when NetBSD is servicing a read request from a local disk, it compares the current read location to the previous read location; if they are logically sequential, NetBSD assumes that the process is reading the file contiguously, and will "read ahead," requesting the next few blocks of the file from the disk before the process requests them. The code that implements this behavior is shared by all files stored on a local disk in the system.

In VINO, this policy decision has been moved to the compute_ra() member function of the open file object, and the default implementation of compute_ra() implements essentially the same algorithm that NetBSD implements, giving VINO the same behavior as NetBSD in the common case. The compute_ra() function can then be overridden on a per-open file basis.

Given this structure, the VINO *grafting* architecture comes into the picture: VINO allows a process to load, or *graft*, a replacement for kernel policy functions.[3] The kernel registers all functions that can be replaced, entering them as *graftpoints* in a graftpoint registry. There are both per-process and global registries: the per-process registry holds the graftpoints that a process can

---

2. Some POSIX features have not yet been fully implemented, but most common Unix utilities run out of the box, including sh, ls, cat, grep, make, gcc, awk, and vi.

graft, and the global registry contains the graftpoints on global objects that can be grafted only by privileged processes.[4] Objects that are shared by multiple processes, such as open files and virtual memory objects, can be grafted only by the process that creates the object.[5]

VINO provides system calls to allow processes to manipulate grafts. A process can look up a graftpoint using the graftpoint's name; this returns a graft handle (opaque pointer). With a graft handle, the process can load a new function at the graftpoint, passing the name of a file that contains the code for the function, or it can unload the graft loaded at that graftpoint, returning to the default behavior of the system.

When a process asks the kernel to graft a function, the graft's code is dynamically linked directly into the kernel's image. At load time, the kernel performs several checks. It verifies that the new code has been processed by MiSFIT, the VINO software fault isolation tool. It also examines the list of symbols that the newly grafted function links against, and ensures that they are all on the list of symbols exported by the kernel for use by grafts.

Most graftpoints are unique to a process, and have simple names; for example, the page replacement graftpoint on a process' virtual address space is "vas_o::pick_victim." Other graftpoints must be disambiguated; the read ahead graftpoint for an open file is based on the file descriptor (number) assigned to the file, and is named "open_file_o::compute_ra/*n*" for the file with descriptor number *n*.

The VINO kernel is multithreaded, so that when the kernel is running grafted code, it is not blocked from doing other work. If a process grafts a function that does not terminate quickly, but does not block the progress of other processes, VINO lets the grafted function run; the time allocated to the grafted function is deducted from the time allocated to the process, and so the process is only hurting itself by loading a graft that does not terminate. However, a graft is only allowed to block the forward progress of others for a limited time: if a graft acquires a lock, and others are waiting for the lock, the graft is given a time limit. If the graft exceeds its time limit, the graft is terminated, so that others may move forward. (As a side-effect of being terminated, the lock, and any other resources allocated to the graft, are freed. This is explained in more detail in Chapter 6.)

3.5 Events

VINO's function grafting model, described in the previous section, works well when the goal is to replace functionality that is bound to a particular kernel object (e.g., a policy). However, there are other kinds of extensions, not well-served by this model, that a user may wish to write.

One example is an extension that is designed to respond to a subset of a stream of incoming (external) events, such as network packets or user activity. We may not wish to give the extension all events as they arrive (for privacy or performance reasons), and so we need to filter the event stream, passing on only the events that meet the appropriate criteria. (This is the event model used by the SPIN extensible operating system [Bershad95]; more on SPIN is found below.)

Filtering for *privacy* ensures that an extension that has permission to, say, listen on one TCP socket is not able to view all incoming TCP packets.

Filtering for *performance* would be done if an extension wants to see only a subset of some type of event, and we do not wish to pay the overhead of invoking the extension on all events of that type. For example, consider an application that is responsible for propagating updates on a set

---

3. Although the discussion above refers to threads, here we refer to processes, to focus on the point that these system calls exist for user-level processes to safely change the behavior of the kernel. If a kernel-level thread wishes to change the behavior of the kernel, it can do so without going through this rigmarole.
4. Here, privileged processes are analogous to root processes on Unix.
5. This restriction is one way to handle the case of shared objects; obviously there are other possible ways.

of files stored on the local disk. This application could run entirely at user level, polling regularly to check whether any of the files on its list have been updated. Alternatively, the application could load a graft into the kernel that would be invoked each time any local file is modified, and check to see if the modified file is on its list. To save the cost of invoking the graft on every file modification event, we could filter the event stream and only invoke the graft when a modification event is generated for a file in which the extension is interested.

VINO provides graftpoint-specific event filtering support in the networking subsystem, the file system, and the timing subsystem. The filtering mechanism is subsystem specific; the network filters on (protocol, port) pairs, the file system filters on file identifiers, and the timing subsystem filters events based on how much time has passed since the last accepted event.

In the networking system [Ghosh98], the packets themselves are events. The network stack is modeled as a graph of protocol objects that communicate via *push* and *pop* functions. For example, when the *IP* protocol object has processed an inbound packet and is ready to hand it off to the *TCP* protocol object, *IP* calls the *push* function of *TCP*. An application can instantiate a new protocol object (e.g., *My-TCP)* and specify that TCP packets addressed to a particular port be directed to that object. The application can then graft the *push* and *pop* functions on the application-specific protocol object.

The file system generates an event whenever a file is opened, closed, created, deleted, or modified. Filters consist of an event type (open, close, etc.) and list of file names (in the case of create events) or inode numbers (in the case of the other event types). When the file system generates an event, it examines the filters for that event type; if an event is accepted by a filter, the corresponding graft is called.

The timing system generates an event on every clock tick (10ms). A timing filter specifies how many ticks to wait between accepting events.

In contrast, in the SPIN operating system, each procedure call is considered an event, and extensions are modeled as handlers for these events (corresponding to VINO's function grafts, described above). When an extension is loaded, it registers itself for one or more events, optionally specifying guard predicates to filter events (corresponding to VINO's event filters). Unlike VINO's filters, which are data-driven, SPIN's event filters are written in an imperative safe language (Modula-3). The SPIN model is more general than VINO's in that the VINO model could be implemented in SPIN's model (where the filters would check port numbers or inodes), but the SPIN model could not be implemented in the VINO model (as VINO currently only allows filters to be installed on specific grafts).

3.6 Summary

Although VINO reuses low-level code from NetBSD (device drivers and bootstrap code), the VINO kernel was designed from the bottom up to support extensibility. The decomposition of the kernel into objects, with policy decisions factored out into member functions, simplifies the addition of graftpoints to the system.

Chapter 4

Comparison of Extensibility Technologies

> *List every recipe ingredient (including garnishes) with exact measurements and complete directions.*
>
> *– Pillsbury Bake-Off Official Rules*

4.1 Introduction

This chapter gives the results of an extensibility technology comparison study, designed to determine which technology was best suited for use with the VINO project. The original study was done in 1995, when the VINO kernel was first designed. Although hardware platforms have changed considerably since 1995, most software technologies have not. At the end of the chapter, I discuss the ways in which the technologies have changed, give results of rerunning the tests with the technology that has changed the most, and discuss how current results might change our choice of technology were we to redesign VINO today.

At the time of the original study, the group was designing VINO and wanted to choose the best technology for supporting extensibility. Deciding *how* to choose the best technology was the first task. Our goal in building VINO was to allow application developers to write extensions that would improve application performance: hence the extensibility study became a performance study. However, raw *extension* performance was not our goal; rather, we sought to improve *application* performance. In order to learn which extension technologies were appropriate to improve application performance, I computed a *break-even point* for each benchmark extension. The break-even point was the point at which the cost of running the extension was equal to the gain in application performance. When an extension, implemented in a given technology, would improve application performance, I considered the technology sufficient to the task; if the extension would degrade application performance, I deemed it unfit.

To develop meaningful benchmark programs, we brainstormed as a group to develop a list of grafts that we wanted VINO to support. The grafts fell into three categories, based on their programmatic structure: *prioritization* grafts (which choose from a set of alternatives), *stream* grafts (which transform a data stream), and *computation* grafts (which compute a function of their inputs).[1] Although there are any number of ways to structure an extension, each of the example grafts (presented below) fell into one of these three categories.

As discussed earlier, there are a number of different isolation techniques available. I chose one representative of each technique (hardware, safe language, software fault isolation, and interpreter) and implemented each example task in the representative technique. I ran the tests on a variety of hardware platforms, in order to control for hardware idiosyncrasies.

4.2 Taxonomy

In this subsection I introduce the three extension structures that was discovered as part of the study. A prioritization graft (Section 4.2.1) selects a candidate from a list of options, a stream graft (Section 4.2.2) process a stream of data as it passes through the kernel, and a computation graft (Section 4.2.3) computes a function of its inputs and current state.

4.2.1 Prioritization Grafts

There are numerous places in a kernel where one of a set of entities is selected. I call this a *prioritization* policy decision, and a graft that replaces prioritization policy code a *prioritization graft*. Examples include:

- *virtual memory system:* which page to evict.
- *buffer cache manager:* which buffer to evict.
- *process scheduler:* which process to schedule next.
- *disk driver:* which disk request to schedule next.
- *network driver:* which pending packet to service next.
- *interrupt handler:* which pending interrupt to service next.

Normally, the VM system and the buffer cache use a least-recently-used (LRU) policy (or a variant thereof), although in some cases a different policy works better. An application might know that each block of a file will be read once, in order, and not read again, in which case it makes sense to use a most-recently-used (MRU) strategy for that file. As another example, a database system processing a query may know which blocks of the database will be needed soon and which blocks are no longer needed.

Use of the recent access pattern of a set of pages is often a good heuristic for deciding which pages will be used in the near future, but it is not an infallible one. For example, a compacting garbage collector moves objects around in memory with the goal of decreasing memory fragmentation. During a compaction, the collector accesses partially-filled pages, but ignores pages that are completely full. Using an LRU paging strategy, least recently used pages (e.g., pages that are full of live objects) will be evicted in favor of keeping recently accessed pages in memory (e.g., pages that the collector has emptied during the compaction). This is the opposite of what is desired.

Process scheduling is another example of a prioritization policy. At each scheduling point the kernel has a list of candidates, and chooses one to run. No scheduling algorithm is appropriate for all application mixes; the demands of interactive applications differ from those of applications with real-time deadlines, and a scheduler optimized for one class will not satisfy the other. Processes may wish to be scheduled as a group: for example, a multimedia application with separate processes running the audio display, the video display, and the audio input, may want to schedule the processes to ensure that the audio input process is scheduled frequently enough that no data is lost, that the audio output process is scheduled frequently enough to ensure continuous playback, and that the video process is given whatever time is left.

---

1. As stated in Chapter 3, I refer to a kernel extension as a *graft* and the process of adding a graft to the running kernel *grafting*.

In general, a prioritization graft is one that is presented with a list of options and must select the item of highest priority. It uses some internally-defined weighting function to choose a candidate. The weighting function may use just the information in the list or it may have access to some data from the application (e.g., which blocks or pages will not be needed again).

The prioritization graft benchmark models a VM page eviction policy. I assume that the kernel maintains a list of pages in LRU order; when it comes time to evict a page, the kernel normally chooses the page at the head of the LRU queue as its candidate. In this model, instead of immediately evicting the candidate, the kernel determines which process owns the candidate page and allows the process to offer one of its other resident pages for eviction. The model application keeps a “hot list” of pages that will be needed in the near future. The goal of the page eviction graft is to ensure that none of these pages are evicted.

The page eviction graft receives a pointer to the head of the LRU queue. The graft checks to see if the candidate (the head of the queue) is on the application’s hot list; if it is not, it accepts the kernel’s candidate. If the candidate is on the hot list, the graft searches through the queue for an acceptable page that is not on the application’s hot list. This page is returned to the kernel.

For this benchmark I model a TPC-B transaction processing benchmark database [TPC90]. The database holds 1,000,000 records in a four-level B-tree; the B-tree has approximately 400 internal pages (1.6MB) and 50,000 leaf (data) pages (200 MB). The b-tree is 50% full, and has one root page, four pages at the second level, 391 pages at the third level, and approximately 50,000 pages at the fourth level; each third-level page in the b-tree points to up to 128 fourth level pages.

The server accesses the database by mapping it into its virtual address space. When the server does a non-keyed lookup, it traverses the b-tree in depth-first order, starting from the root. When it reaches a third-level page it knows which 128 fourth-level pages it will access next, and hence which of the memory mapped pages of the database should not be paged out. During such a search, it constructs a hot list of these 128 pages. If a page fault occurs during a search, our graft is called with the LRU chain head, and it uses the hot list to find an eviction candidate that is acceptable to the application. As each page is processed, its entry is removed from the hot list, so as the simulation runs, the queue grows shorter. I presume that the kernel keeps track of candidate pages and graft-proposed alternates, as in Cao’s system [Cao94], to ensure that an application does not manipulate the VM system to gain more physical memory than it would receive under the default strategy.

This test is not compute-intensive. Instead, it is sensitive to the overhead associated with traversing a list of items. If the extension technology requires extensive pointer checking, or does not support traversal of linked lists, this test will highlight it.

Because there are 50,000 leaf pages in the database, there is a low probability that a page that the application will need in the future is already in the cache (roughly 64/50,000, or once every 781 times). However, if one of the pages is in memory, I want to ensure that it is not evicted. For the graft to be successful, the overhead of checking on each eviction should be low relative to the cost of evicting and then re-faulting a page. The results (presented in Section 4.3.4) include the break-even point, showing how often the graft will need to save an eviction in order to pay for its per-eviction cost.

4.2.2 Stream Grafts

The second graft model is based on the idea of Unix filters and pipes. A *stream* graft consists of filtering code that is inserted into a data stream. Examples of stream grafts are:

- *file compression/decompression:* compress a file as it is written to disk; uncompress when it is read.
- *file encryption/decryption:* encrypt a file as it is written to disk; decrypt it when it is read.

- *packet encryption/decryption:* encrypt the payload of a packet as it is sent to the network, decrypt packets as they arrive.[2]
- *network tunneling:* encapsulate packets generated with protocol P in packets of protocol P' for transmission over a P' network.
- *security:* compute a cryptographic checksum (*fingerprint*) of an executable when it is loaded, to verify that it has not been compromised (e.g., by a virus).
- *journaling file system:* as metadata changes are made to the primary file system, save a log of the changes to a separate, journal file system (for rapid recovery after failure).

The Stream Input-Output System of UNIX [Ritchie84] decomposed the character I/O system of UNIX into a set of filters. Network and terminal protocols were built up by linking filters into chains. Characters read from (or sent to) a device were passed to the first filter in the chain; each filter processed the characters in its input queue and moved them on to the next filter in the chain. This mechanism was used not only to handle erase and kill processing, but also to create pseudo-devices, such as virtual displays and keyboards, so as to construct multiple virtual terminals from a single terminal.

Another example of a stream graft is one that takes a data source (e.g., the disk) and writes it elsewhere (e.g., the network). Research into fast path connections, such as the *x*-kernel work at Arizona [Druschel93], the video server benchmark of the SPIN operating system [Bershad95], and Fall's work in decreasing I/O time through the use of in-kernel copying [Fall93] show that there is a substantial performance gain from saving copies to and from user-level. A stream graft that takes its input and directs it to an output connection, perhaps after transforming the data, could be used to build this type of fast path connection.

Our representative stream graft is an implementation of the MD5 Message-Digest Algorithm [RFC1321], which produces a 128-bit fingerprint of a file. The MD5 fingerprint is both expensive to compute and computationally infeasible to forge. MD5 is useful for ensuring that a file has not been tampered with; a change to the contents of the file will result in a change to the fingerprint. If the fingerprint is kept separate from the file (say, on a small amount of safe media, such as a read-only floppy disk), a change to the file can be detected by computing its MD5 fingerprint and comparing it to the saved fingerprint.

Like many compression and encryption algorithms, MD5 is stream-based. The algorithm maintains a small amount of state as it processes the data; unlike compression and encryption algorithms, the data output is the same as the input; when the algorithm completes, the graft can be queried for the fingerprint, and the computed fingerprint can then be compared to the saved fingerprint.

If the MD5 fingerprint can be computed as quickly as data can be read from the disk, the time spent in the MD5 code can be overlapped with I/O activity. However, if it takes longer to compute a fingerprint than it takes to read the data from the disk, the overall processing time will increase. Our test measures whether a graft written in a given technology can keep up with the disk.

4.2.3 Computation Grafts

The computation graft structure is more general than that of a prioritization or stream graft. A *computation* graft has some number of inputs and some state, and computes a single output. For example:

- *access control:* accept a triple containing an access request, a user ID, and a resource ID, and respond "permit access" or "deny access."
- *read-ahead:* determine the blocks of a file to prefetch (see below).

---

2. Note that in a client-server environment, a user may want to automatically encrypt files as they are written to a shared (untrusted) file server, but not bother to encrypt files written to the local disk.

- *file name redirection:* take a file path name and resolve all or part of it. (We may want to insert names into the file name space that are redirected elsewhere in the file system, or to a system service [Bershad88].)
- *disk block redirection*: map logical block numbers to physical block numbers (see below).
- *scavenger control:* compute, based on the state of the system, whether to run a scavenger (pageout daemon, log-structured file system cleaner, or other garbage collector).

A Logical Disk facility (LD) [deJonge93] sits between the filesystem and the physical disk. The filesystem reads and writes logical blocks, and the LD maps the logical requests to locations on the physical disk. The LD can be used to transparently replicate data, by writing it in multiple places on the same disk or multiple disks, and speed write performance, by writing logically discontiguous blocks on a physically contiguous region. A log-structured file system [Rosenblum92] can be implemented using a logical disk facility; the filesystem lays out blocks as it sees fit, and the Logical Disk reorders and buffers writes to improve write performance.

Our computation test application is a simple logical disk facility that converts random writes to sequential writes. It accepts block write requests, batches them into physical segments, and maintains a mapping from logical block numbers to physical block numbers. If the savings in I/O time due to batching are greater than the cost of translating logical block numbers on each read/write, the graft is effective.

4.3 Performance Analysis

Given the wide range of extension technologies available, it is not obvious which is “best” in any dimension. In fact, for each technology described, there exists an extensible system that employs that technology. In this section, I measure and analyze the performance of our sample extensions.

Note that the hardware platforms and extension technology implementations listed here are circa 1995. At the end of this chapter I include updated results for Java, which have changed substantially as the technology has greatly matured.

4.3.1 Hardware Platforms

I ran tests on four hardware platforms: three RISC workstations and one Pentium-based PC.

- *Alpha:* DEC AlphaStation 400 4/233 (233MHz), running DEC OSF/1 v3.2A, 64MB of memory.
- *HP-UX*: HP PA-RISC 9000/735 (99MHz), running HP-UX A.09.03, 80MB of memory.
- *Linux:* Pentium (90MHz), running Linux 1.1.95, 16MB of memory.
- *Solaris:* Sun SPARCStation 20 (75MHz), running SunOS 5.4, 128MB of memory.

4.3.2 Extension Technologies

I implemented the grafts in C (as a baseline) afirstnd with four different safe extension technologies. Not all tests were run on all platforms; for example, the Omniware compiler ran only on Solaris.

- *C:* compiled with gnu C, `gcc -O` (version 2.6.3 on HP-UX, 2.7.0 on all other platforms).
- *Java:* release Alpha 3.
- *Modula-3:* version 3.5.3 of DEC SRC Modula-3.
- *Omniware:* compiled with Colusa’s omniC++ compiler, version 1.0 beta, release 1.5. (This compiler generates machine-independent code that is translated to native, software fault isolated code at runtime. This pre-release version of the compiler supports write and jump protection, but does not support read protection, and does not include an optimizer for the SFI instructions.)

- *Tcl:* Tcl version 3.7.

4.3.3 Upcall Overhead

As a baseline for comparison, I implemented each benchmark in C, giving us a theoretical minimum extension running time. However, C is not a safe language; in order to run extensions written in C, one would need to use a hardware protection scheme, requiring some form of upcall. In order to make an apples-to-apples comparison between the software protection technologies and hardware protection, we need an estimate of the cost of hardware protection.

If an upcall takes no time, the performance of a system written using hardware protection would be equal to that of one with unsafe C linked into the kernel. As the cost of an upcall increases, the performance of a system written using user-level servers will decrease.

