Exodisk: maximizing application control over storage management

by

Robert Grimm

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 1996

Author____

Department of Electrical Engineering and Computer Science
May 17, 1996

Certified b ____

Gregory R. Ganger
Thesis Supervisor

Certified b____

M. Frans Kaashoek
Thesis Supervisor

Acc ____

Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Exodisk: maximizing application control over storage management

by

Robert Grimm

Submitted to the
Department of Electrical Engineering and Computer Science

May 17, 1996

In Partial Fulfillment for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a new disk system architecture to address the growing performance gap between CPU speed and disk access times. Based on the observation that disk access times can greatly benefit from application-specific resource management, the exodisk system separates resource protection and resource management. It safely and efficiently space-multiplexes disk storage among applications and avoids resource management wherever possible. It exports the raw disk storage using a two-level model: On the first level, it provides applications with self-protected extents of contiguous disk sectors. On the second level, a lightweight *i*-node structure is supported that enables applications to efficiently combine several extents in one larger unit and to co-locate their own application-defined data with exodisk system meta-data. The overhead of storage protection is minimal (less than 10%, and often in the noise) for most operations and data organizations. The flexibility of the exodisk system is illustrated with fast file insertion, disk-directed I/O and application-specific persistence.

Thesis Supervisor: Gregory R. Ganger
Title: Postdoctoral Associate

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor

Acknowledgments

The work presented in this thesis is joint work with Greg Ganger, Frans Kaashoek and Dawson Engler [15]. It is based on initial ideas by Dawson Engler, Frans Kaashoek and Tom Pinckney which were further developed in intensive discussions with Hector Briceno, Dawson Engler, Greg Ganger, Frans Kaashoek, David Mazieres and Tom Pinckney.

I thank Marc Fiuczynski at the University of Washington for realizing the full value of descendant exonodes and Greg Ganger for implementing parts of the Aegis disk drivers.

This work was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985 and by a NSF National Young Investigator Award.

I thank Frans Kaashoek for introducing me to the joys (and pains) of systems research and for providing invaluable input and advice throughout the last two years at MIT, including and especially on this thesis. I thank Greg Ganger for his input on all aspects of this thesis. I really appreciate his patience and insight when discussing numerous design and implementation details with him.

I thank the members of the Parallel and Distributed Operating Systems Group at the MIT Laboratory for Computer Science for their support and advice and for providing me with an ideal research environment.

A special thank-you goes to my parents for their love and support, and for making it possible for me to attend MIT. Another special thank-you goes to my friends, especially Petra Zaus, for "being there" over the years. Thank you to the brothers and sisters at my fraternity house, Delta Psi, without whom I probably would not have stayed at MIT (though I did enjoy moving into my own place a year ago).

Last (and least), Tom and Hector: Keep this office a haven of chaos and snide humor!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While CPU speed has been increasing dramatically over the last decade, disk access times have only improved slowly. Disk I/O thus presents a major bottleneck in many computer systems. This thesis explores a new disk system architecture which, in contrast to traditional storage systems, provides resource *protection* (who can access what data in which way) but leaves resource *management* (how data is organized, allocated, accessed and cached) to applications. Applications can thus implement their own specialized disk storage abstractions on top of the disk system and (potentially) achieve better performance and greater flexibility than in current system architectures. The disk system design is guided by the exokernel design principles [11], and the prototype is implemented as an in-kernel component of the Aegis exokernel operating system [10, 11]. However, it could just as easily be used in other operating systems.

Exodisk, the disk system described in this thesis, safely and efficiently *space*-multiplexes disk storage among applications at a very fine grain (down to a single disk sector), while avoiding resource management wherever possible. The exodisk system exports raw disk storage using a two-level model: On the first level, it provides applications with self-protected extents of contiguous disk sectors. On the second level, a lightweight *i*-node structure is supported that enables applications to efficiently combine several extents in one larger unit and to co-locate their own application-defined data with exodisk system metadata (e.g., protection information). A prototype implementation of the exodisk system shows that overhead for protection is minimal (less than 10%, and often in the noise) for most operations and organizations when compared to the performance of the raw Aegis disk driver. The flexibility of the exodisk system is illustrated with fast file insertion, disk-directed I/O [26] and application-specific persistence.

1.1 Problem and Motivation

CPU speed, network bandwidth, and primary and secondary storage capacities have increased rapidly over the last decade. At the same time, disk access times have only improved slowly [36, 38]. Disk I/O now presents a major bottleneck in many computer systems and will most likely continue to do so [43]. This gap between the performance of the main computer system and the speed of disk access is further exacerbated by changing computer usage in increasingly networked computing environments: The growing demand for file servers and world-wide web servers, as well as storage-intensive content types, such as digital sound and video or three-dimensional graphics (e.g., VRML), results in the need for more flexible and efficient disk storage management.