The cost of an upcall is very dependent upon one's definition of an upcall, and the hardware and software architectures of the underlying system. For example, on comparable hardware one can compare an IPC time of 0.73µs (the L4 microkernel on a 166MHz Pentium [Leidke97]) with a process context switch of 10µs (Linux 1.3.28 on 120MHz Pentium). In this case, the IPC performs little work and assumes an optimal cache layout; the context switch is a full Unix-style process context switch. With comparable software on different hardware, the time to context-switch between processes ranges from 14µs (Solaris 5.5, 167MHz UltraSparc) to 36µs (Solaris 5.5.1, 133MHz Pentium Pro).[3] In reality, the best one can do is to provide a range of possible upcall times.

Using context switch times as the estimate of upcall times is very conservative. Work in the literature showing improvements to exception handling times [Thekkath94] and cross-domain procedure calls [Ford94] makes it clear that it is feasible to implement an upcall mechanism for an extensible operating system that takes less than half the measured context switch time (although it is difficult to imagine achieving the order of magnitude speedup seen in L4 in day-to-day operation).

There are other ways to model upcall times. One is to consider the operations performed by the kernel when doing an upcall, and see if an existing kernel service performs the same operations; this service could then be used to obtain an upcall performance estimate. The work done by the *signal delivery* service is a good approximation of that done by an upcall.

When the kernel makes an upcall to a user-level server, it pushes a new call frame on the server's stack and switches to the server. When the server is done handling the upcall, it returns to the kernel. This is similar to (but simpler than) the work done by the kernel when it posts a signal to a process. First, the kernel pushes a call frame (for the signal handler) on the application's stack, then schedules the application process. When the signal handler is through, the process re-enters the kernel. The kernel cleans up process state and returns the process to the point where the signal was raised.

To estimate the cost of an upcall on each platform, I measured the time required to handle a signal sent to a process.[4] The test program forks a child process, which registers handlers for a group of twenty signals and then suspends itself (by sending itself `SIGTSTP`). When the parent is notified that the child is suspended, it posts the handled signals to the child, then wakes it (by sending it `SIGCONT`). The child awakes, handles the signals, and suspends itself again. When the parent is notified that the child has once again been suspended, it knows that all signals have been handled.

---

3. All context switch times are from Table 10 of McVoy's 1996 paper, and are for switching between two 0KB processes [McVoy96].
4. The design of this experiment was proposed by David Black.

I then measured the time to post the signals to the child when the child ignores (rather than handles) the group of signals. The latter time is subtracted from the former; the result is divided by the number of signals handled, which gives an estimate of the time required to handle a single signal. The results of this test are shown in Table 1.

	Signal Handling Time
Alpha	19.5µs (7.5%)
HP-UX	25.8µs (1.4%)
Linux	55.9µs (0.1%)
Solaris	40.3µs (3.8%)

Table 1: Signal Handling Time

Difference between time required to received and handle twenty signals and time required to receive and ignore twenty signals. The difference is divided by the number of signals to give a per-signal handling time. Each time is the mean of thirty runs of 1000 iterations each (σ in parenthesis).

To calibrate these estimates, I implemented and measured the performance of a simple upcall mechanism on BSD/OS 2.0. On a 486DX2-66 I measured a signal handling time of 63.1µ, and an upcall time of 37.2µs, about 60% of the time required to handle a signal. This implies that the signal handling time is a conservative measure of a practical upcall time, supporting the assumption made above.

4.3.4 VM Page Eviction

As described in Section 4.2.1, our sample prioritization graft takes the LRU-ordered list of page eviction candidates and returns a candidate not on its hot list.

We assume that the application keeps the hot list (of active pages) in its memory at a known location, so that it is accessible to the graft. In addition, we assume that the application keeps the hot list in a form that can be easily traversed by the graft; for example, the C graft searches a linked list of structs, where the Modula-3 graft searches a linked list of Modula-3 RECORDs.

In the model application, the hot list starts out with 128 entries (the number of data pages referenced by a level-three internal page), and as each page is used, it is removed from the hot list. So, on average, the hot list contains 64 pages, which is the number used in the hot list simulation.

The time to check the 64 element hot list in each of the supported technologies is presented in Table 2. To give an intuitive feeling for the performance of each technology, I show the time normalized to unsafe C code.

		C	Interpreted Java	Modula-3	Omniware
Alpha	raw	2.9µs	—	3.2µs	—
	normalized	1.0		1.1	
	break-even	8655		7843	
HP-UX	raw	6.0µs	159µs	6.8µs	—
	normalized	1.0	26.5	1.1	
	break-even	2983	113	2632	
Linux	raw	3.7µs	237µs	9.1µs	—
	normalized	1.0	64	2.5	
	break-even	1270	20	516	
Solaris	raw	4.5µs	141µs	6.3µs (2.8%)	6.3µs
	normalized	1.0	31.3	1.4	1.4
	break-even	1533	49	1095	1095

Table 2: VM Page Eviction Test

Time required to search a 64 element "hot list" of page numbers. Raw times and time normalized to unprotected C code (on the same platform) are given. The break-even point is the number of times the graft can run in the time it takes handle a page fault. Each raw time entry is the time to perform one search, based on the mean of 30 runs of 100,000 searches. $\sigma < 2\%$, except where noted.

To determine the break-even point for this graft, I measured the page fault time on each of our test platforms (as shown in Table 3). The times listed indicate how long it takes to handle a page fault event, not how long it takes to bring in a single faulted page; Alpha and HP-UX bring in multiple disk pages on each fault.[5] We assume that the systems that read multiple pages are performing read-ahead in order to take advantage of (expected) locality of reference. However, our model database server would not benefit from this policy, as the faulted data pages are scattered throughout the database. (The page fault read-ahead policy exhibited here is an obvious candidate for grafting; if one is able to control how many pages the system brought in on a fault, the per-fault time can be reduced.)

	Fault Time	Pages	Time Per Page
Alpha	25.1 ms (5.0%)	16	1.6 ms
HP-UX	17.9 ms (0.8%)	4	4.5 ms
Linux	4.7 ms (0.5%)	1	4.7ms
Solaris	6.9 ms (3.2%)	1	6.9 ms

Table 3: Page Fault Time

Measured using *lmbench v1.0* [McVoy96]*;* σ in parenthesis. Alpha and HP-UX bring in more than one disk page on a fault, performing read-ahead, even though the test performs random accesses to memory.

5. The number of pages faulted was determined by running the page fault test on an otherwise unloaded system while watching the output of iostat and vmstat.

Once the page fault time has been computed, we can determine the break-even point for this graft. We divide the page fault time by the time required to run the graft; the result is the number of times we can run the graft for each page eviction saved and still be ahead of the game.

Remember that our model application would find a page to save, on average, once every 781 invocations. If the break-even point is less than this, our model application would not benefit from this graft. Worse yet, if the number is less than one, the amount of time to run the graft is greater than the page fault time, hence under no circumstances would the graft be beneficial.

In Table 2, we see that the relative times of Omniware and Modula-3 on Solaris are quite close, each running about 40% slower than unprotected C. (Remember, however, that this version of Omniware does not include read protection, which gives it a performance advantage over Modula-3.) On Alpha and HP-UX, the Modula-3 code runs 10% slower than the C code, which is not a significant difference for this test.

On Linux, we see a 150% slowdown for Modula-3, a greater difference than we see on other platforms and with other tests. Examining the code generated by the Modula-3 compiler, I found that it includes a runtime check against NIL (location zero) on each pointer access. The code generated on the other platforms (Solaris, Alpha, and HP-UX) does not include explicit NIL checks. The Modula-3 language specification [Nelson91, p. 50] states that dereferencing NIL should cause a runtime error; on Solaris and Alpha, dereferencing location zero causes a segmentation violation, which is trapped by the Modula-3 runtime system. This is not the case on Linux, so the runtime check is needed. (I found that although runtime checks not are generated by the HP-UX version of the Modula-3 compiler, dereferencing location zero does *not* cause a segmentation violation; this was later confirmed by the compiler's maintainer as a bug in the HP-UX Modula-3 compiler.)

For our purposes (i.e., operating system extensions), the Alpha and Solaris slowdowns are the more appropriate comparison: since we can ensure that dereferencing location zero causes a fault, we would not need runtime NIL checks. (This fault would not be handled by a signal mechanism in the kernel, but would instead require some support by the normal kernel fault logic.)

On Solaris, the interpreted Java code runs at about 1/30th the speed of compiled C code, and about 1/20 of Modula-3. On this platform we see that the break-even point for Modula-3 is about 1100 pages; for interpreted Java, the break-even point is about 50 pages, too low to benefit our model application.

To compare these results to the overhead of performing an upcall on each eviction, Figure 1 shows the maximum number of graft invocations per page fault possible, given a specified upcall time.[6] The page fault times were taken from Table 3, and are not normalized (e.g., on Alpha, the time used is 25.1ms, for 16 pages, and not 1.6ms, the per-page fault time). On machines where the page fault time is large relative to the graft run time, the upcall time necessary to meet the break-even point for this test seems more achievable (e.g., 29.2µs on Alpha) than it does on machines where the page fault time is small relative to the graft run time (e.g., 4.3µs on Solaris). In the future, as processor speed is increasing more rapidly than disk speed, upcall time requirements will become more liberal, and hardware protection will be more attractive. Given the properties of the systems tested here, however, a hardware technique would not be competitive in most cases.

Measurements of Tcl, an interpreted language, showed that it is four orders of magnitude slower than the compiled languages (C and Modula-3), taking 40ms on Solaris, as compared with the C version, which took 4.5µs. With a break-even point close to or less than one (i.e., the Tcl code would have to save a page each time it was called in order to not degrade performance), Tcl is not the appropriate tool for this job.

---

6. Thanks to Dawson Engler for suggesting presenting the break-even time as a function of upcall time.

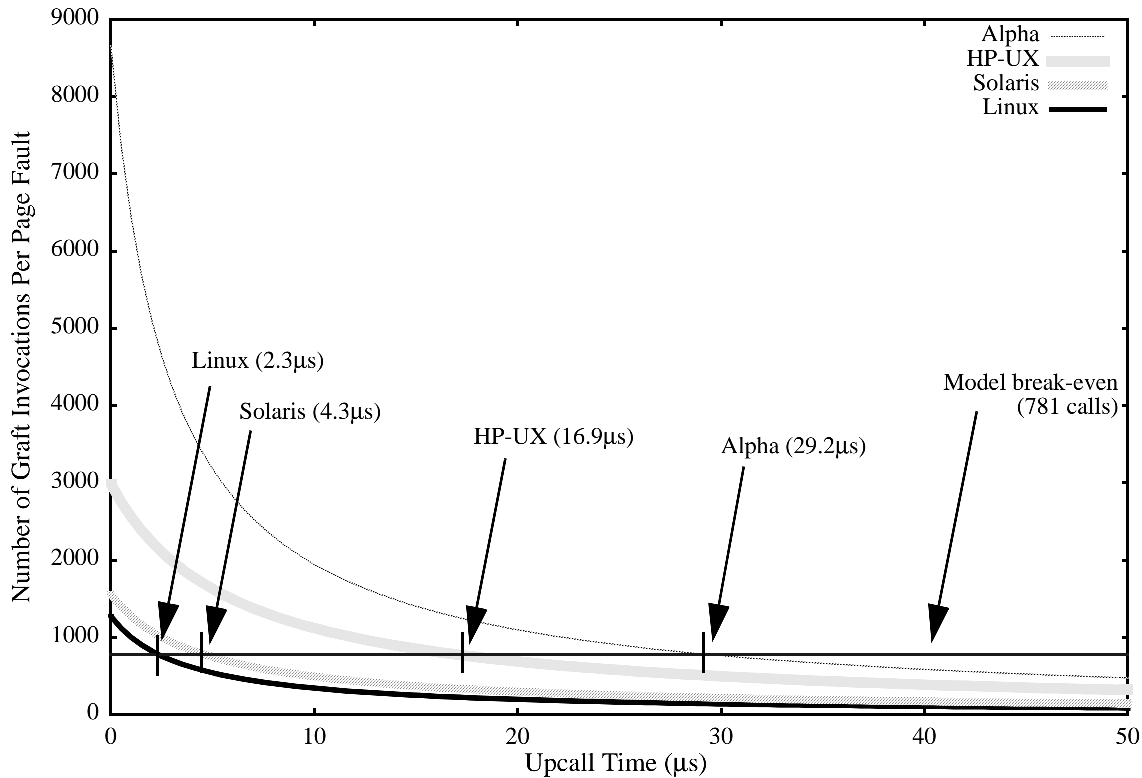

Figure 1. Graft Invocations Per Page Fault vs. Upcall Time

For a given upcall time cost, the number of times that a graft written in C can be invoked, per page fault. We need to be able to invoke the graft 781 times per page fault for this application, in order for graft to break even. Note that because the Alpha platform has very long page fault time (25.1ms) and fast graft run time (2.9µs), we need only a 29.2µs upcall time to break even (which may be achievable). Solaris, with a much faster page-fault time (6.9ms) relative to its graft run time (4.5µs) requires a much more aggressive upcall time (4.3µs) in order to break even for this application.

4.3.5 MD5 Fingerprinting

Starting with the reference implementation of the MD5 algorithm (found in RFC1321), I modified or reimplemented it for each of the test platforms. The MD5 algorithm makes heavy use of array access and unsigned 32-bit arithmetic. The reference implementation takes advantage of the fact that C code silently ignores numeric overflow, and, when it occurs, performs computation modulo 2^{32}. The code worked correctly in C on all platforms, and was easily translated into Java and Modula-3.

This test is compute intensive. It uses almost none of the support facilities of the extension technology (e.g. no data types other than array), but instead gives an indication of the cost of performing array access and numerical computation. The results are shown in Table 4.

		C	Interpreted Java	Modula-3	Omniware
Alpha	raw	159ms	—	207ms	—
	normalized	1.0		1.3	
	MD5/disk	0.67		0.9	
HP-UX	raw	239ms	23987ms	352ms	—
	normalized	1.0	100	1.5	
	MD5/disk	0.43	43	0.64	
Linux	raw	202ms	22887ms	387ms	—
	normalized	1.0	113	1.9	
	MD5/disk	0.33	38	0.64	
Solaris	raw	146ms	10368ms	294ms	219ms
	normalized	1.0	71	2.0	1.5
	MD5/disk	0.46	32	0.92	0.68

Table 4: MD5 Fingerprinting

Mean time required to compute the MD5 fingerprint of 1MB of data. The time is compared to the time needed to read 1MB from the disk. If this number is less than one, the computation of the fingerprint can be overlapped with I/O. If it is greater than one, computing the fingerprint will decrease throughput. The mean of 30 runs is reported. $\sigma < 2.5\%$.

Because the cost of computing the fingerprint is so high, the cost of an upcall is comparatively low. If we assume that there would be one upcall to the user-level extension for every 8KB read from disk, we would need to add the cost of 1MB/8KB = 64 upcalls. If we assume a (very pessimistic) upcall time of 50µs, this would add 32ms to the raw C time, which is not significant when added to compute times of 150ms to 800ms.

To compute a break-even point, I measured the write bandwidth of the disk on each system (Table 5). This measurement was used to estimate the time required to read 1MB of data. If the time needed to compute the checksum of 1 MB of data is less than that required to read the data from the disk, computation time can be hidden by I/O time. (This optimistically assumes that the disk read requires no CPU cycles and is thus a best-case break-even calculation.)

	Bandwidth (KB/s)	1 MB Access Time
Alpha	4364 (1.2%)	235ms
HP-UX	1855 (13%)	552ms
Linux	1694 (5.7%)	604ms
Solaris	3126 (11%)	320ms

Table 5: Disk I/O Time

Write bandwidth in KB/s on each platform, measured using *lmbench*. From this, the time to access 1MB of data is computed. Each time is the mean of 30 runs. σ shown in parentheses.

For this test, Omniware is faster than Modula-3. To see if the size of the test was affecting the results, I ran a larger test (64MB) and saw a similar overhead (14480ms for Omniware vs. 9498ms

for C, an overhead of 50%). I believe that this overhead is representative of this implementation of Omniware for MD5. (Once again, this version of Omniware does not include read protection, which gives it a performance advantage over Modula-3.)

Unlike C, Modula-3 does not ignore arithmetic overflow. The Modula-3 Word package supports computation modulo the native wordsize, so on the 32-bit platforms (HP-UX, Linux, and Solaris) I was able to implement MD5 efficiently. On the 64-bit Alpha, the Word package performs computation modulo 2^{64}, which produces incorrect results for MD5. I implemented two versions of MD5 on Alpha: one uses 64-bit integers and the Word package, doing roughly the same amount of work as the 32-bit implementations (but computing an incorrect checksum), and the other using 32-bit integers, which produces the correct checksum, but generates more instructions (by a factor of four). For our performance measurements I used the 64-bit version, which gives a more accurate comparison of the relative performance of the technologies. I found that the 32-bit version took approximately ten times as long to run as the 64-bit version. (This slowdown is an artifact of the Alpha compiler implementation, and should not reflect negatively on the Alpha processor or on Modula-3.)

The Modula-3 code runs from 1/2 to 3/4 the speed of compiled C. There is no reason for the numerical computation to be slower; the difference is attributable to run-time array bounds checking.[7] On all platforms, the time required to compute the fingerprint was less than the time to read the data from the disk, so a Modula-3 implementation of MD5 could keep up with disk access and overlap its computation with I/O time.

On the other hand, neither interpreted Java nor Tcl were able to keep up with the disk. The interpreted Java code, at best 1/30th disk speed, seems unlikely to be used for this application. And, as above, I found that our Tcl implementation was four orders of magnitude slower than one written in a compiled language, and hence too slow for this type of graft. (On Solaris it took 50 minutes to complete, as compared with 1.9 seconds for C code.)

4.3.6 Logical Disk

Our simulation models a logical disk designed to support a log-structured layer between a filesystem and the physical disk. The simulation accepts write requests for logical blocks and maintains the mapping between these logical blocks and the physical blocks onto which they are stored. As with the system implemented by de Jonge et al. [deJonge93], our simulation maintains all data structures in main memory.

In this experiment, I simulated a 1 GB physical disk with 4 KB blocks and 64 KB (16 block) segments. The simulation uses a stream of block write requests that are skewed so that 80% of the requests are for 20% of the blocks. Because the simulation does not include a cleaner (which

---

7. When the test is run without these bounds checks, the performance of the Modula-3 implementation is equivalent to that of the C code [Sirer96].

would add no-longer-needed blocks to the free block list), I ran it for 262,144 iterations (the number of blocks on the disk). The results are shown in Table 6.

		C	Interpreted Java	Modula-3	Omniware
Alpha	raw	0.7s	—	1.3s	—
	normalized	1.0		1.75	
	per block	2.8µs		5.0µs	
HP-UX	raw	1.3s	32.2s	2.1s	—
	normalized	1.0	25	1.6	
	per block	5.0µs	123µs	8.0µs	
Linux	raw	1.3s	46.5s	1.7s	—
	normalized	1.0	36	1.3	
	per block	5.0µs	177µs	6.6µs	
Solaris	raw	1.9s	24.6s	2.9s	2.2s
	normalized	1.0	13	1.5	1.16
	per block	7.2µs	94µs	11.1µs	8.4µs

Table 6: Logical Disk

Time to handle bookkeeping for 262,144 writes to a Logical Disk. The time is normalized to compiled C code. The per-block overhead is how much time must be saved on each write in order for the graft to break even. The mean of 30 runs is reported. $\sigma < 2\%$.

I measured the absolute time required to maintain the mapping between logical and physical blocks for the entire run. To justify using this graft, it must save more time than it takes: the overhead incurred per write should be less than the time saved by batching writes into segments. I found that the compiled technologies (Omniware and Modula-3) add a sub-10µs overhead per write, which is on the order of 1% of a typical disk seek time.

The interpreted Java overhead is on the order of 100µs, roughly 10% of a typical seek, so we would need to save one seek for every ten blocks written. This is not an unreasonable assumption to make, and so interpreted Java would work for this task. (Because of performance of Tcl on the first two tests, I did not take Tcl measurements for this test.)

When looking at a user-level server for managing the mapping, we assume that there is one upcall on each block write, with a upcall estimate of 10µs. This would double the overhead per write, but still keep it substantially lower than that of interpreted Java. The performance of a user-level server for this task would be close to that of compiled code.

4.4 Updated Java Performance Results

The technologies that have changed the most since these tests were run are Java (which is now typically compiled to machine code at run time with a “just-in-time” (JIT) compiler), and Tcl (for which a byte code compiler exists, as of Tcl 8.0). Upcall times, being bounded by hardware improvements, have not changed much relative to CPU speedups, and the quality of compiled code produced for mature compiled languages (e.g., C and Modula-3) has not changed in any significant way.[8]

According to the Tcl 8.1 release notes [Scriptics], with the byte-code compiled Tcl “you should see speedups of 2-3x on many programs and you may see speedups as much as 10-20x in

some cases.” Based on the measurements reported above, this performance improvement is not sufficient to bring Tcl into the running for the operating system extensions described here.

Java is another matter. With a just-in-time compiler for Java, previous work suggests that we should assume roughly a 20x speedup [Hölze94], which would bring Java into the same ballpark as Modula-3. New tests were run under Windows 95 on a 100MHz Pentium, comparing the performance of code compiled with C (gcc 2.8.1), run in a Java interpreter (Sun JDK 1.1.6), and run with a JIT (also Sun JDK 1.1.6, although originally developed by Symantec). The results are found in Table 7.

		C	**Interpreted Java**	**Java JIT**
Page Eviction	raw	4.8μs (7.1%)	86μs	7.5μs (3.7%)
	normalized	1.0	17.9	1.6
MD5 Fingerprinting	raw	271ms (11%)	6371ms	—
	normalized	1.0	23.5	
Logical Disk	raw	1.5s	14s	2.2s (2.9%)
	normalized	1.0	9.3	1.5

Table 7: Updated C vs. Java Tests, Windows 95, Sun Java SDK
Performance comparison of C code, interpreted Java, and Java processed by a just-in-time compiler. Tests were run on a 100MHz Pentium running Windows 95. $\sigma < 2\%$, except where noted. Tests were run as described above.

The Sun 1.1.6 interpreter is substantially faster than the previously tested Alpha 3 interpreter, by roughly a factor of four in all three cases.[9] With the JIT, we see an additional 6x speedup in the case of the Logical Disk test, and a 12x speedup in the case of the Page Eviction test. The normalized results for these tests show Java JIT running about 50% slower than compiled C, which brings Java into the performance ballpark of Modula-3 and Omniware.

The MD5 fingerprinting test did not run with the Sun JDK 1.1.6 JIT. By querying Sun’s on-line bug database [SunJavaBugs] I learned that a bug in the 1.1.6 JIT prevents the JIT from compiling certain functions containing complex arithmetic expressions.[10] To obtain an estimate of the speedup possible for MD5, I reran the test using Microsoft Visual J++ 1.1 (Microsoft’s implementation of the Java language), and, although it is clearly not an apples-to-apples comparison, MD5 ran in just under 1.5 seconds, or roughly one-fifth the speed of the compiled C code.

Unlike the Page Eviction and Logical Disk tests, the MD5 (partial) result does not place Java JIT in the same performance league as Modula-3 and Omniware. However, if one considers that the Sun JIT generates code that is as efficient as Modula-3, it is safe to assume that, once fixed, it will be able to generate code for MD5 with similar performance characteristics. This would make the performance of Sun JIT Java good enough for the extensions discussed in this chapter.

---

8. As an indication, gcc has gone from version 2.7.0 to version 2.8.1 (four releases), and Modula-3 from 3.5.3 to 3.6 (one release)
9. This estimate comes from comparing the normalized times measured previously on Linux (on a 90MHz Pentium) to the new results (measured on a 100MHz Pentium, on Windows 95).
10. In the report for bug 414295, an example of the code that fails is given; to my amusement, I realized that it is a Java implementation of a section of the MD5 algorithm. Although the bug is marked as closed in the report of bug 4138234, when I applied the patch available, performance *decreased* relative to the interpreted version. Based on some investigative work, I believe that the JIT is attempting to compile the code, failing, and invoking the interpreted form instead, which explains the overhead.

4.5 Conclusions

This taxonomy of grafts and of graft architectures encompasses a significant share of the operating system extension space. The sample grafts (VM page eviction, MD5 fingerprinting, and Logical Disk) are equivalent in structure and performance characteristics to the policy, performance, and functionality grafts I envision.

The anecdotal evidence has suggested that structuring a system with fine-grained user-level extensions and upcalls is not feasible; our VM page eviction test supports this position for three of the four hardware platforms tested. On the other hand, the coarse-grained and computationally expensive MD5 test shows that there are some situations where this overhead does not matter, and the Logical Disk simulation shows that upcalls can be successfully hidden in the face of a large number of I/O operations.

Given that our goals include both safety and high performance, a compiled technology, such as Java (with a JIT), Modula-3 or SFI is the best choice. Because Modula-3 is unfamiliar to many developers, Java (with the syntax of C++ and the safety of Modula-3) may be a better choice for a safe language. By all accounts, acceptance of Java by C programmers has been overwhelming. This is especially surprising considering the general lack of acceptance of Modula 3, which offers many of the features of C favored by low-level hackers, such as address arithmetic and the LOOPHOLE construct, which is analogous to C's type casting. (Admittedly these features are only available in modules explicitly marked unsafe—but they are in the language.)

Neither of the interpreted technologies are up to the task. While interpreted Tcl and interpreted Java are well suited for interactive applications—where the relevant metric is human perceptual time—they are not suitable for building kernel extensions because system events occur at a finer timing granularity.

For the VINO project we chose a home grown implementation of SFI, which is described in the next chapter. This choice was made before Java JIT compilers were available, as well as before Leidke's work on very high performance IPC was published. Were we starting today, we would likely make a different decision.

Leidke's work on L4 points to the possibility of building an extensible kernel on the x86 using very low-cost hardware protection [Leidke97]. Although later work on Linux-over-L4 [Härtig97] did not have as impressive results, the implication of the initial work is that, when starting from scratch with a new kernel (which is what we would be doing), hardware protection can be very inexpensive.

On the other hand, Java is very *au courant*. Given the strong argument for building a large system with a single object and data model, and the performance of JIT-compiled Java, it makes sense to consider writing an extensible operating system completely in Java.

As is shown in the next chapter and in Chapter 7, the overhead of protecting extensions with my software fault isolation tool, MiSFIT, is sufficiently small to allow significant improvements in end-to-end application performance. The increased application performance possible through the use of a highly-efficient hardware protection scheme, or the (potential) simplification of using an off-the-shelf software protection scheme, might be nice to have but is not essential for the task at hand.

Chapter 5

Kernel Safety

> *Early [extensions] continually caused system crashes. They interfered not only with DOS but also with each other. Some were tenacious as bulldogs in taking control of the machine.*
>
> *– Terry Dettmann [Dettman89]*

5.1 Overview

This chapter describes the design and performance of MiSFIT, the Minimal i386 Software Fault Isolation Tool. MiSFIT transforms x86 assembler code, generated by a C or C++ compiler, into safe binary code.[1] Combined with a runtime support library, VINO uses MiSFIT to fault-isolate the kernel from end-user extensions.

As discussed in Chapter 2, *Software fault isolation* is a technique for transforming code written in an otherwise unsafe language into safe code. At transformation time, each read, write, and jump instruction is analyzed and, if necessary, transformed to ensure that it will not reach outside the memory region (*sandbox*) assigned to the code. This technique can be used to construct safe extensible systems that can be dynamically extended with code supplied by untrusted parties. Because software fault isolation ensures that misbehaved extensions cannot compromise the safety of the system, extensible systems constructed in this way do not suffer the instability of unprotected extensible systems such as MS-DOS.

Through its implementation of software fault isolation, MiSFIT can be used to fault-isolate dynamically linked extensions to World Wide Web browsers (e.g., Netscape Navigator), kernel extensions (which are supported by a variety of current systems), and client code linked to a database server (e.g., the Illustra database server).

5.2 SFI Is Not Enough

There are three reasons why the use of MiSFIT alone is not enough to protect from misbehaved extensions.

1. An introduction to the x86 instruction set architecture is found in the Appendix.

First, guarding against from errant writes and calls is not sufficient protection; the application or kernel must provide a safe interface to the extension, or a safe environment in which it can run. Protection against illegal stores is useless if the extension can call **bcopy()** with arbitrary arguments. Safe equivalents of many other commonly used routines, such as **read(), write(),** and **printf(),** may also be needed, depending on the interface provided to the extension.

Second, and more important, software fault isolation (or any other memory protection mechanism) is not a substitute for a more general resource management strategy. An extension should not be allowed to allocate memory, obtain a lock for a critical data structure, or even be given the freedom to run on the CPU, unless some mechanism is provided for such resources to be revoked. In Chapter 6, I discuss the technique of wrapping each extension invocation in a transaction; if the extension fails or holds resources for too long, our system aborts the transaction and nullifies any changes made by the extension.

The third way in which MiSFIT does not provide a complete solution is that an additional mechanism is needed to verify that a piece of code has been processed by MiSFIT. There are at least two methods for obtaining this verification. The first requires that extension writers distribute source code for their extensions. Users installing the extension would compile and run MiSFIT on the code themselves before installing it. This technique may be reasonable for operating system extensions, and is currently used with loadable kernel modules in NetBSD and Linux.

A second method for obtaining verification of MiSFIT processing is more end-user-friendly, but logistically more complex. Code processed by MiSFIT can be given a cryptographic digital signature, either by the tool itself or by a signing authority. This signature would then be checked at load time. In order to support this scheme, it would be necessary to find a trustworthy authority willing to MiSFIT and sign code, or somehow safely hide the apparatus for generating the signature within MiSFIT itself (as is currently done in VINO).

Although there are supplemental pieces needed to make MiSFIT it a complete environment for building extensible systems, the pieces are both technically tractable and application-specific. For the VINO project we have developed a protected runtime environment (Chapter 3), resource management infrastructure (Chapter 6), and code signature scheme (Section 5.3.10) for use with MiSFIT. Other applications of MiSFIT would necessarily have a different safe runtime environment and resource management infrastructure.

5.3 MiSFIT Design and Implementation

Software fault isolation techniques can be implemented in a compiler pass [Silver96], a filter between the compiler and assembler (as with MiSFIT), or a binary editing tool [Wahbe93]. MiS-FIT was implemented as an assembler-level filter for several reasons.

First, oding so greatly simplified the task at hand. An x86 binary editing tool needs to parse, disassemble, patch, and reassemble x86 binary code. A new pass for a C compiler would require involvement with the internals of the compiler, and tracking changes to the compiler over time.

Second, our implementation model conforms to the Unix tool-oriented approach for building systems. By not building MiSFIT into a compiler, MiSFIT gains a degree of compiler independence. Although MiSFIT has only been run with the Gnu C and C++ compilers (gcc and g++), it could easily be modified to accept the syntax generated by other x86 compilers, such as the Microsoft C compiler.

MiSFIT acts as a filter, scanning the output of the compiler and building an in-memory representation for the module. It then processes each instruction of the module in turn. If any intrinsically unsafe instruction (e.g., **halt**) appears, the module is rejected. The arguments for each store, call, and (optionally) load instruction are examined and, if necessary, transformed. Once the module has been processed, simple peephole optimization is performed to remove any redundancies introduced by the transformations.

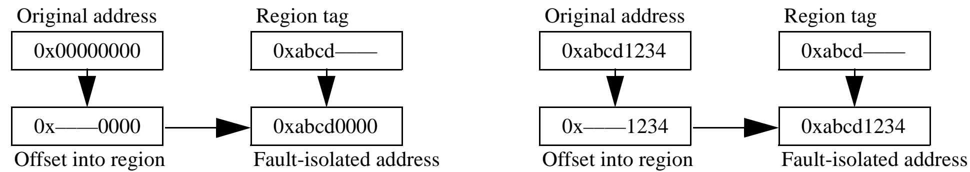

Figure 2. Example MiSFIT Transformations

In this example, the region tag is the top sixteen bits of the address and has the value 0xabcd. In the example on the left, the original address is invalid, so the fault-isolated address differs from the original address. In the example on the right, the original address is within the region, so the fault-isolated address is the same as the original address.

Protecting against illegal stores and jumps is necessary for safety, but protection from illegal reads is usually a question of security, not safety. If an extension could read outside its memory bounds, it would be able to read data to which it should not have access. This is undesirable, but not as bad as an extension being able to write or jump to an arbitrary location in memory, which compromises the safety of the host program. In one case, however, read protection is also a safety issue—when running in privileged mode (inside the kernel) on a machine with memory-mapped device registers that can be destructively read (i.e., once read, the value is lost). An extension that can read arbitrary locations in memory could read (and reset) device registers, thus wreaking havoc on the state of the system.

5.3.1 Indirect Loads and Stores

Loads and stores that use an indirect address (one computed at run-time) are potentially unsafe. MiSFIT inserts code to *sandbox* arguments of these instructions to force the indirect address to fall within a legal range.

Each user extension is assigned a contiguous region of memory into which it can write, and a region from which it can read. (These regions will normally overlap, but it is not necessary that they do so.) In order to preclude the code from modifying itself (and thus potentially circumventing MiSFIT's transformations), the writable region must not overlap the region assigned to the extension's code.

MiSFIT requires that each memory region be a power of two bytes in size. Because of this, the high bits of each address in a given memory region (the *region tag*) will be the same. To sandbox a memory reference, MiSFIT inserts code that sets the high bits of the reference so that it matches the region tag of the assigned memory region. Any load or store that would have accessed memory outside its region is thus forced to fall somewhere inside the extension's memory region.

Note that if the fault-isolated target address were already in the extension's memory region, the target address would not change. The fault isolated address differs from the original target address only if the original target address was outside the extension's memory region (and therefore illegal). Examples of this transformation are shown in Figure 2.

MiSFIT modifies loads and stores in the following way. First, it inserts code to load the target address into a register, if it is not already in one. The high bits of the register are then cleared, and the region tag of the associated memory region is loaded into the register. The register is then used in place of the operand from the original instruction.

Depending on whether the target address was already in a register, this technique adds either two or five instructions. If the original operand is an indirection through a single register (with no constant offset) only two instructions are needed, one to clear the high bits of the register and one to set the region tag. If the target address is not already in a register, MiSFIT inserts five instructions: MiSFIT obtains a scratch register (by pushing its current value on the stack), loads

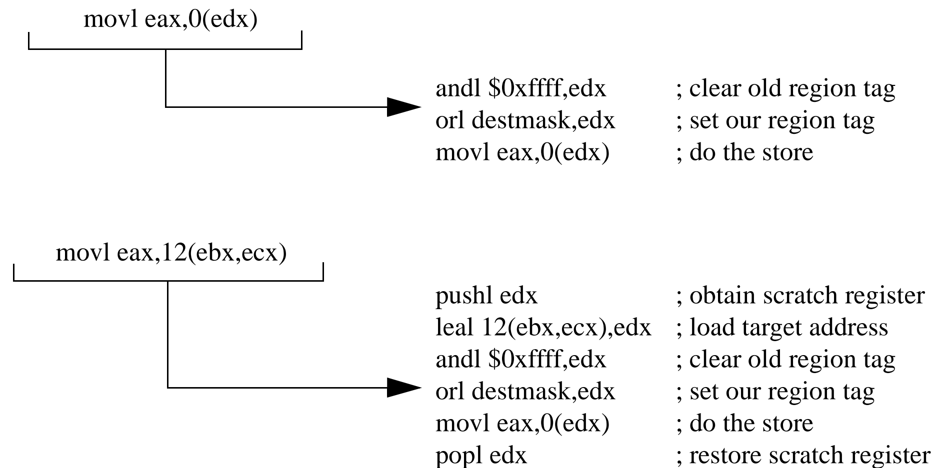

Figure 3. Sandboxing transformations for a store instruction.

In the first case the target is a simple indirection through a register; in the second case it is a complex indirection, so a scratch register is first made available and the target is loaded into the scratch register before sandboxing. In this example, the size of the assigned memory region is 64KB and thus the argument to the andl is 0xffff. Note that the added instructions take one or two cycles on the Pentium (assuming that the stack targets of the push and pop are in the first level cache).

---

the effective target address into the scratch register, masks in the region tag as described above, and restores the scratch register.

Examples of these transformations are shown in Figure 3. Note that in the second case it would be possible to save the scratch register push and pop if MiSFIT were able to determine that there was a register available for use as a scratch register. MiSFIT overhead is small enough that I have not been tempted to implement this optimization.

5.3.2 Indirect Jumps and Calls

When an indirect function call takes place, MiSFIT must verify that the target address is one that the extension is permitted to call. If the extension were allowed to call any address, it not only might invoke an unsafe function, it also might jump into the middle of an instruction or into data space, which would open all sorts of security and safety holes.

MiSFIT restricts the extension by replacing the indirect call with a call to a routine that searches a table of valid function targets. If the target address is in the table, the function call is made; if not, the call is not made. Runtime code (provided with MiSFIT) creates this table at program start-up, using the application's symbol table to find the start address of each function that an extension may call.

Although there may be an arbitrarily large number of valid target addresses, search time is limited by storing the valid addresses in a sparse, open-addressed, hash table, which has expected case linear search time [Cormen91]. By decreasing the density (increasing the size) of the table, it is possible to reduce the number of probes needed arbitrarily close to unity. With a table that has a 50% density (half the slots are empty) the expected number of probes per indirect call is 1.5. The overhead of each probe is roughly ten instruction cycles, so, on average, approximately fifteen instruction cycles are added to each indirect function call with this table density.

Indirect calls are common in C++ code, since virtual functions are implemented as indirect calls. When protecting C++ code with MiSFIT, the table of valid function targets can become quite large but the per-invocation cost remains low, because the number of probes into the table is

independent of the size of the table, depending only on its density, which is under MiSFIT's control.

Indirect jump instructions are generated by a compiler to implement jump tables, and are most often associated with C and C++ "switch" statements (and their analogs in other languages). Just as MiSFIT assumes the code it is processing was generated by a well-behaved compiler, it assumes that any indirect jumps were generated as part of a switch statement and are safe. If desired, the technique used for indirect calls could be applied for intra-function indirect jumps, where each function would have associated with it a table consisting of the code labels for that function. If MiSFIT were to be used with input from an untrusted source (e.g., user-written assembler, or a compiler that is ill-behaved) this transformation could easily be added. (This transformation falls into the same category as the stack checking instructions discussed in the next subsection.)

5.3.3 Matching Pushes With Pops

Code generated by a well-behaved compiler will leave the stack in the state in which it found it; in other words, every **push** instruction will be matched by a corresponding **pop** (or equivalent). Because MiSFIT assumes that the code it transforms was generated by a well-behaved compiler, it does not generate code to ensure that this is the case. It does not check ensure that an active function does not overflow or underflow its stack, nor does it ensure that the stack pointer (**esp**) has the same value on function exit that it had on function entry.

In the common case, static analysis can be used to verify that the stack does not overflow or underflow. As an example, the x86 passes function arguments on the processor's stack, which is addressed via the **esp** register. When calling a function, code will typically push the function arguments onto the stack (using the **push** instruction); when the called function returns, the stack pointer is reset by adding a constant to **esp**.

It is very rare for code to jump into, or out of, a sequence of instructions that performs a function call. In the common case, MiSFIT could compute the amount by which the stack pointer is decremented before the function call and compare it with the amount by which it is incremented when the called function returns. If the two match, no runtime check is needed. If MiSFIT were to behave conservatively, MiSFIT could refuse to process any module that does not meet this criteria.

Additionally, a more conservative implementation of MiSFIT could reject any module that uses the **push** and **pop** instructions in a potentially unsafe manner, only accepting code that it could statically verify was safe. Alternatively, MiSFIT could generate code that verified, at runtime, that the stack pointer did not overflow or underflow.

For the common case that MiSFIT was intended to address the current technique of trusting the compiler is workable; if MiSFIT were to be used in a more general (less trustworthy) environment, one of the techniques discussed in this section should be implemented.

5.3.4 Global Data and Virtual Function Tables

Because MiSFIT sandboxes global memory references, any data accessible to an extension must be placed in the memory region assigned to that extension. If there is global data that the extension should be able to access, that data should be placed in the extension's memory region as well. This applies not only to global program data, but to other shared state, such as the virtual function tables used by C++.

This restriction on global program data is a problem if multiple extensions are to be granted access to the same datum. A work-around is for the application to provide functions to access the data; each extension would be given permission to call these accessor functions, and would use them instead of directly reading and writing the data.

This technique has an impact on performance that is difficult to quantify, as the cost depends on the amount of data that is protected in this way, the frequency of access, and the type of interface the functions provide. I do not quantify this cost in the tests discussed in this chapter.