Research on disk storage management provides ample evidence that disk access times can greatly benefit from application-specific resource management: Stonebraker [50] argues that inappropriate file system implementations can have considerable impact on the performance of database managers. Cao *et al.* [3, 2] show that application-level control over file caching, prefetching and disk scheduling policies can reduce average application running time by 46%. Patterson *et al.* [39] use application provided access patterns to dynamically regulate prefetching and caching, and show that application running time can be reduced by up to 42%. Seltzer *et al.* show in [48], by comparing the 4.4BSD log-structured file system (which is based on ideas first explored in Sprite LFS [42, 44]) and the BSD fast file system [29, 41, 27], that the highest performing data organization is highly workload dependent. Temporary, memory-based file systems [30, 35] have been added to many systems because conventional file creation/deletion semantics are too heavy-weight for some applications, costing more than a factor of two in application performance [16]. All of these examples show that storage management should be tailored to the needs of individual applications to get the best performance.

1.2 Solution

To effectively provide application control over storage management it is necessary to move away from general high-level storage abstractions (as provided, for example, by file systems). Fixed high-level abstractions can hurt application performance, since there is no single way to abstract disk storage or to implement a disk storage abstraction that is best for all applications and all workloads (as discussed above). Fixed high-level abstractions also hide information from applications, which they might need to implement their own resource management. For example, applications need information about the exact layout of data and metadata on disk to implement their own allocation policies, but this information is usually hidden within the abstraction of a file system. Furthermore, fixed high-level abstractions limit the functionality of applications, because they define the only interface to disk storage. For example, an application, even if it knows the exact layout of data and metadata on disk, can not implement its own allocation policy, since a standard policy is usually hard-wired into the file system and can not be modified. The "end-to-end" argument [45] thus applies to disk storage management as well as low-level communication protocols: Applications know better than operating systems which storage abstractions they require and how these abstractions should be managed. The overriding goal of a storage system should thus be to maximize application control over storage management.

The exodisk system thus provides a low-level storage architecture in which applications can utilize their own disk management policies. The exodisk system space-multiplexes disk storage among applications at a very fine grain, that is at the level of one or more contiguous disk sectors. While separate raw disk partitions allow for different storage managers and on-disk data layouts in conventional systems, they have two major limitations: First, partitions must be statically allocated when a disk drive is formatted. Changing them later entails loosing all on-disk data. Second, partitions introduce a considerable performance overhead, due to seeks, when data in separate partitions is accessed concurrently. For example, when interleaving the data of two concurrently running applications that write and read 1,000 10 KByte files each, the overall application latency is reduced by 34% when compared to placing the data 250 MByte apart. In contrast, fine grain multiplexing avoids both the inflexibility of statically allocated partitions and the performance overhead associated with long seeks between partitions.

The exodisk system protects disk storage through a two-level protection model: On the first level, the exodisk system provides self-protected extents of one or more contiguous disk sectors. Self-protected extents belonging to different applications or file systems can be arbitrarily interleaved on disk. On the second level, several extents can be combined in a larger unit, called an *exonode*. Applications can also use exonodes to co-locate their own application-defined data with exodisk system metadata. Exonodes can be viewed as exodisk system *i*-nodes, which differ from conventional file system *i*-nodes in that they can be sized and configured to suit application needs. Applications can build their own storage abstractions (that best fit their needs) on top of this simple storage system.

1.3 Prototype Implementation

The prototype implementation of the exodisk system is implemented as a component of the Aegis exokernel and runs on DECstation 3100 and 5000 computers. A set of micro-benchmarks is used to measure the cost of exodisk system protection for both small and large file I/O, and to determine the benefits of fine-grained interleaving. The benefits of application-specific storage management are exemplified by three experiments: (1) A fast file *insert* operation that exploits the freedom to directly manipulate disk system metadata to perform up to a factor of six better than insertion by copying (for large files). (2) Disk-directed I/O, instead of issuing one disk request at a time and waiting for it to complete, issues several requests without synchronizing, allows the disk to service them in any order it wants, and processes the requests as they complete. For random I/O and C-LOOK disk scheduling, it processes 39% more requests per unit of time than synchronous I/O when issuing 128 requests at a time. (3) Application-specific persistence models allow application-writers to make the trade-off between persistence guarantees and application performance on a per-file basis and greatly increase the flexibility of applications.