Virtual function tables are a different matter. If MiSFIT is configured to use read protection, virtual function tables need to be in a region of memory that is readable by the extension. The solution we have chosen for VINO is to store all virtual function tables in a contiguous region of memory and map it into the read-only region of each extension.[2]

5.3.5 Block Instructions

The x86 instruction set includes memory-to-memory move and comparison instructions, **movs** and **cmps**. These instructions can be used to construct block move and compare sequences, using the x86 **rep** instruction as a prefix. The **rep** prefix instructs the processor to repeat the memory-to-memory instruction for *count* times, where *count* is the value in the **ecx** register. The block move instruction sequence has a lower per-move overhead than a sequence or loop of individual memory-to-memory move instructions, and can be used to perform structure copies and in-line expansions of **strcmp()** and **bcopy**().

MiSFIT transforms the base addresses and repeat count of arguments to the block instruction, sandboxing the compound instruction as a whole. Although this adds a high fixed overhead to the block instruction (roughly 26 cycles), there is no per-element cost. The alternative, transforming the block instruction into a loop and sandboxing the instructions in the loop, has a high per-element overhead; the break-even point for the two techniques is at three or four **movs** instructions. Block instructions are typically used for copying or moving more than four data elements, so the fixed overhead imposed by MiSFIT's technique is preferable.

5.3.6 Saved Registers and Returned Addresses

Protecting the contents of the stack is problematic. The stack is used not only for local variables (which must be accessible to the user extension) but also for saved registers and the function return address (which should not be accessible to the user extension). If the user extension can write to arbitrary locations on the stack, the return address of the function can be set to an arbitrary value, circumventing the call protection offered by MiSFIT.

A second problem is that the process stack is not ordinarily in the same region of memory as the heap and global data, but MiSFIT's technique depends on all valid memory references falling within a single region of memory. In a multi-threaded environment (either a multi-threaded operating system kernel or multi-threaded end-user application) each thread of control is assigned its own stack. In an environment where the extension can be run as a separate thread of control, MiSFIT can co-locate the stack assigned to the thread (i.e., assigned to the extension) with the memory region assigned to the extension. Then all valid memory references made by the extension will fall within a single region.

When this is not possible, MiSFIT can provide the same type of protection by providing each extension with its own stack, located in its memory region. When the extension is invoked, the application switches to the stack associated with the extension. When the extension returns to the application, the process switches back to the original stack.

To solve the problem of an extension overwriting a return address on the stack, MiSFIT stores the return address of the function outside the extension's writable region. MiSFIT then replaces each **ret** instruction with a jump to a support routine that loads the saved return address and jumps to it. If the extension overwrites the contents of its stack, the system still returns to the correct

2. As of this writing, this has not yet been implemented.

location. Similarly, to ensure that register values are preserved across the invocation of the extension, MiSFIT saves the contents of all callee-saved registers on entry to an extension, and restores them when it returns.

5.3.7 Dynamic Linking

MiSFIT modifies the operands of indirect calls, and can statically verify that references to operands within the module are safe. References to functions outside the module need to be checked by the dynamic linker at extension load time. Under some circumstances, it may be the case that not all extensions will be given access to the same set of entrypoints. If this is so, the dynamic linker is responsible for determining which entrypoints a given extension should be allowed to access.

MiSFIT includes sample code that performs dynamic linking. This code can be modified to restrict extensions to a specific subset of the extensible application's functions.

5.3.8 Optimizations

In general, I have found that the overhead of MiSFIT protection is low enough that complex optimizations seem unnecessary. However, MiSFIT does perform two simple code optimizations.

The first is a peephole optimization that removes or performs strength reduction on pop/push instruction pairs, which are generated by the sandboxing code in order to obtain a scratch register. Popping the top of the stack into a register and immediately pushing it has no effect other than loading the contents of the top of the stack into the register. If the target register is known to be dead when the pop/push is done, the pop/push pair is removed. If the register is not known to be dead, the pop/push pair is replaced with an instruction that loads the target register from the top of the stack.

The second optimization removes unnecessary saves and restores of the x86 flags (condition codes) register. The instructions inserted by MiSFIT to sandbox an address set the condition codes, so it is possible for code inserted by MiSFIT to change the contents of the flags register. One way to solve this problem would be for MiSFIT to add instructions to save and restore the flags register whenever it inserts instructions to sandbox an address. Unfortunately, the instructions to save and restore the flags register are very expensive (four and nine cycles, respectively, on a Pentium).

We have found that condition codes, although frequently set on the x86 (e.g., by most arithmetic instructions), are rarely used, so it is often the case that code inserted by MiSFIT need not worry about preserving the condition codes. In order to determine when it is safe to overwrite the condition codes, MiSFIT performs live/dead analysis of the flags register. If, when an address is sandboxed, the condition codes are dead, MiSFIT does not generate instructions to preserve them. We have found this optimization to be very effective; in code processed by MiSFIT, the condition codes have rarely needed to be preserved.[3]

5.3.9 Stubs

When an extension is invoked, a small stub function, similar to an RPC stub, is called. This stub is responsible for configuring the extension's sandbox, saving callee-saved registers and initializing the global variables that hold the region tags for the read and write regions assigned to the extension. It copies any arguments passed to the extension onto the extension's stack, switches the stack pointer to the extension's stack, and jumps to the extension. When the extension completes, the stub copies any returned values to the caller's stack and then returns to the caller.

---

3. MiSFIT prints a informational message when it needs to preserve condition codes. The message has been printed only once.

The stub generator is driven by a specification, which marks parameters as input, output, or in/out. The stub generator uses standard techniques for creating the stubs. Unlike their RPC counterparts, arguments to MiSFIT stubs do not need to be marshalled, but simply copied from the caller's stack to the extension's stack (on entry) and the results copied back (on exit); hence, MiSFIT stubs can generally have less overhead than RPC stubs.

5.3.10 Code Signing

As described in Section 2.3.3.4, a system using any software protection technique requires verification that loaded code has been transformed appropriately. MiSFIT supports the use of a cryptographic signature, which can be verified at extension load time by the kernel. After an extension is assembled, the MD5 checksum [RFC1321] of the code segment, concatenated with a secret key, is computed, and stored with the extension.[4] When the extension is loaded, the kernel recomputes the checksum and verifies that the two match. If they match, the extension is loaded; if not, the extension is rejected.

This technique has its problems, the greatest of which is the need for a secret key shared between MiSFIT and the kernel. If the key becomes available to an untrusted party, the untrusted party could apply a signature to unsafe code and submit it for grafting. In the current implementation of VINO, I make the simplifying[5] assumptions that the key can be safely embedded in the code signing program and that the code signing program will only be invoked as part of a secure compilation process. For a production-quality system, VINO would need a more robust key management scheme.

5.4 MiSFIT Overhead

This section compares the performance of unprotected code (written in C or C++) with the corresponding MiSFIT-protected versions. Performance numbers for both write-call protection (where only store and call instructions are protected) and read-write-call protection (where load, store, and call instructions are protected) are included. As pointed out above, read protection is typically a requirement for security, not for safety (except in the case of kernels with memory-mapped device registers).

The following tests were run on a 100MHz Pentium running NetBSD 1.2.1. All results are the mean of 20 runs. The values shown in the figures are relative to unprotected code. In order to best estimate the overhead of protection, time spent in the operating system (and hence in unprotected code) has been factored out.

5.4.1 Operating System Extension Benchmarks

In Chapter 4, I examined the suitability of various extension technologies for constructing operating system extensions. Three tests were developed and used, with each test representing a class of possible OS extensions. The following is a short description of each test; for more detail, the reader is directed to the earlier chapter.

- *Page Eviction:* choose which page to evict from a linked list of page descriptors.
- *Logical Disk:* simulate the operation of a logical disk layer [deJonge93].
- *MD5:* compute the MD5 checksum of 1MB of data [RFC1321].

---

4. The secret key must be shared between the code signing application and the kernel. The details of key management techniques are outside the scope of this document.
5. i.e., unrealistic

Each test and its data fit into main memory. Normalized results are shown in Figure 4 and test run times in Table 8.

	Unprotected	Write-Call	Read-Write-Call
Page Eviction	352	446	1205
Logical Disk	1109	1454	1799
MD5	187	217	284

Table 8: Overhead of MiSFIT Protection on Operating System Extension Benchmarks
Time in milliseconds to run each extension test. Each figure is the mean of 20 runs. In all cases, $\sigma < 3\%$.

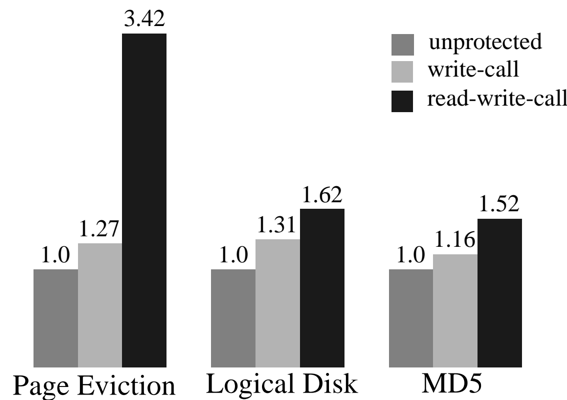

Figure 4. Overhead of MiSFIT Protection on Operating System Extension Benchmarks.
Results normalized to the unprotected case. Each figure is the mean of 20 runs. In all cases, $\sigma < 3\%$. Read protecting *Page Eviction* is expensive because of the large number of memory reads (pointer derefences) in the test.

The write-call overhead for these tests ranges from 16% to 31%, but the overhead for read-write-call protection can be much higher, for example, 242% for the Page Eviction test.

The write-call-protected Page Eviction test is 27% slower than the unprotected version; there are only a small number of calls and very few write instructions executed during the test. However, the core of the test repeatedly scans a linked list of page descriptors, and thus it executes a large number of read instructions. This is reflected in the performance of the read-write-call protected version of the test, where the overhead is over 240%.

The Logical Disk test has a write-call overhead of 31%, and read protection adds another 31%. This test is not as read-intensive as the Page Eviction test, so the added overhead of read protection is much lower. The MD5 test has similar performance characteristics, with a 16% write-call overhead and an additional 36% overhead for read protection.

In Chapter 4, I included a break-even point for each extension, with the assumption that along with the cost of running an extension, there is also some intended benefit (such as reduction in the number of page faults, or decreased disk access time). If the cost of using the extension is below the break-even point, the extension will improve overall system performance; if it above this point, it will degrade system performance. The three write-call-protected tests fall far below the break-even point, and the read-write-call-protected Logical Disk and MD5 tests are both below it. The read-write-call-protected Page Eviction test is at or just above the break-even point for this example extension.

In Chapter 7, I discuss the end-to-end performance of an application that uses a page eviction graft. The specifics of that test and graft differ from those here, and the application shows a performance gain from using the graft.

5.4.2 SPECInt Benchmarks

This experiment shows the performance overhead of MiSFIT protection on several SPEC benchmarks from the 1992 and 1995 integer suites. We ran tests using write-call and read-write-call protection. Normalized results are shown in Figure 5 and test run times in Table 9.

	Unprotected	Write-Call	Read-Write-Call
compress95	1125.9	1445.6	1597.7
eqntott92	18.6	21.7	33.4
espresso92	6.7	8.1	11.9
go95	296.2	357.4	507.7

Table 9: MiSFIT Protection on SPECInt benchmarks.
Time in seconds to run each SPEC test. Each figure is the mean of 20 runs. In all cases, $\sigma < 2.5\%$. These measurements give an estimate of the cost of protecting "typical" C code with MiSFIT.

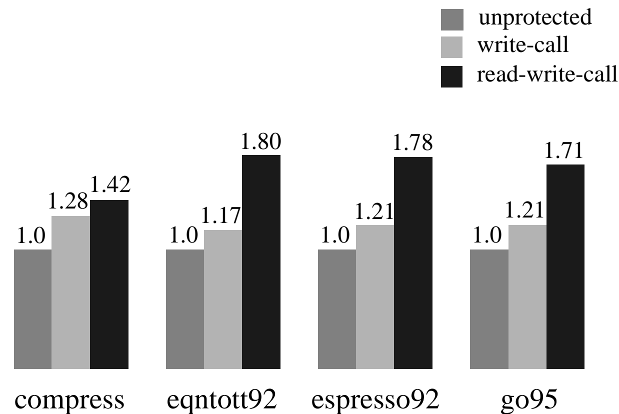

Figure 5. Overhead of MiSFIT Protection on SPECInt benchmarks.
Results are normalized to unprotected code. Each figure is the mean of 20 runs. In all cases, $\sigma < 2.5\%$. These measurements give an estimate of the cost of protecting "typical" C code with MiSFIT. There are some performance differences between the tests, but the results are similar.

It is highly unlikely that anyone would want to load a SPEC benchmark into a web browser, kernel, or database server. However, these results give a feeling for the overhead imposed by MiSFIT on "typical" code. To better estimate the overhead imposed by MiSFIT, the tables only include time spent at user level.

MiSFIT's write-call overhead for the SPECInt code is comparable to its write-call overhead for the operating system extension benchmarks, ranging from 17% to 28%. As seen in Figure 5, the overhead of read-write-call protection is higher, from 42% to 80%, which is comparable to the overhead seen for Modula-3 and JIT-Java in Chapter 4.

For memory-intensive applications, such as data copies, a higher overhead should be expected. In general, the overhead observed is a function of the ratio of protected instructions to unprotected instructions.

5.5 Conclusions

MiSFIT allows applications and kernels to be protected from end-user extensions written in otherwise unsafe languages. As part of an end-to-end solution to the problem of constructing an extensible system, MiSFIT can provide safety with a reasonably small amount of overhead.

Although MiSFIT ensures safety, it does not ensure kernel consistency. In the next chapter I discuss the kernel consistency mechanism used by VINO, Lightweight Transactions.

Chapter 6

Kernel Consistency Maintenance

A popular misunderstanding has it that transactions are bound to databases. Although it is true that historically, transactions have evolved from the database domain, it must be understood clearly that they establish a general execution paradigm that ideally covers all the subsystems invoked in their course.

– Andreas Reuter
[Gray93]

6.1 Introduction

As discussed in Section 2.3, an isolation mechanism ensures that a graft does not violate the interface boundary provided by the kernel—it only allows the graft to call the kernel functions exported to it. It does not, however, ensure that the graft will leave the kernel in a consistent state. Along with kernel safety, we need a mechanism to ensure kernel consistency.

When designing an interface for grafts, there are essentially two ways to guarantee that the system remains in a consistent state: one, ensure that (when the interface is used as intended) the kernel remains in a consistent state; or two, allow the graft to temporarily move the kernel into an inconsistent state, but ensure that the kernel eventually returns to a consistent state. The former style I term an *atomic interface* and the latter a *transactional interface*. In this chapter I define these interface styles in more detail and discuss why a designer would choose one over the other.

I then proceed to explain why we chose to use a transactional interface for VINO. The bulk of the chapter consists of a description and performance analysis of *VINO Lightweight Transactions* (VLTs), which provide high-performance ephemeral transactions.

6.2 Atomic Interfaces and Transactional Interfaces

Kernel code and grafts must be able to assume certain invariants about the kernel state. Without these assumptions, it is difficult, if not impossible, to make any forward progress. For example, in order to traverse a linked list, a piece of code normally assumes that the *next* field always points either to NULL or to a valid element. If it cannot make this assumption, it needs to check to make sure that the *next* field points to something valid—but how can it do so? One way would be to

include a tag field in each element; the code could then check the tag field of the memory pointed to by *next*. But this is no guarantee; the referenced address could hold any value, including the tag value. Another way would be to keep a second data structure that contains the addresses of all of the valid elements—but how can the code be sure that this second data structure remains consistent? In order to get work done, code must be able to assume that some set of invariants is maintained. How can we ensure that such assumptions are warranted?

First, let us consider the kernel as a state machine: if the kernel's memory is thought of as a sequence of bits, then each state of the kernel-as-state-machine is one of the possible setting of these bits. Alternatively, it may be more comfortable to think of the kernel as a set of kernel variables; each kernel state is then one possible settings of the kernel's variables. Either model works.[1]

The kernel's interface can then be modeled as a set of transition functions, which move the kernel from one state to another. As a simple example, consider a kernel function that changes the priority of the root thread from level 0 to level 1: this function moves the kernel from a state where the root thread has priority zero to the corresponding state where the thread's priority is 1.[2]

Now let us define a notion of *consistency*: assume that the set of kernel states is partitioned into two sets, one containing the states that are *consistent* (where the necessary invariants hold) and the other containing the states that are *inconsistent* (where they do not). For the following argument, it is not important how the states are partitioned, just that a partition exists.

An *atomic interface* is one that contains only functions that take the kernel from one consistent state to another; as long as the kernel starts in a consistent state, and the interface is respected (as we assume it will be, through the use of an isolation mechanism as discussed in Chapter 4), the kernel will never be in an inconsistent state.

Restricting oneself to atomic interfaces is the simplest way to ensure that the kernel is always in a consistent state: as there is no way for it to be moved into an inconsistent state, it will necessarily remain consistent. But what if we would like to allow grafts to *temporarily* violate an invariant, moving the kernel into an inconsistent state? As an example, assume that we have a kernel where the consistent states include all of those with every kernel thread on the *run* list or the *sleep* list; the inconsistent states include all those with one or more kernel threads on neither list. (Or, more intuitively, the kernel is consistent only if every thread is on one list or the other. I'll use the more intuitive, less verbose formulation for the remainder of this chapter.) We want to permit grafts to move threads from one list to the other, and provide two functions, *elt = pop(list)* and *push(list, elt)*. Between the time an element is popped from one list and the time it is pushed on the other, the kernel is in an inconsistent state.[3]

If the kernel offers an interface that allows intermediate inconsistent states, we must ensure two things. First, because grafts assume that the kernel is in a consistent state, they must be sheltered from these interim inconsistent states. Second, the kernel must eventually be moved to a consistent state; if the graft terminates while the kernel is in an inconsistent state, the kernel needs to be able to move itself to some consistent state.

To handle the first requirement, grafts should be *isolated* (in the ACID sense, as defined in Section 2.4) from inconsistent states caused by concurrently running grafts. This can be done by running the grafts serially; as the executions of the grafts do not overlap, they can never see an

1. The state machine is finite: since the kernel has a finite amount memory (or finite number of variables), there are only a finite number of possible values, and hence a finite number of states.

2. In actuality, as can be seen from even this simple example, each kernel function models a *set* of transition functions; in this case, the set of functions that map kernel states where the root thread's priority level is 0 to the states where the root thread's priority level is 1.

3. The kernel could obviously provide a function that atomically moved an element from one list to the other, with no intervening inconsistent state. This simple example is for expository purposes.

interim state. Although this approach is appealingly simple, the lack of concurrency may be too great a performance burden to bear. To enable concurrent execution of grafts, what is typically done is to place an exclusive lock on any state changed by a graft and hold the lock until the graft completes; while the lock is held, no other graft can see the potentially inconsistent state.[4] Other concurrency control mechanisms are possible, based on the amount of knowledge that the concurrency control mechanism has about the semantics of the system. Kung's work on the classification of concurrency control mechanisms shows that, although serial scheduling is the best possible when there is no information about which data are modified, for the case with which we are concerned, the style of concurrency control described here is optimal [Kung83].

Once graft isolation is taken care of, we are left with the problem of ensuring that the kernel is left in a consistent state by a graft when it terminates. And a graft will eventually terminate, either because it exits, it fails, or the kernel kills it (more on forced termination below). When the graft terminates, if the terminal kernel state is inconsistent, the graft is said to have *aborted.*[5] A graft that terminates successfully is said to have *committed.*

When a graft aborts, the kernel must move itself from the terminal inconsistent state to some consistent state. If we only cared that the kernel was in *some* consistent state, from the perspective of kernel-as-a-state-machine, all we would need to do would be to move the kernel from its current state to *any* state in the consistent set. However, moving the kernel to an arbitrary consistent state is not enough. To see this, consider a policy of moving inconsistent kernels to the start state, which is equivalent to rebooting.[6] This would mean that if a graft misbehaves and puts the kernel into an inconsistent state, the kernel's response would be to reboot—not at all the behavior we desire.

What we'd like to do is move the kernel to some "reasonable" consistent state, one that has some connection to its current state. We can move the state machine "forward" (applying additional transitions) or "backward" (undoing transitions) to a nearby consistent state. Moving "forward" may not be possible: there may be no transition from the current (inconsistent) state to a consistent state. If we move "backward," we could roll the kernel back to any of the consistent states through which it passed on its way to its current state.[7] Since the graft was misbehaved (leaving the kernel in an inconsistent state) it might be best to roll back through all of the transitions requested by the graft, to the point where it started. In this way, it is as if the graft never ran. This is termed *(failure) atomicity* (as discussed in Section 2.4), because the operations performed by the transaction are applied (or not) as an atomic, indivisible unit.

I define a *transactional interface* as one that provides the combination of isolation and atomicity.

It is possible to define interfaces that are neither atomic nor transactional. However, any interface that permits the kernel to move to an inconsistent state must provide isolation (to ensure that intermediate inconsistent states are hidden) and guarantee that the kernel is moved to a consistent state on graft abort (to ensure that terminal inconsistent states are not exported to other grafts).

The kernel could choose some other consistent state (rather than implement failure atomicity), but that would result in the kernel either spontaneously applying changes to itself (if it moved forward), undoing *more* than the aborted graft did (which is intuitively unappealing), or applying some consistency-preserving prefix of the sequence of operations performed by the graft. An argument can be made, in some cases, for the latter behavior: undo the invalid actions performed by the graft, but leave the effects of valid actions intact. In the case of VINO, the simplicity of failure atomicity made it the most appropriate choice.

---

4. This is a simplified description of the database community's *two-phase locking* [Bernstein87].
5. This is not the conventional way to define abort, but, in this model, it is equivalent to the standard definition.
6. A freshly-booted system is a great example of a consistent kernel state.
7. Moving back to the start state is a drastic example of rolling back to a consistent state.

6.3 VINO Lightweight Transactions

In order to ensure consistency in the face of misbehaved grafts, VINO protects itself with *VINO Lightweight Transactions (VLTs).* These are ephemeral (non-durable) transactions, designed to isolate concurrently operating extensions from each other and to provide failure atomicity.

Following one of the design goals of VINO, a graft loaded into the kernel will be permitted to run as long as it is not blocking the forward progress of any other computation (other than the process that installed the graft). For example, a process may load a graft that, once called, computes the first thirty million digits of π; as long as the graft is not holding resources that are needed by other computations, it will not be terminated. However, if a graft is blocking some other computation (say, by holding a lock that is currently in demand), its lifetime will be limited and, when its time limit is reached, the transaction system will abort the graft's execution.

In actuality, it is not the graft's execution that is aborted on a timeout, but the VLT that wraps the graft's invocation. When the kernel invokes a graft, it creates a VLT. Inside the VLT, any modifications that the graft makes to kernel state are logged (see below) and the modified state is locked. If the graft is aborted, the changes to the kernel's state are undone; if the graft terminates successfully, the locks are freed, releasing to the world the kernel changes made by the graft.

Along with the exclusive (write) locks for kernel state modified by the graft, the transaction mechanism provides weaker shared (read) locks for kernel state examined by the graft. Any number of transactions can hold read locks concurrently, but a write lock can only be held if no other locks are held. These locks are necessary to ensure that the graft is presented with a *consistent read set.* Without read locks, it would be possible for the execution of two grafts to be interleaved in such a way that the first graft could examine a piece of kernel state both before and after the state was changed by the second graft. Although both states would be consistent, in order to simplify graft programming, the transaction mechanism eliminates the possibility of this occurring.

6.3.1 Abort and Commit Logs

The VINO transaction mechanism must be able to undo changes made to the state of the kernel if a transaction aborts. Databases typically accomplish this using an undo log, which records changes made to the state of the database. If the transaction commits, the undo log is discarded. If the transaction aborts, the changes in the undo log are performed. Logging can be *physical,* in which log entries contain the data value overwritten by the transaction, or *logical*, in which a description of the operation is placed in the log. When deciding between physical and logical logging schemes, it is easiest to use the following rule of thumb: if it's cheaper to log instructions to recreate the initial state of the data than it is to log the data, use logical logging; if it's easier to just store the data, use physical logging. Changes to database state are often simple (e.g., "set the age field in record 137 to 42"), so physical logging is often used. Because of the generality of the changes made to VINO's state, the VINO transaction mechanism uses logical logging.

The VINO transaction mechanism uses two logs, an *abort* log and a *commit* log. The abort log contains log records corresponding to changes to kernel state that need to be undone if the transaction aborts; the commit log contains log records corresponding to changes that should be made to the kernel if the transaction commits. The commit log exists to hold operations that are difficult or impossible to undo, such as deleting an object, or writing a block to disk.[8]

Each log record consists of a function to call and the arguments to pass to the function, thus resembling the call frame that would be pushed on the stack were the function to be called. In point of fact, the abort and commit logs are constructed in the same manner as call stacks; when a record

8. The motivation for a commit log is that you can't put the missiles back into the silos once they've been launched.

is added to a log, the arguments and the address of the function to call are pushed in the same way that they would be were a function call being made. When it comes time to process a commit or abort log, the transaction system (effectively) switches to the log and evaluates the call frames. (To the best of my knowledge, this is a novel style of logging. I have been unable to find a reference to a similar style in the literature.)

When a transaction commits, the work done processing the commit log ends up being equivalent to the work that would have been done had the changes not been postponed: the steps taken to create the log record (pushing arguments onto a stack) would have been done anyway, and the cost of evaluating the log itself is minimal (a handful of instructions per log record).

6.3.2 Nested Transactions

The purpose of the VINO transaction mechanism is to make it safe for a graft to call into the kernel. What if the kernel then causes another graft to be invoked? When this happens, VINO creates a *nested transaction* [Moss85]. Unlike two unrelated transactions, a nested child transaction operates inside the context of its parent transaction. If a child transaction attempts to obtain a lock that is already held by its parent, the acquisition succeeds. In the case of two unrelated transactions, this is not the case: if transaction A holds a (read or write) lock on a datum, transaction B may not obtain a write lock on the same datum. If, however, a child of transaction A attempts to obtain a write lock on the datum, and transaction A holds the only lock on the datum, the acquisition will succeed; in this way, the child transaction "inherits" the locks of its parent.

When a child transaction aborts, its abort log is run, and its changes are undone. If, however, it commits, rather than run its abort log, the child's commit log and abort log are appended to the parent transaction's commit and abort logs. Additionally, when a child transaction commits, any locks held by the child transaction are not released, but are instead "anti-inherited" by the parent. Because the changes made to kernel state by the child transaction will not be released until the parent transaction commits, any locks held by the child transaction must be held until the parent commits.

6.3.3 Graft Safe and Graft Callable Functions

Any function that can be called in the context of a transaction needs to follow the rules of the transaction system: it must obtain locks on kernel data as needed, and it must log changes to kernel state in the transaction's commit and undo logs. Functions that can be called directly by grafts (which are untrusted code) must additionally verify their arguments as carefully as system calls do; these functions must ensure that arguments are declared to be of a given type (e.g., a pointer to a thread object) are, in fact, of that type, and that their values are in the appropriate range (e.g., the graft isn't trying to **bzero** the system's page tables).

Functions that follow the rules of the transaction system are termed *graft safe* (can be called in the context of a graft). Further, graft safe functions that perform the requisite argument checking are *graft callable* (can be called directly by a graft). The kernel runtime grafting support ensures that a graft only calls graft callable functions.[9] Although not currently implemented, it would be possible to ensure that graft safe functions only call other graft safe functions through the combination of static techniques (by building a call graph of the kernel) and dynamic techniques (by checking that any function called by a graft safe function was graft safe).

---

9. Static function calls are verified at graft load time, by the runtime linker; dynamic function calls are checked via the hash table, as described in the previous chapter.

6.4 Performance Overhead

I measured the overhead of the basic operations of VLTs. These tests were run on a 100 MHz Pentium (Intel Premiere/PCI II motherboard, Intel Neptune chipset) running VINO. All times are the mean of 100 runs, cold cache (expected time) numbers, and the standard deviation in all cases was less than 4%. All timing measurements were taken using the CPU's on-chip cycle counter, which has a resolution of 10 ns.

Transaction begin is the overhead of initializing a transaction when a graft is invoked. This includes allocating transaction structures and logs.

Transaction commit is the time needed to commit a transaction with no locks and no commit or abort records. Remember that the abort log is simply released on transaction commit, and work done processing the commit log is simply work that has been postponed until the end of the transaction (with a very small additional overhead to process the log). The work done includes verifying that there are no locks to release, and nothing on the commit log to run, and freeing the structures associated with the transaction.

Lock acquire is the time required to obtain an uncontested lock. This is larger than might be expected because of the cost incurred in creating a “lock object” when a lock on a datum is first acquired; until a transaction attempts to obtain a lock on a datum, the corresponding lock object does not exist.

Lock release is the cost incurred when a single lock is released at transaction commit time. There is a constant overhead incurred when releasing locks, roughly 2.6µs, plus a 2.3µs per-lock cost.

For comparison with the lock times, I include the time required to acquire and release a VINO mutex. The mutex times are a lower bound on many of these costs, as at least one, and in some cases multiple, mutexes must be acquired while performing these operations.

When examining the times in Table 10, keep in mind that these costs are very low when compared to the costs of traditional, durable transactions. Without special hardware support (e.g., NVRAM) a durable transaction cannot commit until a synchronous write to a disk has completed; such a write takes on the order of five to ten milliseconds, which is two to three orders of magnitude longer than the time needed to commit a VLT.

transaction	begin	7.3µs
	commit	8.4µs
lock	acquire	8.8µs
	release	4.8µs
mutex	acquire	1.4µs
	release	0.7µs

Table 10: VINO Transaction Overhead

Microbenchmarks measurements of the component costs of VLTs. End-to-end costs are measured in the next chapter. Each figure is the mean of 100 runs. $\sigma < 4\%$ in all cases.

6.5 Conclusions

VINO Lightweight Transactions provide the required support for isolation and failure atomicity that VINO needs for its concurrently running grafts. The overhead of the mechanism, at the microbenchmark level, is very low; as will be seen in the next chapter, the overhead of protection in end-to-end tests is so small that it is imperceptible.

Chapter 7

Application Results

7.1 Overview

In this chapter I discuss the results of three end-to-end performance studies. The goal is to show that the VINO grafting model is both powerful enough and cheap enough to allow for meaningful performance improvements.

7.1.1 Database Application Performance

The first study employs well-researched techniques for improving database application performance. The first part of the study measures the performance improvement that can be obtained by giving an application control over VM page eviction. The second part shows the gains that a database application can achieve when it controls its read-ahead policy. These experiments demonstrate that the overhead of the grafting infrastructure for safe extensibility (MiSFIT and VLTs) is low enough to allow improved application performance.

In these experiments, I take three measurements of application performance: (a) without the graft installed, (b) with the graft installed (using MiSFIT and VLTs), and (c) with the graft code written in unprotected C, hard-coded in the kernel. There are three possible outcomes:[1]

- *grafting is not worthwhile:* the grafted application (b) is slower than the application without the graft (a).
- *grafting is worthwhile (but expensive):* the grafted application (b) is faster than the ungrafted application (a), but much slower than the application with an unprotected graft (c).
- *grafting is cheap:* the grafted application (b) is faster than the ungrafted application (a), and not much slower than the application with an unprotected graft (c).

1. At most three outcomes, if we assume that (c) performs better than either (a) or (b).

7.1.2 Events

The second study demonstrates the use of file system event grafts, described in Section 3.5, and measures the end-to-end cost of transactions, described in Chapter 6. The study compares the costs of implementing an application at user level (where it must periodically poll to see if the state of the system has changed) with the use of event grafts (where the graft is notified when the event takes place).

The first experiment draws the line between situations in which it is more efficient to use event grafting and those in which it is more efficient to write an application in a conventional style. The experiment does this by measuring the component costs of each approach and computing the break-even point between the techniques.

The second experiment measures the overhead of the transaction mechanism on the event graft. In this experiment, the graft writes a bookkeeping record to the disk on each event. The transaction mechanism ensures that when the graft terminates, all resources allocated by the graft are returned. I compare the time to run the protected graft (with transactions), the unprotected graft (without transactions), and the cost of running the graft's code at user level.

7.1.3 File System Layout

In the third study, I show how giving an application control over the on-disk layout of its files can improve application read performance. The extension used in this study determines where a disk block should be placed based on information it obtains from the operating system. Once information has been provided to the extension, the operating system must ensure that the property of isolation holds—that the data does not change once it has been given to the extension, and that the extension's changes to the state of the system are kept hidden from view until the extension's transaction commits. Isolation is ensured through the use of the VINO Lightweight Transaction mechanism.

7.2 Test Platform

All tests run in this section were performed on a 100 MHz Pentium computer (Intel Premiere/PCI II motherboard, Intel Neptune chipset) with 16 MB of memory. The disk subsystem consisted of an Adaptec 1515 SCSI controller and a 2.2 GB 7200 RPM Seagate ST32272N disk, with an average access time of 9.4 ms. To improve repeatability, the system was run single user with no non-system processes running, and was freshly rebooted before each set of tests. All timing measurements were taken using the CPU's on-chip cycle counter, which has a resolution of 10 ns.

7.3 Database Performance Improvements

The first two tests (Section 7.3.3 and Section 7.3.4) use the Wisconsin Benchmark database to investigate whether giving a database application control over page eviction and read ahead can improve its performance. I include a short introduction to the Wisconsin Benchmark in Section 7.3.1. In Section 7.3.2, I give a baseline measurement for the database experiments. The first experiment, which explores application control over page eviction, is in Section 7.3.3, and the second experiment, on application control over read ahead, is in Section 7.3.4.

7.3.1 The Wisconsin Benchmark

The Wisconsin Benchmark was originally designed to compare the DIRECT database machine (developed at the University of Wisconsin) with the Ingres database (developed at the University of California, Berkeley) [DeWitt91]. At the time, the only database benchmark programs available were tied to specific applications; the developers of the Wisconsin Benchmark had as their goal to develop a benchmark suite that was application-neutral and concentrated on the performance of the underlying database.

The benchmark database consists of a set of tuples (records), where each tuple includes thirteen (four byte) integer fields and three 52-byte string fields, for a total of 208 bytes. When originally designed, the size of the benchmark database was set to 10,000 tuples. This was later found to be inadequate, as machines rapidly became available that would hold a 10,000 tuple database in main memory. The authors then redefined the database size and queries in terms of the size of the main memory cache, which can be at most 20% of the size of the database. In the following tests I use a 100,000 record (21 MB) database and limit the main memory database cache to 4 MB.

The benchmark suite is composed of thirty-two different queries, from selection[2] using a clustered index, to joins using a nonclustered index, to searching a subset of the database using no index.[3] In clustered-index and unindexed queries (where the database would be read sequentially), standard heuristics (LRU page eviction and linear read ahead) provide the behavior that best fits the needs of the database. Because of this, the tests performed here concentrate on queries that use a non-clustered index (where the database system has information about its future behavior that can be used to improve on the operating system's heuristics).

7.3.2 Baseline

In the two tests I run a standard Wisconsin Benchmark query that performs a ten percent selection via a nonclustered index (query 6 of the Wisconsin Benchmark test suite). "Ten percent selection" means that ten percent of the database (10,000) records are read by the benchmark program. The use of a nonclustered index means that the records are (pseudo-) randomly distributed within the database.

One question that needs to be asked is whether any of the improvements seen here are a result of VINO having poor default algorithms or intrinsic performance problems that would simplify the task of finding performance improvements. In Table 15, I show the runtimes of the ungrafted test on both NetBSD 1.2.1 and VINO. The test runs slightly slower on VINO (3.7%), and this difference should be kept in mind when comparing grafted and ungrafted application performance.

The performance difference is not surprising. NetBSD is a mature, single-threaded kernel, and VINO is a very young, multi-threaded kernel. We have not yet completed significant profiling or optimization of the kernel, although the process is underway.

---

2. A *selection* is a retrieval of a subset of records, usually those matching some predicate.

3. A *clustered index* is one on which the database is physically arranged, and so a selection of a range of records using a clustered index results in reading a contiguous subrange of the database. An *unclustered index* is one on which the database is not sorted, so a selection via an unclustered index requires reading a randomly distributed set of records from the database. A selection done using no index entails reading the entire database (this would normally be done sequentially).

OS	Time (seconds)	% VINO Time
VINO	104.2	—
NetBSD 1.2.1	100.3	96.3%

Table 11: Comparison of VINO and NetBSD

Test run with VINO (using default algorithms; taken from Table 12) and NetBSD 1.2.1. Each figure is the mean of five runs. In all cases, $\sigma < 1\%$.

7.3.3 Look Ahead Page Eviction

This test harks back to the VM Page Eviction test of Section 4.3.4, demonstrating the value of application control over virtual memory page eviction.

Because the cache can hold 20% of the database (4MB) and records are read randomly there is a 20% chance that, once the cache is warm, a previously read record will be in memory. Several policies will give us this 20% hit ratio: for example, we could just keep the first 20% of the database in memory, we could randomly replace pages as necessary, or we could use the default operating system policy (LRU).

However, this benchmark has an index that tells it which records (and hence which pages) it will be reading next. Rather than be satisfied with a 20% hit ratio, the benchmark can foretell the future and evict pages that will not be needed anytime soon. This *look ahead* policy should reduce the page fault cost of the benchmark, with an accompanying gain in performance.

The style of file access seen here (and hence the prospective usefulness of this extension) is not restricted to non-clustered, indexed database reads; any time an application's future use pattern is known to the application and does not match LRU, this type of extension will be applicable.

In this experiment, I run the benchmark query against the database while varying the algorithm used (LRU vs. look ahead), the number of pages looked ahead, and the level of protection (MiSFIT and VLTs[4] vs. no protection). The measurements are shown in the top half of Table 12. The overhead of calling the graft when it does no work is low (look ahead of zero, 0.8% overhead), and the possible benefit is large (look ahead of 4096, 22.3% improvement).

At first, as the amount of look ahead increases, the number of page faults decrease, as does end-to-end application runtime. Since the application does almost no work other than I/O, the percentage decrease in application runtime should equal the percentage decrease in page faults. This is nearly the case—the application runtime improvement is close to the reduction in page faults, and over 90% of the reduction in all cases. For example, with 2048 records of look ahead, we see a reduction from 7724 page faults to 6301 (81.6% of the original) and runtime decreases from 104.2 seconds to 88.7 seconds (85.1%), or 96% of the gain possible from the reduction in page faults.

At 8192 records of look ahead, the application runs over a cliff, and page faults and end-to-end application runtime increase greatly. As can be seen in Table 13, graft time goes from less than five seconds to thirteen *minutes*, and the number of graft failures (invocations where the graft can not find a suitable candidate) goes from zero to 1474. This is because the graft is trying to keep more blocks in the cache than will fit. In a simulation study, I measured how large a cache would be needed to ensure that the look-ahead graft would always succeed. The results are found in Table 14. A graft looking ahead 4096 records or fewer will succeed with our 4 MB (1024 block) cache, but an 8192-record look ahead requires a 9.5 MB (2441 block) cache. This result emphasizes the

---

4. If a MiSFIT-protected graft does not call into the kernel, it can not change kernel state, and hence does not need a transaction. In such cases, VINO will normally not transaction-protect such a graft. For these tests, this optimization was disabled.

	Look Ahead (records)	Page Faults (pages)	% LRU Faults	Time (seconds)	% LRU Time
LRU	—	7724	—	104.2	—
look ahead MiSFIT and VLT protection	0	7817	101.2%	105.0	100.8%
	128	7738	100.2%	104.3	100.0%
	256	7580	98.1%	102.4	98.3%
	512	7322	94.8%	99.6	95.6%
	1024	6904	89.4%	94.3	90.5%
	2048	6301	81.6%	88.7	85.1%
	4096	5513	71.4%	81.0	77.7%
	8192	6229	80.6%	899.3	863.1%
look ahead no protection	0	7758	100.4%	105.0	100.1%
	128	7677	99.4%	102.5	98.4%
	256	7522	97.4%	101.7	97.6%
	512	7269	94.1%	99.2	95.2%
	1024	6852	88.7%	94.0	90.2%
	2048	6268	81.1%	88.3	84.7%
	4096	5486	71.0%	80.3	77.1%
	8192	6124	79.3%	849.0 (3.4%)	814.8%

Table 12: LRU and look ahead page replacement page faults.
A comparison of the number of system page faults measured when running the benchmark with LRU replacement and look ahead replacement, along with the LRU-relative improvement. Measurements were taken both with and without protection (MiSFIT and VLTs). The benchmark is I/O bound, and application performance tracks the reduction in page faults, except in the case of 8192 records of look ahead. Each figure is the mean of five runs. $\sigma < 1\%$ except where noted.

importance of choosing the appropriate algorithm: by choosing the wrong algorithm (in this case, looking ahead too far), performance can drop precipitously.