1.4 Exokernels

The exodisk system is designed as a component of Aegis, an exokernel operating system [10, 11]. This new operating system architecture is based on the observation that the high level of abstraction provided by most operating systems severely limits the flexibility of application designers, discourages changes in the implementation of existing abstractions (if possible

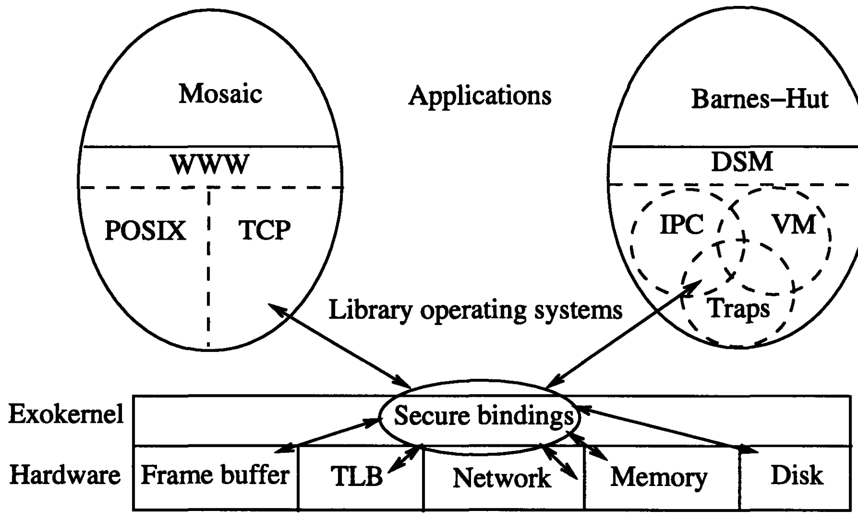

Figure 1-1: An Example Exokernel-Based System.

at all) and denies applications the advantages of application-specific optimizations. The exokernel architecture instead supports application-level, untrusted resource management on top of a minimal kernel that provides applications with secure access to the underlying physical resources. Higher-level abstractions are provided by library operating systems that work above the exokernel interface.

The exokernel architecture consists of a thin exokernel veneer that multiplexes and exports physical resources securely through a set of low-level primitives. Library operating systems then use these low-level primitives and implement their own higher-level abstractions on top of the exokernel interface. Library operating systems can provide standardized abstractions and interfaces (for example, a POSIX compliant library operating system) or they may be specialized for a particular application, providing an abstraction that better meets the particular performance and functionality goals of that application.

Figure 1-1 (which is the same as Figure 1 in [11]) illustrates an example exokernel-based system. The exokernel exports the physical resources (such as network and memory) to library operating systems through secure bindings. Each library operating system implements its own system abstractions and resource management policies. Applications link against standard libraries (e.g., WWW, POSIX, and TCP libraries for World-Wide Web applications) or against specialized libraries (e.g., a distributed shared memory library for parallel applications).

1.5 Thesis Overview

Chapter 2 discusses the design of the exodisk system. It presents the design principles of the exodisk system, gives an overview of the major components of the exodisk system and discusses several issues related to multiplexing disk storage. Chapter 3 presents a prototype implementation of the exodisk system within the Aegis exokernel. Chapter 4 discusses how clients can use the exodisk system. It gives examples of how storage managers can be implemented on top of the exodisk system, shows how applications can share on-disk data and how unused on-disk data can be reclaimed. Chapter 5 presents an evaluation of the exodisk system prototype using a set of micro-benchmarks. Chapter 6 discusses related work, and Chapter 7 concludes this thesis. Appendix A lists the numerical results for some of the experiments discussed in Chapter 5.

Chapter 2

Exodisk System Design

This chapter describes the design of the exodisk system. Section 2.1 presents the design principles and gives an overview of the major components of the exodisk system. Section 2.2 discusses persistence of secure bindings in the exodisk system. Section 2.3 discusses several issues related to multiplexing disk storage. Section 2.4 identifies the limitations of the exodisk system. Finally, Section 2.5 summarizes the exodisk system design.

2.1 Basic Design

The fundamental goal of the exodisk system is to maximize application control over disk storage management. The following principles, which are derived from this fundamental goal, guide the design of the exodisk system (this discussion follows the discussion of the general exokernel design principles in [11]). Each principle is presented along with the related components of the exodisk system. While the exodisk system is designed as a component of an exokernel operating system, it could be used as the disk system for any operating system.

2.1.1 Securely Expose Disk Hardware

The exodisk system uses a two-level protection model to securely expose the raw disk capacity. On the first level, disk storage is multiplexed among applications at a very fine grain, that is at the level of a disk sector and of contiguous extents of disk sectors. Fine grain multiplexing avoids both the inflexibility of statically allocated partitions and the performance overhead associated with long seeks between partitions. Each such extent (a