However, given that look ahead can cut application run time from 104.2 seconds to 81 seconds (77.7%), it is clear that *grafting is worthwhile*: using grafting, the application gets a significant performance boost from a small effort on the part of the application writer. (The graft is about thirty lines of code; the text of the graft is found in Section 7.7.1.) The question that remains is whether *grafting is expensive* or *grafting is cheap*. Running the tests again, without protection, we see that the "% LRU Time" (benefit) difference between the protected and unprotected cases is minimal, less than 1% in most cases (the bottom half of Table 12)—so for this application, the cost of protection is very low, and *grafting is cheap*.

7.3.4 Random Access Read Ahead

Read ahead is a technique used by the operating system to decrease the amount of time an application spends waiting for data to be fetched from the disk. Most file reads are sequential [Baker91]; optimizing for the common case, operating systems attempt to detect sequential and take advantage of them through "reading ahead" blocks that have not yet been requested.

This technique is effective for several reasons:

- The disk driver can sort disk requests to minimize the total amount of seeking done. With a larger list of disk requests, the driver can do a better job of ordering them to minimize disk head motion.
- On modern disks, there is often a *track buffer*, which holds the contents of the most recently accessed track of the disk. If multiple blocks are read from the same track, all but the first may, under certain circumstances, be satisfied by the track buffer without performing any disk I/O.

	Look Ahead (records)	Failures	Graft Time (seconds)
look ahead **MiSFIT and VLT protection**	0	0	0.01 (15%)
	128	0	0.15 (5.9%)
	256	0	0.27 (7.1%)
	512	0	0.43 (2.9%)
	1024	0	0.76 (7.0%)
	2048	0	1.47 (9.1%)
	4096	0	4.27 (6.2%)
	8192	1474 (1.9%)	775.32 (1.5%)
look ahead **no protection**	0	0	0.01 (11%)
	128	0	0.15 (2.4%)
	256	0	0.28 (5.1%)
	512	0	0.43 (9.9%)
	1024	0	0.72 (8.0%)
	2048	0	1.39 (5.9%)
	4096	0	3.86 (4.8%)
	8192	1433 (1.3%)	727.96 (3.7%)

Table 13: Total Time Spent In Graft

For each test, the number of graft invocations that were unable to find a suitable candidate, and the total time spent in the graft. Each figure is the mean of five runs. $\sigma < 1\%$ except where noted. Although σ is relatively large, the mean values in the protected and unprotected cases are very similar.

Look Ahead (records)	Cache Required	
	Blocks (4KB)	Bytes (M)
128	9	0.1
256	14	0.1
512	38	0.2
1024	114	0.5
2048	356	1.4
4096	1009	3.9
8192	2441	9.5

Table 14: Cache Size Required for Success of Look Ahead Algorithm

Results of a simulation study showing how large an in-memory cache is needed for a given look ahead algorithm to always succeed for this application. In this context, success means always being able to find an unneeded block to evict. The in-memory cache used by the application holds 4 MB of data.

- If the application performs substantial computation as well as I/O, reading ahead may allow the application to overlap the two, which would increase the application's throughput. For example, an MPEG player may spend as much time computing as it does waiting for I/O; if the operating system can run ahead of the MPEG player, loading data into memory before it is needed, the player will not block on I/O.

There is a fly in the ointment, however; read ahead only works if the operating system knows which blocks the application will use next. One commonly used heuristic is to look at the last few (or even one or two) disk reads; if they were logically sequential, the operating system assumes that the application is reading the file sequentially and begins to read ahead. If the last disk reads

were not logically sequential, the operating system decides that the application is randomly accessing the blocks of the file and does no read ahead at all.

The test run here shows that, even with random reads, there can be a performance advantage to reading ahead. Because the application is using an index, it knows which records it will use in the near future and passes this information on to VINO. When configured for *n*-record read ahead, the application supplements every *n*th request with the blocks of the next *n-1* records it will be requesting. VINO then schedules these blocks for reading, with the hope that they will arrive in memory before they are needed by the application.

In this test I compare the time required to read 10,000 records (10% of the database) using the default policy (no read ahead) with the time required for the same read using a graft that indicates which (and how many) blocks to read ahead. I repeat the test without MiSFIT and VLT protection to show the overhead imposed by protection. The performance results are summarized in Table 15.

	Read Ahead (blocks)	Elapsed Time (seconds)	Default-Relative Performance
No read ahead	—	104.2	—
read ahead MiSFIT and VLT protection	**0**	105.5	101.2%
	20	83.2	79.8%
	40	79.0	75.9%
	60	76.9	73.8%
	80	75.7	72.7%
	100	74.1	71.1%
	120	74.3	71.3%
read ahead no protection	**0**	104.6	100.3%
	20	82.5	79.2%
	40	78.2	75.1%
	60	76.1	73.1%
	80	74.7	71.7%
	100	73.5	70.6%
	120	73.3	70.3%

Table 15: Read Ahead Benefit

Measurements of the time required to read 10,000 randomly-distributed records with varying number of read-ahead blocks. Measurements were taken both with and without protection (MiSFIT and VLTs). Each figure is the mean of five runs. In all cases, $\sigma < 1\%$.

We see that in both the protected and unprotected cases, the time required to run the application decreases as the number of blocks of read ahead increases, which is the desired result. The overhead of having the graft in place but doing no work (read ahead of zero) is low (1.2%), and the possible benefit (up to 30% improvement) is great.

We also see, when comparing the protected and unprotected results, that the overhead of protection is 1% or less in all cases. With a significant performance increase and an imperceptible protection overhead, we find in this test that *grafting is cheap.* (The body of the graft is three lines of code; the text of the graft is found in Section 7.7.2.)

The application's performance improves with the number of blocks of read ahead, but the rate of improvement slows as the amount of read ahead increases. With 20 blocks of read ahead, the runtime drops from 104.2 seconds to 83.2 seconds (about a 20% improvement), but increasing

read ahead by another 100 blocks only drops the runtime to 74.3 seconds (about a 30% improvement over the base case).

Although performance appears to decrease slightly between 100 and 120 blocks of read ahead in the protected case, the difference is less than 0.2%, which is in the statistical noise for these measurements.

The slowdown of the performance improvement as the number of blocks of read ahead increases can be ascribed to the (relative) decrease in the number of requests that can be satisfied by the disk without moving the disk head to a new cylinder. If we consider the read-ahead of *n* records of the database as, instead, a random selection (with replacement) of *m* cylinders of the *N*-cylinder database, we can compute *r,* the expected number of *different* cylinders selected [Feller50].[5]

$$m = N\left\{\frac{1}{N} + \frac{1}{N-1} + \ldots + \frac{1}{N-r+1}\right\} \quad \textbf{(EQ 1)}$$

The cylinders of the test disk are approximately 360 KB in size, so the database (21MB) fills at least 60 cylinders.[6] Given *N* (60 cylinders) and *m* (the number of records of read ahead) we can iteratively solve for an approximation of *r* (the number of unique samples found). The results are found in Table 16.

If we assume that the disk driver sorts requests so that blocks from the same cylinder are read in sequence, the table tells us what percentage of the requests will be satisfied from the disk without its needing to move the disk head. If, for purposes of argument, we assume a disk request from the current cylinder is satisfied in the time it takes the disk to perform half a rotation (4.2 ms in this case), and the time to seek between cylinders is, on average, 5.2 ms (starting with the average access time of 9.4 ms and subtracting the 4.2 ms half-rotation time), we can estimate the cost to read a group of records as well as the a per-record cost. Given these estimates, we can generate an estimated relative performance, given the number of records of read ahead. These results are found in Table 16.

Read Ahead (n records)	Cylinders Represented	Access Time (n records)	Per-Record Access Time	Relative Performance
default	—	—	9.4 ms	100.0%
20	17	172 ms	8.6 ms	91.7%
40	29	319 ms	8.0 ms	84.8%
60	38	450 ms	7.5 ms	79.7%
80	45	570 ms	7.1 ms	75.8%
100	49	675 ms	6.8 ms	71.8%
120	52	774 ms	6.5 ms	68.7%

Table 16: Analytic Model of Read Ahead Performance Improvement

When randomly selecting *n* records of the database to read ahead, what is the expected number of different cylinders selected? Given this number, and parameters of the disk, what is the per-record access time and what is the default-relative performance?

The estimated speedup derived from this analysis (a large initial improvement, with diminishing returns) is in line with the speedup measured in the experiments. The analytic results

5. Although we are selecting *n* records without replacement, it is equivalent in this case to selecting *m* cylinders with replacement: over 1700 records fit on each cylinder, and we are reading ahead at most 120 records.

6. If the database is not allocated contiguously on disk, it will be in a larger number of cylinders, which will change the results in the table. This analysis is merely a back-of-the-envelope estimate.

are not a perfect match, but given the simplicity of the model, the correspondence between the two supports the argument that this analytic model reflects reality.

In summary, for this experiment as well, the cost of protection is very low relative to the savings seen, and large benefits are possible. For this test, *grafting is cheap.*

7.4 File Unlink Events

As discussed in Section 3.5, VINO supports graftpoints that are called in response to system events. A filter is associated with the graft, and the graft is only invoked when an event is accepted by the filter. In this section's test, the application keeps track of a list of files, performing some action whenever one of the files is unlinked.[7] This graft could be used if a system administrator wanted to be notified whenever important programs or files were removed or replaced (e.g., if the system were attacked and the login program replaced), or if a user wished to automatically notify a centralized document-indexing service whenever a local document was replaced (as would be the case with a web search engine).

If the application were unable to use an event graft, it would need to periodically poll the status of each file of interest (checking to see if it had been unlinked since the last poll); using the event graft mechanism, the graft is invoked automatically whenever a file of interest is unlinked. The graft can either do the work of the application in the kernel, or pass a signal to the application and have the work done at user level.

Unlike the database application in Section 7.3, there is no *grafting is expensive* vs. *grafting is cheap* comparison to be made here. The break-even analysis shows us, given a set of parameters, how to decide between the two techniques.

In this test the graft is loaded at the file unlink graftpoint, and its filter consists of the list of identifiers (file system and file numbers) of the files in which it is interested. The first experiment, discussed in Section 7.4.1, compares the overhead of filtering (at kernel level) with the overhead of checking for changes at user level, ignoring the cost of any grafts run. In Section 7.4.2 I repeat the experiment, this time installing a graft that writes a bookkeeping record to a file. For this graft, transaction protection overhead is significant, and I show how it affects the break-even point for the application. In Section 7.4.3, I modify the graft so that it writes the bookkeeping record synchronously, and show how this modification changes the break-even point.

7.4.1 Events vs. Polling

The costs of the event and polling approaches can not be directly compared, as they incur different expenses, and provide different functionality:

- The event-based approach offers immediate notification, but incurs a cost each time any file is unlinked. Because the event handler is called on each event, all such events will be processed by the extension.
- The polling approach does not discover that a file has been unlinked until the next time it polls, but only incurs a cost each time it polls. In addition, it is possible for the polling approach to miss an event. As the frequency of polling increases, the frequency of missed events decreases, but we can only estimate the frequency of lost events; we can not eliminate them.

A user-level application can control how often it polls, trading overhead (polling cost) for latency (how quickly it learns of a change). The cost of the event-based approach is a function of

---

7. In the VINO file system (as in the Unix Fast Filesystem), disk files are reference counted, and each name associated with the file is a reference. When the reference count of a file drops to zero, the file is removed. *Unlinking* a file is removing a name of a file (which reduces its reference count). In the common case (the case presented here) a file will have only one name, so unlinking the file is tantamount to deleting it.

how often events are generated (in this case, how often files are unlinked) and the cost of checking each event against the filter. The cost functions of the two approaches are:

$$Cost_{graft} = Cost_{filter} \bullet Freq_{unlink} \quad \text{(EQ 2)}$$

$$Cost_{userlevel} = Cost_{poll} \bullet Freq_{poll} \quad \text{(EQ 3)}$$

$Cost_{poll}$ and $Cost_{filter}$ are fixed, and can be measured. The application controls $Freq_{poll}$, but $Freq_{unlink}$ depends on system activity, which is outside the control of the application.

There is a fixed cost associated with having event notification code in the kernel, even if no events are ever dispatched. I measured the time required to run the unlink event code with no handlers registered, as well as the event processing cost with registered handlers of varying filter size ($Cost_{filter}$). In this experiment, the event is not accepted by the filter, and so the results include the cost of checking each of the elements of the filter, but do not include the cost of invoking a graft.[8] The results are shown in Table 17.

Filter Size (files)	$Cost_{filter}$ (Time µs)	Time Per File (Time µs)
no filter	9.6	—
1	14.4	14.4
2	14.7	7.4
3	15.1	5.0
10	15.9	1.6
50	22.4	0.4

Table 17: Filtering Cost

Per-unlink base cost of filtering code (9.6µs) and cost with filters of varying size (number of files checked). The increase between no filter and one filter is the cost of invoking the code that checks the filter. Mean of 1000 runs, $\sigma < 3\%$.

We see that the fixed overhead of the event mechanism is low (roughly 10µs), as are the costs of checking a single file (roughly 5µs) and of checking additional files (0.3µs or less per file).

We now need to measure the cost associated with checking the status of one or more files under VINO ($Cost_{poll}$). Because this path through the kernel has not been optimized, I also measure the time required to perform the analogous task running NetBSD 1.2.1 on the same hardware. These times are shown in Table 18. Because of the difference between the VINO and NetBSD times, I use the more conservative NetBSD figures for the following discussion.[9]

Given this data, we can now compute the break-even point where $Cost_{graft}$ equals $Cost_{user\ level}$, which is given in terms of the relative frequency of unlink operations and polls. We start by setting

$$Cost_{poll} \bullet Freq_{poll} = Cost_{filter} \bullet Freq_{unlink} \quad \text{(EQ 4)}$$

and find that the relationship we seek is:

$$Freq_{poll} = \frac{Cost_{filter}}{Cost_{poll}} \bullet Freq_{unlink} \quad \text{(EQ 5)}$$

The break-even points for the cases in question are found in Table 19.

---

8. The filter is stored as an unsorted array, and each test requires a linear search of the array. A more efficient data structure could be used, such as a sorted array or hash table, although this implementation is still fast relative to polling.
9. These figures are conservative in that they bias against grafting in favor of the user-level technique.

Number of Files	Cost$_{poll}$ (time µs)			
	VINO		NetBSD 1.2.1	
	Total	Per File	Total	Per File
1	35.2	35.2	15.3	15.3
10	302.8	30.3	93.1	9.3
50	1580.2	31.6	481.8	9.6

Table 18: Time Required To Check File Status

Time required on VINO and NetBSD 1.2.1 to check the status of a number of files, using *vino_getattr()* and *fstat()*, respectively. Each test program first opens the files and then measures the amount of time required to get the status of the files. Each figure is the 1% trimmed mean of 1000 runs, $\sigma < 4\%$ in all cases. The 1% trimmed mean, which discards the fastest 1% and slowest 1% of the measurements, is used to drop roughly five outliers.

Number of Files	Poll:Unlink Break-Even Ratio
1	10:11 (0.92)
10	1:6 (0.17)
50	1:20 (0.05)

Table 19: Break Even Ratio

Given a number of files, the relative frequency of poll operations to unlink operations at which the two approaches break even.

This gives us the relative frequency of polls to unlink events at which the two techniques have equal cost. For example, if the application is keeping track of 50 files, and it polls 1/20 as often as file unlink events take place, the overhead of the user-level technique will be the same as the overhead of the event graft technique. If the application polls less frequently, the user-level technique will be cheaper; if it polls more frequently, the event graft technique will be cheaper.

We see that when checking only one file the relative costs of polling and using event grafts are close, but as the number of files increase, the relative cost of using event grafts drops. This is because the fixed cost of running a filter is high (10µs) and the incremental cost to check a file is small (less than 0.3µs), where the cost of polling a file is relatively high and constant (about 9µs on NetBSD).

These results beg the question of the frequency of unlink events on a typical system. To obtain an estimate for this figure, I measured the number of file unlink operations that take place when building the VINO kernel from scratch (`make clean; make depend; make`), a process that takes roughly twenty-three minutes (1382 seconds) on a 133MHz Pentium running BSD/OS 3.0. A total of 2099 unlink system calls were made, or 1.52 unlinks per second. If we assume our test application is running on a system under such a heavy load, we can compute the break-even points for this application. These are found in Table 20.

Number of Files Checked	Seconds Between Polls
1	1.6
10	8.9
50	32.7

Table 20: Example Polling Frequency

If we assume that unlink operations occur as frequently on the application's system as they do when the VINO kernel is being built (1.52 unlinks per second), what is the break-even polling frequency?

In the example of Table 20, I assume that unlinks are frequent (1.52 per second). Under lighter load, where unlinks might occur one or two orders of magnitude less frequently, the number of seconds between polls would also need to increase by the same factor in order to maintain the break-even point.

In summary, if the application is checking a small number of files, if unlink events take place very frequently, or if the application can poll infrequently, then the user-level implementation can be less expensive. If these conditions are not met, the event grafting approach makes more sense.

Although this experiment focuses on unlink events, the results can be immediately applied to other events, such as the *file modified* event. Although files are modified more often than they are unlinked (and hence the polling frequencies would change), the costs of polling and filtering would be the same, and the measurements found in this section would apply equally as well. In addition, there is no mechanism available in the standard POSIX interface for an application that desires to keep track of events such as *file open* and *file close*. While this experiment concentrated on showing the relative cost of the event graft model, it is important to note that the model also provides new functionality to applications that is not available by other means.

7.4.2 Asynchronous Write Graft

When computing the break-even point in the previous section, I assumed that the only work done was to filter events—that no graft was ever called. In this section I describe an example graft, measure the cost of transaction protection for the graft, and show how the cost of running the graft affects the break-even point. The example graft writes the identity of the unlinked file and the time it was unlinked to a bookkeeping file. If the graft fails, the transaction mechanism cleans up after the graft, undoing or discarding any changes the graft has made, such as writing the record to disk. I measure the cost of running the graft code with and without protection (in order to measure the cost of protection) and at user level (in order to compute a break-even point).

Unlike the grafts in the database study of Section 7.3, this graft mutates the state of the kernel, and thus the transaction protection cost for this graft will be higher than that of the earlier grafts. Specifically, the transaction-protected write function postpones writes until the graft commits, by copying the data to be written to a temporary buffer and adding a write call to the transaction's commit log. This allocation and data-copy overhead will not be incurred by unprotected or user-level versions of the test.

Even though the graft is writing to a disk file, the fact that disk writes are performed asynchronously means the graft does not incur the overhead of a synchronous disk write. In this case, the write can be thought of as a glorified copy operation, in which the data to be written is copied from the caller's buffer to a file system buffer. In the unprotected case, the data is copied directly from the caller's buffer to the file system's buffer; in the protected case, a temporary buffer is allocated, the data is copied to the temporary buffer, and, at commit time, the data is copied from the temporary buffer to the file system buffer and the temporary buffer is deallocated.

I measure the cost of running the graft as unprotected code in the kernel, as protected code in the kernel, and as unprotected code at user level. The results of these measurements are found in Table 21. The text of the unprotected graft is found in Section 7.7.3, and the text of the protected graft and user-level code is found in Section 7.7.4.

The cost of protection is high—the protected code takes nearly three times as long as the unprotected code, because the transaction-protected write function allocates control blocks and a data buffer, and copies the data an extra time.

The cost of the user-level code is much higher than the kernel-level protected code. Because of the way the VINO system call mechanism is implemented, the user-level implementation does nearly all of the work of the protected code (but for one data copy). In addition, it must make two

	Time (μs)
Unprotected	69 (1.0%)
Protected	204 (1.1%)
User-level (VINO)	328 (1.0%)

Table 21: Asynchronous Graft Time
The time to invoke a graft that writes a bookkeeping record to a file. Each result is the 5% trimmed mean of 1000 runs, σ in parentheses.

---

user-kernel round-trips (one for each system call), and traverse the (currently unoptimized) system call code.

With these figures, I can now recompute the break-even point using equation 5. The updated break-even points (based on the measurements from Table 17, Table 18, and Table 21) are found in Table 22.

Number of Files Checked	$Cost_{filter}$ (μs)	$Cost_{poll}$ (μs)	Poll:Unlink Break Even Ratio
1	218	343	3.2:5 (0.64)
10	2056	3373	3:5 (0.61)
50	10222	16882	3:5 (0.60)

Table 22: Break Even Frequency
Given a number of files, the relative frequency of poll operations to unlink operations at which the two approaches break even. These values take into account the cost of writing a bookkeeping record to a file on each event.

Note that, unlike the results in Table 19, the break-even ratio does not change significantly as the number of files increases. Because the time to run the extension is much larger than the time to run a filter or check a file status, the extension time dominates the equation.

There are two important results from this analysis. First, the overhead of the filtering and polling mechanisms has little or no effect on the break-even point, once the cost of a non-trivial operation is taken into account. Second, because the operation cost scales linearly with the number of invocations, the break-even point is independent of the number of invocations.

These calculations tell us how often we can poll from user level, relative to the frequency of unlink events, and still break even. If we assume that we would poll much less frequently than unlink events occur, it would be more efficient for us to poll.

Remember, however, that the functionality offered by the event-based implementation differs from that of the polling implementation. Not only is the graft notified immediately when a file is unlinked, but no unlink events will be lost; in the case of the polling model, we can make a stochastic argument that few unlink events will be lost (approaching zero as the polling frequency approaches infinity), but cannot be guaranteed that events will never be lost. One part of the break-even decision cannot be captured quantitatively, the value of never losing an event. If the occasional loss of an event is not important, the polling mechanism will be more efficient. If it is important to never miss an event, the event-based mechanism should be used.

7.4.3 Synchronous Write Graft

The graft discussed in the previous section does not force the bookkeeping data to disk, instead depending on the kernel to asynchronously write the data to disk. VINO may postpone writing the data to disk for as long as thirty seconds; if the system were to crash in that period, the recently-written data would be lost. The graft can force the data to disk synchronously after each write, although it will incur a considerable cost to do this. I ran modified versions of the tests where the

graft and application force the data to disk; the measurements are found in Table 23. The text of the unprotected graft is found in Section 7.7.5, and the text of the protected graft and user-level code is found in Section 7.7.6.

	Time (μs)
Unprotected	6,966 (7.9%)
Protected	6,998 (7.9%)
User-level (VINO)	8,247 (3.0%)

Table 23: Synchronous Graft Time

The time to invoke a graft that writes a bookkeeping record to a file and forces the record to disk. Each result is the 1% trimmed mean of 1000 runs, σ in parentheses.

Note that the cost of protection is now less than 0.5% (comparing the unprotected and protected cases); the cost of performing a synchronous disk write completely overwhelms the cost of protection.

I now recompute the break-even point, using the figures from Table 23; the results are found in Table 24. As with the previous experiment, the break-even ratio does not change significantly as the number of files increases. Once again, this is because the graft time dwarfs the time required to run a filter or check file status. And because the costs of user-level and protected code are much closer than they were in the previous example, the break-even point is even closer to one. This example reinforces the point that if we were to poll less frequently than unlink events occur, polling is less expensive.

Number of Files Checked	$Cost_{filter}$ (μs)	$Cost_{poll}$ (μs)	**Poll:Unlink Break Even Ratio**
1	7012	8262	17:20 (0.85)
10	69,996	82,563	17:20 (0.85)
50	349,922	412,832	17:20 (0.85)

Table 24: Break Even Frequency

Given a number of files, the relative frequency of poll operations to unlink operations at which the two approaches break even. These values take into account the cost of writing a bookkeeping record to a file on each event.

7.5 File System Layout

The time required to read a file from a disk depends on how the blocks of the file are arranged on the disk. Previous work by Smith and Seltzer [Smith97] showed that the layout algorithm used by the Unix Fast Filesystem (FFS) causes a decrease in read and write throughput for files that are more than twelve blocks in size.[10] A minor change to the layout algorithm (termed “NoSwitch” by Smith and Setlzer) removes much of this performance decrease.

VINO’s default layout algorithm is the same as the one used by FFS. In this study, I repeat Smith’s experiments, showing the baseline performance of the VINO filesystem, the performance of the VINO filesystem with the NoSwitch layout algorithm grafted into the system, and the performance of the VINO filesystem with the NoSwitch layout algorithm with no protection.

10. The first twelve blocks of a file are directly referenced by the file’s i-node (root data structure); all subsequent blocks are referenced by indirect blocks, and are intentionally stored in a section of the disk that is not contiguous with the first twelve blocks, increasing file fragmentation and decreasing file read throughput.

VINO filesystem blocks are 4096 bytes in size, so a file with twelve data blocks is 48 KB in size, or 49,152 bytes. In this experiment I measure the write throughput (grafting overhead) for files of size 64 KB, 128 KB, and 256 KB, and the read throughput of the file system for files of size 32 KB, 64 KB, 128 KB, and 256 KB. (As the graft would not be invoked in the 32 KB cass, it is not necessary to run a 32 KB write test.)

The experiments were run as described in Section 4 of Smith's paper [Smith98]. Each test reads or writes 32 MB of data, 4 MB at a time. The 32 MB of data is divided into the appropriate number of files for the file size being tested (i.e., for 32 KB files, 1024 files are created; for 64 KB files, 512 files are created). The write phase creates and writes the files and the read phase reads the files in the order in which they were created. The in-memory file cache is flushed between phases. The time measured includes the time to create (if necessary), open, and close the files, and in the case of the grafted write throughput test, the time required to load the graft for each file as it is opened and invoke the graft as necessary.

7.5.1 Write Throughput

The write throughput measurements are found in Table 25. In the write tests the performance of the baseline and unprotected tests was very similar, which is as would be expected, given that disk writes are asynchronous, and little extra work is done in the unprotected case. The performance of the grafted case is substantially worse (only 47% of the baseline in the case of 64 KB files).

File Size		Write Throughput (MB/s)		
		Baseline	Grafted	Unprotected
64 KB	raw	0.596	0.282	0.578
(512 files)	normalized	—	47.3%	97.0%
128 KB	raw	0.853	0.486	0.870
(256 files)	normalized	—	57.0%	102.0%
256 KB	raw	0.935	0.662	0.945
(128 files)	normalized	—	70.8%	101.7%

Table 25: File System Write Throughput: Baseline, NoSwitch Grafted, and NoSwitch Unprotected
Each figure is the harmonic mean of ten measurements, $\sigma < 1\%$ in all cases. As with Smith's original tests, the write throughput measurement is based on the end-to-end time, which includes the time to create, open, and close the file, and (in the grafted case) to load the graft.

Given that we saw in the earlier studies that the cost of invoking a graft is very small, why might the the grafted tests perform so poorly here? In the earlier studies, the graft was installed at the beginning of the test and invoked many times over the course of the test. In this study the graft is installed once for each file, and invoked only once each time it is installed (as the file is written). The logical assumption to make is, therefore, that the additional overhead is due to the cost of installing the graft.

To check this assumption, I measured the total time spent in the test and the overhead attributable to installing the graft. These measurements are found in Table 26, where they are compared to measurements of the unprotected test runs. We see that once the installation overhead is factored out, the net run time of the grafted and unprotected cases are equivalent, and hence their throughput is equivalent. The cost of running the graft, therefore, is lost in the noise—graft load time is, as assumed, the culprit.

File Size	Grafted Run Time	Installation Overhead	Net Run Time	Unprotected Run Time
64 KB *(512 files)*	118.2 (1.6%)	59.8 (4.7%)	58.4	56.0
128 KB *(256 files)*	69.3	31.4 (9.0%)	37.9	38.5
256 KB *(128 files)*	50.5	15.4 (10.5%)	35.1	35.3

Table 26: NoSwitch Write Throughput:

Times are reported in seconds. Each figure is the mean of ten measurements, $\sigma < 1\%$ except where noted. We see that the performance difference is attributable to the extra time spent opening the file and installing the graft, as the net run time is equivalent to the unprotected run time.

7.5.2 Read Throughput

The read throughput measurements are found in Table 27. As we can see, the throughput for 32 KB files is effectively the same for the baseline, grafted, and unprotected cases. This is as expected, because the layout of these files is not affected by the NoSwitch policy.

We see higher (better) throughput for files larger than 32KB, the files that benefit from the NoSwitch policy. The grafted and ungrafted cases have roughly the same performance (within 2% of each other), and perform 8% to 11% better than the baseline. Although the throughput improvement is smaller than the improvement seen by Smith (who saw as much as 87%), these results are not directly comparable with his, as important parameters, such as block size, file system size, file system age, and disk hardware, differ between the two experiments.

File Size		Read Throughput (MB/s)		
		Baseline	Grafted	Unprotected
32KB	raw	0.992	0.975	0.985
	normalized	—	98.3%	99.3%
64KB	raw	0.624	0.681	0.673
	normalized	—	109.1%	107.9%
128KB	raw	0.819	0.897	0.910
	normalized	—	109.5%	111.1%
256KB	raw	0.830	0.895	0.896
	normalized	—	107.8%	107.9%

Table 27: File System Read Throughput

File read throughput, in MB/s, for files of varying size. Each figure is the mean of ten measurements, $\sigma < 1\%$in all cases.

7.5.3 File Layout Summary

The cost of grafting the write benchmark is high; in the terminology of Section 7.3, for the write benchmark, *grafting is not worthwhile*. However, the reason to install the layout graft is not to improve write performance, but to improve read performance—which it does, as shown in Table 27. With respect to read performance, because there is no cost associated with the graft at file read time, *grafting is cheap* (in fact, infinitely so). In order to determine the value of grafting the write

policy, the cost at write time (a substantial reduction in throughput) must be weighed against the benefit at read time, and the ratio of reads to writes of this file. If file creation/write and file read occurred equally frequently, the increased cost at write time would not be outweighed by the benefit at read time. If, however, reads occurred much more often than writes, the 10% increase in read throughput could outweigh the 30-50% decrease in write throughput.

7.6 Summary

In the database study, I showed a clear case where the grafting mechanism can improve end-to-end application performance. The grafts loaded into the kernel for this test are small (see below) and can significantly improve application performance (decreasing run time by as much as 30%).

In the event graft study, I showed that the cost of the event mechanism is low, and is insignificant when the cost of running a graft is factored in. Although transaction protection is expensive, running code from user level is more expensive. I computed the break-even point for the two approaches, showing that when it is acceptable to drop events and poll infrequently the polling approach is less expensive. If these conditions are not acceptable, the event-based approach is more appropriate.

In the file system layout study my measurements show that graft load time can be a performance problem; where the earlier studies assumed that grafts were loaded infrequently (and the cost of loading a graft could be amortized over a large number of invocations) in this study the graft was invoked only once each time it was loaded. The cost of graft installation (in this case, roughly 120 milliseconds) was high relative to the immediate gain (little or none). In the database study, where grafts were loaded once, and then used for the duration of the test (roughly 100 seconds); for this application, the graft load time is immaterial.

This serves as a reminder that along with per-invocation costs, there is a fixed overhead associated with grafting; this overhead can be high, relative to any benefit seen, and can not always be hidden or “amortized away.” Although it is likely that some optimization of the graft load code path could be done, there is an intrinsic cost associated with installing a graft; at times, this cost will outweigh the graft’s benefit. As always, the costs and benefits of grafting must be weighed before a decision is made whether a particular graft is worthwhile.

In all of these cases, the amount of code grafted into the kernel is small. In the next section I include the code for the grafts discussed in this chapter. In the following chapter I summarize the contributions of this thesis and discuss future directions for this work.

7.7 Graft Code

Following is the source code for the three grafts discussed in this chapter. Some declarations and error-checking code have been omitted for brevity.

7.7.1 Page Eviction Graft

```
status_t
vas_o::pick_victim(link_o<coremap_entry_o *> *linkp,
                  coremap_entry_o *candidate,
                  coremap_entry_o **choice)
{
    if (ok(candidate)) return (SUCCESS);

    // check each alternate until we find a good one
    while (linkp != NULL) {
        if (ok(linkp->data)) {
            *choice = linkp->data;
            return (SUCCESS);
        }
        linkp = linkp->next;
    }
    return (EINVAL);
}

// make sure it's OK to evict this candidate
bool ok(coremap_entry_o *candidate)
{
    int num_maps, i, j, va[MAX_MAPS];

    // Retrieve all the VA that map this page
    num_maps = bsd_pmap_rev_lookup(NULL, candidate->get_pa(), va,
                                          MAX_MAPS);
    // see if it's mapped at this address
    for (i = 0; i < num_maps; ++i) {
        for (j = 0; j < shared.count; ++j) {
            if (va[i] == shared.keep[j]) {
                return (false);
            }
        }
    }
    return (true);
}
```

7.7.2 Read Ahead Graft

```
status_t
open_file_o::compute_ra(off_t, size_t, off_t *where, size_t *howmuch)
{
    *where = shared.idx[shared.cur++].off;
    *howmuch = 208;
    return ((shared.cur == shared.count) ? SUCCESS : EAGAIN);
}
```

7.7.3 Unprotected Asynchronous Write Graft

```
void
file_unlink(struct file_uid *guy, int size)
{
    // The unprotected graft can use the VINO internal kernel
    // write API, which is has lower overhead than the API used
    // by grafts and user-level applications.

    struct unlink_data data;
    uio uiop;
    iovec iovecp;

    // fill in the bookkeeping record
    data.who = *(struct guy *)arg;
    data.when = the_clock->gettimeofday();

    // construct the control blocks for the file write operation
    iovecp.iov_base = &data;
    iovecp.iov_len = sizeof(data);
    uiop.uio_iov = &iovecp;
    uiop.uio_iovcnt = 1;
    uiop.uio_offset = 0;
    uiop.uio_resid = sizeof(data);
    uiop.uio_segflg = UIO_SYSSPACE;
    uiop.uio_rw = UIO_WRITE;
    uiop.uio_vas = curthread->vas;

    // perform the write operation
    file->write(&uiop);

    return;
}
```

7.7.4 Asynchronous Write Graft

```
void
file_unlink(struct file_uid *guy, int size)
{
    struct unlink_data data;

    // fill in the bookkeeping record
    data.who = *guy;
    gettimeofday(&data.when, NULL);

    // perform the write operation
    write(fd, &data, sizeof(data));

    return;
}
```

7.7.5 Unprotected Synchronous Write Graft

```
void
file_unlink(struct file_uid *guy, int size)
{
    // The unprotected graft can use the VINO internal kernel
    // write API, which is has lower overhead than the API used
    // by grafts and user-level applications.

    struct unlink_data data;
    uio uiop;
    iovec iovecp;

    // fill in the bookkeeping record
    data.who = *(struct guy *)arg;
    data.when = the_clock->gettimeofday();

    // construct the control blocks for the file write operation
    iovecp.iov_base = &data;
    iovecp.iov_len = sizeof(data);
    uiop.uio_iov = &iovecp;
    uiop.uio_iovcnt = 1;
    uiop.uio_offset = 0;
    uiop.uio_resid = sizeof(data);
    uiop.uio_segflg = UIO_SYSSPACE;
    uiop.uio_rw = UIO_WRITE;
    uiop.uio_vas = curthread->vas;

    // perform the write operation and force the modifications to disk
    file->write(&uiop);
    file->sync();

    return;
}
```

7.7.6 Synchronous Write Graft

```
void
file_unlink(struct file_uid *guy, int size)
{
    struct unlink_data data;

    // fill in the bookkeeping record
    data.who = *guy;
    gettimeofday(&data.when, NULL);

    // perform the write operation and force the modifications to disk
    write(fd, &data, sizeof(data));
    fsync(fd);

    return;
}
```

7.7.7 File System Layout Graft

```
status_t
ffs_active_file_o::block_buddy(enum fs_blocktype type, int blockno, int depth,
                              int *near_this)
{
    status_t ret = EINVAL;

    // if the block being allocated is an indirect block, and the associated data
    // block indicates that it's the first indirect block allocated, locate it next
    // to the previous data block. Use bmap() to find the disk location of that block.
    if (type == fs_block_indir && blockno == NDADDR)
            ret = bmap(this, (NDADDR - 1)*fs_bsize, near_this);
    return (ret);
}
```

Chapter 8

Discussion and Conclusions

Dissertations are not finished; they are abandoned.

– attributed to Fred Brooks

8.1 Overview

This thesis opened with the statement that operating systems exist to run applications, and should do the best they can to support these applications. In the preceding chapters I have shown that constructing a safe, extensible operating system is not only tractable, but also affordable, and that applications can garner significant performance gains by using such an operating system. I summarize the results and contributions of the thesis in Section 8.2.

Although VINO implements a POSIX interface and can run standard Unix applications, it is a research system, and, like dissertations and great art, research systems are never finished. In Section 8.3, I discuss the parts of VINO that I would revisit were I starting over, as well as projects that logically follow from the work described here. I conclude in Section 8.4.

8.2 Results and Contributions

The primary results of this work are:

- *Grafting is cheap*: extensions can significantly improve application performance, and the overhead of safety is imperceptible, less than 1%.
- *Sandboxing works here:* the performance results of the MiSFIT tool show that this application of sandboxing is inexpensive and workable.
- *Lightweight transactions are cheap:* the cost of using the VINO Lightweight Transaction mechanism is very small; the overhead of a VLT begin/commit is under 16µs, where the latency of a traditional persistent transaction is bounded by the time to perform a disk operation (in the 5-10 ms range).
- *Software protection makes sense:* when an extension is small, and domain crossings are frequent, software protection is an efficient and useful technique.

- *Java, Modula-3, and sandboxing have comparable overhead:* when comparing Modula-3, MiSFIT, and a modern Java runtime, we see that the overhead of the three technologies is roughly equivalent, adding on the order of 50% overhead.

In short, the results show that VINO can give untrusted applications control over kernel policies through the use of safe extensibility, and that the cost of safety is small enough to be more than compensated by the benefit of the extensions.

8.3 Future Directions

As with any long-term project in computing, the technology has changed since the VINO project's inception, and with it, some of the parameters that helped us decide how to build it. In the first subsection, I discuss how I would revisit the issue of safety and protection given today's technology.

One technical problem that has not been solved to my satisfaction is that of ensuring that grafted code has been processed by MiSFIT. The current method consists of a user-level program (graftsign) computing a digital signature for a graft after it has been processed by MiSFIT, and writing the signature into the graft's object file. VINO recomputes the signature at load time, and compares it to the stored signature; if they match, VINO loads the graft.

This works only if:

- graftsign can be sure that the code it signs was processed by MiSFIT.
- VINO can be sure that the code it receives was processed by graftsign.

Our current model attempts to address the first requirement by assuming the existence of a trusted compilation environment, one in which graftsign will only be run on object code that was processed by MiSFIT before it was assembled. Although this is an appealing model, it is difficult in practice to ensure that the MiSFIT-assembler-graftsign pipeline retains its integrity. To plug the holes, we would need to integrate MiSFIT, the assembler, and graft signing into a single program. Although this solution is technically superior to the one we use now, it creates configuration management and maintenance problems, as we would need to maintain our own version of the assembler.[1]

Our solution to the second problem is weak as well. We currently embed a secret key in the graftsign executable, and assume that it can not be discovered by an untrusted party. In order to keep the key a secret, we must assume that the executable can not be read. If an adversary can find the secret key, the adversary could then trick the kernel into loading arbitrary code as a graft.

Both problems exist because we need to establish a connection between MiSFIT and the code being grafted. There are at least two alternatives for solving these problems; both of them require moving the functionality of MiSFIT into the kernel.

The first one involves moving both MiSFIT and the assembler into the kernel. Applications would then graft assembler source code (rather than object code), and the kernel would MiSFIT and assemble the code before loading it into memory. No signatures would be needed, as the kernel would know that the code was processed by MiSFIT before it was loaded.

The second solution consists of rewriting MiSFIT so that it processes object code instead of source code, and moving the object-code MiSFIT into the kernel. An application would graft unprotected object code, and the kernel would MiSFIT the code as it was loaded into memory. This approach is the one taken by the originators of sandboxing [Wahbe93], and has real technical advantages but, on the x86, thornier implementation issues as well. Modification of x86 binary code is a difficult problem, but some tools exist (e.g., Etch [Romer97]) that would simplify the task.

---

1. Up to now we have not made any major modifications to the tool set; to a large extent, we use the standard GNU versions of the compiler, assembler, and linker.

Another way of solving this problem would be to revisit the choice of safety technology. For example, code written in Java now runs as quickly as code written in Modula-3, or MiSFIT-protected C code. Given Java's overwhelming popular support, if I were starting the VINO project today with the same goals in mind, I would seriously consider writing the entire system in Java. Some of the problems we encountered (such as dangling references and stray pointers) do not come up when using a modern language such as Java, and other nuts-and-bolts pieces of the system that we needed to design and implement (threading, synchronization, dynamic linking, run-time type checking, collection data types) come for free with Java.

Going in a different direction, hardware protection has been shown to be much less expensive than we thought it needed to be. Although, to the best of my knowledge, Leidke's work [Leidke97] has not been independently verified, his results are very encouraging, and imply that it may be possible to build an extensible system like VINO using hardware protection.

Another project worth considering is a study of the costs and benefits of the transaction protection offered by VINO. Up to now, few of the extensions that we have developed have exercised the transaction mechanism. In the general case, it may be that VLTs costs more than they are worth. On the other hand, it may be that our relative inexperience with a transaction-based development model led us to be too cautious, and VINO's structure and performance would benefit from a more aggressive use of transactions. These issues should be studied in more depth.
Although the VINO extensibility model was designed to be general-purpose, its implementation shows a bias towards the support of policy-replacement extensions. Other extensible systems (e.g, SPIN [Bershad95] and Exokernel [Engler95]) do not share this bias, and it is my belief that differences in extensibility architecture induce a differences in how extensions are structured and in which extensions are written. One worthwhile project would be to look at the extensions developed using the other models to determine how well they are supported by the VINO model.

Because VINO is a monolithic operating system at heart—although the implementers attempted to factor out policy decisions—there is a lot of code in the kernel, and much of the system's functionality is hard-coded. Using VINO's extensibility model as a base, another interesting project would be to strip the kernel to minimal scaffolding, and load subsystems (such as the file system and virtual memory system) as extensions. This work would show the limitations of the extensibility model, and generate ideas as to how to augment it to improve its usefulness. Initial work to decouple the subsystems of the kernel (so that a minimal kernel can be built) has been done as part of the eVINO (embedded VINO) project, currently underway at Harvard.

8.4 Conclusion

Although I might have argued at the beginning of this project that we could test the idea of safe extensibility in the context of another operating system, I now understand that there is no more convincing proof of the design of a new operating system (to both the designer and the skeptic) than booting it up on bare hardware and seeing it run real applications.

With this system, the VINO group pulled together ideas from different areas and showed that they worked together well. It is my hope that the unambiguity of the results from our research system will influence the design of new production systems. Giving applications control over the kernel's behavior may be a disturbing thought to us as operating system designers—but, running applications is the *raison d'être* of our work, and we should be open to techniques that help them run better.

Appendix

The Instruction Set Architecture of the Intel x86 Processor Family

The Intel x86 is a complex instruction set (CISC) processor with a limited register set and variable length instructions. When compared to a RISC CPU, the x86 instruction set seems, at best, baroque, with support for memory-to-memory instructions, string copy, search and comparison instructions, BCD arithmetic, and array bounds checking.

There are six general-purpose 32-bit registers (eax, ebx, ecx, edx, esi, and edi). Some of these registers have special uses specified by the hardware (e.g., ecx, esi, and edi are used by the string instructions).

There is hardware support for a stack, indexed by the esp register. In part because of the lack of general purpose registers, the stack push and pop instructions are very efficient, taking only one and two instruction cycles, respectively, in the general case.[1] Simple register-to-register arithmetic and logical operations (e.g., add, sub, and, or) are executed in a single cycle as well. If one of the arguments is in memory (but found in the level one cache), the instruction executes in two cycles.

Condition codes are stored in a flags register. Both saving the flags register (by pushing it on the stack) and loading the flags register (by popping it off the stack) are substantially more expensive than saving and loading a general purpose register (by a factor of four). Although saving the flags register does not appear to require any more work than saving a general purpose register, loading the register may require that the processor change its I/O privilege level, interrupt flag, and trap flag, which are stored in the flags register.[2]

Instructions are variable length and can include embedded immediate operands. Because of this, it is not always possible to determine *a priori* if the target of an indirect jump or call instruction is a valid (legal, permitted) instruction or an invalid instruction (e.g., one that halts the processor). (This is important to the discussion of how the VINO software fault isolation tool handles indirect jumps and calls.)

---

1. Instruction timings here refer to the Intel Pentium processor, a common member of the x86 processor family
2. The I/O privilege level flag determines which processor modes can perform I/O, the interrupt-enable flag determines whether the CPU will accept device interrupts, and the trap flag is used to put the processor in single-step mode for debugging.

Glossary

"When I use a word," Humpty Dumpty said in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."

– Humpty Dumpty [Carroll70]

abort

A transaction that fails is said to abort. When this happens, all changes made in the context of the transaction are discarded, as if the operations performed during the transaction never took place.

ACID properties

The properties of **atomicity, consistency, isolation,** and **durability**, which are typically guaranteed by **transaction** mechanisms.

atomicity

When a **transaction** completes, either all of the changes it makes are applied (on **commit**) or discarded (on **abort**). Also called *failure atomicity.*

break-even point

The point at which the costs associated with two (or more) systems are equal. Used in this thesis to show at what theoretical component performance costs two techniques have equal overall performance costs. Also see **cost-benefit approach**.

consistency

If a system is thought of as a state machine, its states can be partitioned into consistent and inconsistent states. Consistent states are those that satisfy some set of invariants, defined in terms of the system.

For example, in the case of a system call, all locks and other resources used by the operation are relinquished when it completes . Any transient inconsistencies in the state of the system are cleared up when the transaction completes. Temporary buffers allocated for use during the transaction are reclaimed, and locks and all other resources needed solely during the transaction are released.

cost-benefit approach

An approach to research that endeavors to quantitatively relate the costs (in space, time, or other resources) to the benefit (in similar terms) of a technique, algorithm, feature of a system, or system as a whole.

destructive operation

An operation that cannot be undone. In a transaction model, destructive operations must be delayed until transaction commit. Sometimes called a *physical operation*.

durability

The guarantee that the results of an operation will not be lost after the transaction commits. In a system that uses stable storage, the results of the transaction must be written to stable storage for the transaction to be considered durable. If the transaction's operations only modify ephemeral data, the results of the transaction need not be written to stable store to be considered durable. In essence, the results of the transaction need only be made as persistent as the other resources of the base system.

isolation

A property specifying that the intermediate state of the system during a transaction is not visible within other, concurrent transactions. Within each transaction, it appears that transactions are running serially, not concurrently.

If client A and B are running concurrently, client B's view of the system should either be that of the system before client A started, or after client A ends.

transaction

A mechanism by which a system enforces the **ACID properties** for the invocation of a collection of entrypoints by a client.

two-phase locking

A locking protocol that assures the properties of **atomicity** and **isolation** are met. In the first phase, locks are acquired (and none are released). In the second phase, locks are released (and none are acquired).

References

[Accetta86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., "Mach: a New Kernel Foundation for UNIX Development," *Proceedings of the 1986 Summer USENIX Conference* (July 1986).

[Adl-Tabatabai96] Adl-Tabatabi A., Langdale, G., Lucco. S., Wahbe, R., "Efficient and Language-Independent Mobile Programs," *Proceedings of the 1996 Conference on Programming Language Design and Implementation*, pp. 127–136, Philadelphia, PA (May 1996).

[Apple94] Apple Computer, *Inside Macintosh: Operating System Utilities*, Addison-Wesley, Reading, MA (1994).

[Baker91] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System," *Proceedings of the 13th Symposium on Operating Systems Principles,* pp. 198–212, Pacific Grove, CA (October 1991).

[Baker92] Baker, M., Sullivan, M. "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," *Proceedings of the Summer 1992 USENIX Conference*, pp. 31–43, San Antonio, TX (June 1992).

[Banerji96] Banerji, A., Panteleenko, V., Wyant, G., Cohn, D., "Quantitative Analysis of Protection Options," University of Notre Dame Technical Report TR-96-20 (1996).

[Beck98] M. Beck, Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D., *Linux Kernel Internals, Second Edition*, Addison Wesley Longman, Reading, MA (1998).

[Bernstein87] Bernstein, P., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing, Reading, MA (1987).

[Bershad88] Bershad, B., Pinkerton, C. "Watchdogs—Extending the UNIX File System," *Computing Systems 1*, 2, pp. 169–188 (Spring 1988).

[Bershad89] Bershad, B., Anderson, T., Lazowska, E., Levy, H., "Lightweight Remote Procedure Call," *Proceedings of the 11th Symposium on Operating Systems Principles,* pp. 102–113, Litchfield Park, AZ (December 1989).

[Bershad95] 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," *Proceedings of the 15th Symposium on Operating Systems Principles,* Copper Mountain, CO (December 1995).

[Campbell95] Campbell, R., Tan. S., "µChoices: An Object-Oriented Multimedia Operating System," *Proceedings of the Fifth Workshop on Hot Topic in Operating Systems*, pp. 90–94, Orcas Island, WA (May 1995).

[Cao94] Cao, P., Felten, E., Li, K., "Implementation and Performance of Application-Controlled File Caching," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation,* pp. 165-177, Monterey, CA (November 1994).

[Carroll70] Carroll, Lewis, *Through the Looking-Glass,* Macmillan & Co., London, England (1870).

[Chrysanthis90] Chrysanthis, P., Ramamritham, K., "ACTA: A Framework for Specifying and Reasoning about Transaction Structures and Behavior," *Proceedings of the ACM SIGMOD International Conference on Management of Data,* pp. 194–203 (1990).

[Colusa95] "Omniware Technical Overview," Colusa Software, http://www.colusa.com (1995).

[Cormen91] Cormen, T., Leiserson, C., and Rivest, R., *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1991, pp. 232–241.

[Custer93] Custer, H., *Inside Windows NT*, Microsoft Press, Redmond, WA (1993).

[DEC96] Digital Equipment Corporation, *Digital Unix Version 4.0 System Administration*, Order Number AA-PS2RD-TE, Digital Equipment Corporation, Maynard, MA (1996).

[deJonge93] de Jonge, W., Kaashoek, M. F., Hsieh, W., "The Logical Disk: A New Approach to Improving File Systems," *Proceedings of the 14th Symposium on Operating Systems Principles*, pp. 15–28, Asheville, NC (December 1993).

[Dettman89] Dettman, T., *DOS Programmer's Reference, Second Edition*, pp. 321, Que Corporation, Carmel, IN (1989).

[DeWitt84] DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., Wood, D., "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1–8, Boston, MA (June 1984).

[DeWitt91] DeWitt, D., "The Wisconsin Benchmark: Past, Present, and Future," in *The Benchmark Handbook*, J. Gray, ed., Morgan Kaufmann Publishers, San Mateo, CA (1991).

[Druschel93] Druschel, P., Peterson, L., "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proceedings of the 14th Symposium on Operating Systems Principles,* pp. 189–202, Asheville, NC (December 1993).

[Engler95] Engler, D., Kaashoek, M. F., and O'Toole, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proceedings of the 15th Symposium on Operating Systems Principles,* Copper Mountain, CO (December 1995).

[Fall93] Fall, K., Pasquale, J., "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability," *Proceedings of the 1993 Winter USENIX Conference,* pp. 327–334, San Diego, CA (January 1993).

[Feller50] Feller, W., *An Introduction to Probability Theory and Its Applications, Third Edition*, pp. 224–225, John Wiley & Sons, New York, NY (1950).

[Ford4] Ford, B., Lepreau, J, "Evolving Mach 3.0 to a Migrating Thread Model," *Proceedings of the 1994 Winter USENIX Conference*, pp. 97-114, San Francisco, CA (January 1994).

[Ford98] Ford, B., Hibler, M., Lepreau, J., McGrath, R., Tullmann, P., "Interface and Execution Models in the Fluke Kernel," Univeristy of Utah Computer Science Technical Report UUCS-98-013 (August 1998).

[Garcia-Molina87] Garcia-Molina, H., Salem, K., "Sagas,*" Proceedings of the ACM Conference on Management of Data*, pp. 249–259, San Francisco, CA (May 1987).

[Gawlick85] Gawlick, D., Kinkade, D., "Varieties of Concurrency Control in IMS/VS FastPath," *IEEE Database Engineering 8*, 2, pp. 3–10 (1985).

[Ghosh98] Ghosh, N., *The Network Architecture of the VINO Operating System*, A.B. thesis, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA (1998).

[Gosling96] Gosling, J., Joy, W., Steele, G., *The Java Language Specification*, Addison-Wesley, Reading MA (1996).

[Gray93] Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Francisco, CA (1993).

[Guillemont91] Guillemont, M., Lipkis, J., Orr, D., Rozier, M., "A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility," *Proceedings of the 1991 Winter USENIX Conference,* pp. 13–21, Dallas, TX (January 1991).

[Härtig97] Härtig, H., Homuth, M, Leidke, J., Schönberg, S., Wolter, J. "The Performance of μ-Kernel-Based Systems," *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pp. 66–77, St. Malo, France (October 1997).

[Hitz94] Hitz, D., Lau,. J., Malcolm, M., "File System Design for an NFS File Server Appliance," *Proceedings of the 1994 Winter USENIX Conference*, pp. 235–245, San Francisco, CA (January 1994).

[Hölze94] Hölze, U., Ungar, D., "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback," *Proceedings of the 1994 SIGPLAN Conference on Programming Language Design and Implementation,* Orlando, FL (June 1994).

[Jones96] Jones, R., Lins, R., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, Chichester, England (1996).

[Kimbrel96] Kimbrel, T., Tomkins, A., Patterson, R. H., Bershad, B., Cao, P., Felten, E., Gibson, G., Karlin, A., Li, K., "A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching," *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 19–34, Seattle, WA (October 1996).

[Kung83] Kung, H. T., Papadimitriou, C. H., "An Optimality Theory of Concurrency Control for Databases," *Acta Informatica 19*, pp. 1–11 (1983).

[Lee94] Lee, C.-H., Chen, M., Chang, R. C., "HiPEC: High Performance External Virtual Memory Caching," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation,* pp. 153–164, Monterey, CA (November 1994).

[Leidke97] Leidke, J., Elphinstone, K., Schönberg, S., Härtig, H., Heiser, G., Islam, N., Jaeger, T., "Achieved IPC Performance," *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pp. 28–31, Cape Cod, MA (May 1997).

[Lindholm97] Lindholm, T., Yellin, F., *The Java Virtual Machine Specification*, Addison-Wesley, Reading MA (1997).

[Liskov91] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., Williams, M., "Replication in the Harp File System," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles,* pp. 226–238*,* Pacific Grove, CA (October 1991).

[Liskov95] Liskov, B., Curtis, D., Day, M., Ghemaway, S., Gruber, R., Johnson. P., Myers, A., "Theta Reference Manual," MIT LCS Programming Methodology Group Memo 88 (February 1995).

[Lowell97] Lowell, D., Chen, P., "Free Transactions With Rio Vista," *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles,* pp. 92–101*,* St. Malo, France (October 1997).

[May87] May, C., "MIMIC: A Fast System/370 Simulator," *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices 22*, 7, pp. 1-13, St. Paul, MN (July 1987).

[McCanne93] McCanne, S., Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *1993 Winter USENIX Conference,* San Diego, CA (January 1993).

[McKusick96] McKusick, M. K., Bostic, K., Karels, M., Quarterman, J., *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Publishing Company, Reading, MA (1996).

[McVoy96] McVoy, L., Staelin, C., "lmbench: Portable Tools for Performance Analysis," *1996 USENIX Conference*, pp. 279–294, San Diego, CA (January 1994).

[Microsoft95] Microsoft Corporation, *Microsoft Windows NT 3.51 Resource Kit*, Microsoft Press, Redmond, WA (1995).

[Mogul87] Mogul, J., Rashid, R., Accetta, M., "The Packet Filter: An Efficient Mechanism for User-level Network Code," *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 39–52, Austin, TX (November 1987).

[Moss85] Moss, J. E. B., *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, MA (1985).

[Mössenböck94] Mössenböck, H. "Extensibility in the Oberon System," *Nordic Journal of Computing 1*, 1, pp. 77–93 (Spring 1994).

[Mowry96] Mowry, T., Demke, A., Krieger, O., "Automatic Compiler-Inserted I/O Prefetching for Out-Of-Core Applications," *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 3–18, Seattle, WA (October 1996).

[Necula96] Necula, G., Lee, P., "Safe Kernel Extensions Without Run-Time Checking," *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 229–243, Seattle, WA (October 1996).

[Nelson91] Nelson, G., ed., *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ (1991).

[Patterson95] Patterson, R. H., Gibson, G., Ginting, E., Stodolsky, D., Zelenka, J., "Informed Prefetching and Caching," *Proceedings of the 15th Symposium on Operating Systems Principles*, pp. 79–95, Copper Mountain, CO (1995).

[Pike97] Pike, R., Trickey, H. *Inferno Programming*, http://inferno.bell-labs.com/inferno/book (1997).

[Rees86] Rees, J., Levine, P., Mishkin, N., Leach, P., "An Extensible I/O System," *Proceedings of the Summer 1986 USENIX Conference*, pp. 114–125, Atlanta, GA (1986).

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm," *Network Working Group RFC 1321* (April 1992).

[Ritchie84] Ritchie, D., "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal 63,* 8 pp. 1897–1910 (October 1984).

[Romer97] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., Chen, B., "Instrumentation and Optimization of Win32/Intel Executables Using Etch," *Proceedings of the USENIX Windows NT Workshop*, pp. 1–8, Seattle, WA (August 1997).

[Rosenblum92] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 1–15, Pacific Grove, CA (October 1991).

[Schmuck91] Schmuck, F., Wyllie, J., "Experience with Transactions in Quicksilver," *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 239–253, Pacific Grove, CA (October 1991).

[Schneier96] Schneier, B., *Applied Cryptography, Second Edition*, John Wiley & Sons, New York, NY (1996).

[Scriptics] ftp://ftp.scriptics.com/pub/tcl/tcl8_1

[Seidl97] Seidl, M., Zorn, B., "Predicting References to Dynamically Allocated Objects," University of Colorado Technical Report CU-CS-826-97 (1997).

[Seltzer93] Seltzer, M., Bostic, K., McKusick, M., Staelin, C., "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the 1993 Winter Usenix Conference*, pp. 307–327, San Diego, CA (1993).

[Seltzer96] Seltzer, M., Endo, Y., Small, C., Smith, K., "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pp. 203–216, Seattle, WA (October 1996).

[Silver96] Silver, S., "Implementation and Analysis of Software-Based Fault Isolation," Dartmouth College Technical Report PCS-TR96-287, 1996.

[Sirer96] Sirer, E. G., Savage, S., Pardyak, P., DeFouw, G., Alapat, A., Bershad, B., "Writing an Operating System with Modula-3," *Workshop Record of the Inaugural Workshop on Compiler Support for Systems Software (WCSSS'96)*, Tucson, AZ (February 1996).

[Small94] Small, C., Seltzer, M., "VINO: An Integrated Platform for Operating System and Database Research," *Harvard University Computer Science Technical Report TR-30-94* (1994).

[Small95] Small, C., Seltzer, M., "Structuring the Kernel as a Toolkit of Extensible, Reusable Components," *Proceedings of the Fourth International Workshop on Object Orientation in Operating Systems*, pp. 134–137, Lund, Sweden (August 1995).

[Small96] Small, C., Seltzer, M., "A Comparison of Operating System Extension Technologies," *Proceedings of the USENIX 1996 Annual Technical Conference*, pp. 41–54, San Diego, CA (January 1996).

[Smith97] Smith, K., Seltzer, M., "File System Aging–Increasing the Relevance of File System Benchmarks," *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 203–213, Seattle, WA (June 1997).

[SunJavaBugs] http://developer.javasoft.com/developer/bugParade/index.html

[Terry95] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles,* pp. 172–183, Copper Mountain, CO (December 1995).

[Thekkath94] Thekkath, C., Levy, H., "Hardware and Software Support for Efficient Exception Handling," *Proceedings of the Sixth Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 110-119, San Jose, CA (October 1994).

[TPC90] Transaction Processing Performance Council, "TPC Benchmark B," Standard Specification, Waterside Associates, Fremont, CA (1990).

[Vahalia96] Vahalia, U., *Unix Internals: The New Frontiers*, Prentice-Hall, Upper Saddle River, NJ (1996).

[Wahbe93] Wahbe, R., Lucco, S., Anderson, T., Graham, S., "Efficient Software-Based Fault Isolation," *Proceedings of the Fourteenth Symposium on Operating Systems Principles,* pp. 203–216, Asheville, NC (December 1993).

[Waldspurger94] Waldspurger, C., Weihl, W., "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pp. 1–12, Monterey, CA (November 1994).