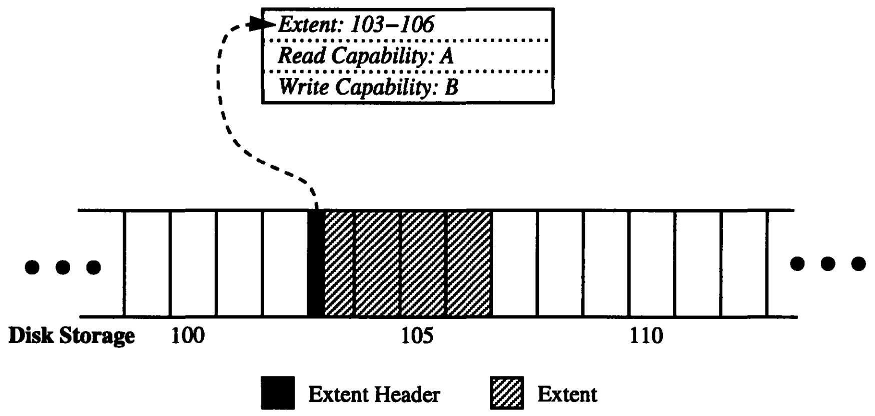

Figure 2-1: An Example Use of a Self-Protected Extent.

single sector is an extent of length one) is protected by self-authenticating capabilities [4] for *read* and *write* access. Such an extent is called a *self-protected extent.* The metadata necessary to support this mapping from self-protected extents to capabilities is stored at the beginning of each individual extent within the *extent header.* An application accesses data within a self-protected extent by presenting the exodisk system with the start sector of the self-protected extent on disk (which uniquely identifies each self-protected extent), the correct capability and the range of sectors to be accessed. If the capability matches the corresponding capability in the extent header (and the range of sectors is part of the extent), the requested data is read or written. If the capabilities do not match, the exodisk system returns an error code to the application and terminates the request. Applications can not access extent headers directly, but only through system calls (which ensures that the protection model can not be violated by applications).

If the disk system client does not want to shift all user data by the length of this metadata, it can store the user data that is overwritten by the exodisk metadata at some other location. For example, an application could maintain a list (perhaps in another extent) of all extents that it uses to store data. In this list, the application could record the beginning and length of each extent and the associated capabilities. It could also store the data overwritten by the extent header in each list entry.

Figure 2-1 illustrates the use of self-protected extents in the exodisk system. It shows an extent from Sector 103 to Sector 106. The extent is protected by the *read* capability A and the *write* capability B. The exodisk system metadata for this extent is stored in the extent header at the beginning of the extent in Sector 103.

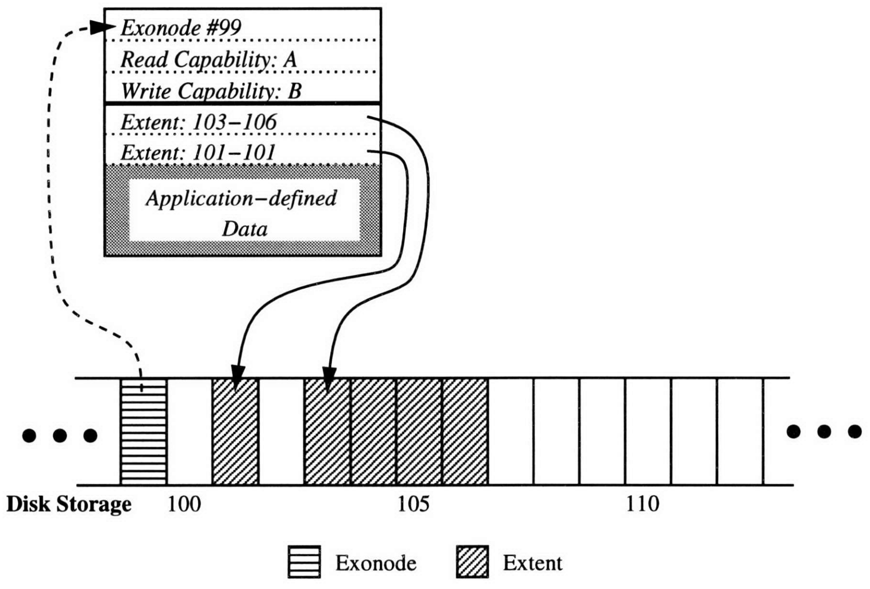

Figure 2-2: An Example Use of an Exonode.

On the second level, several extents can be combined into one larger logical unit protected by a single pair of (self-authenticating) capabilities. Such a logical unit is called an *exonode*, and represents an *i*-node implementation within the exodisk system. The list of extents within an exonode can be augmented with arbitrary-length application-defined data, allowing the co-location of application-level metadata (e.g., link counts and modification times) and exodisk system metadata (i.e., capabilities). The exodisk metadata necessary to support this mapping from collections of extents to capabilities is not stored within the individual extents (i.e., within the extent headers) but within the exonode. The individual extents thus do not have extent headers. The exonode itself is stored in its own, separate disk location. An exonode and the data protected by it are accessed by presenting the exodisk system with the address of the exonode (which uniquely identifies it), the correct capability and the range of sectors to be accessed. Similarly to self-protected extents, if the capability matches the corresponding capability in the exonode (and the sectors to be accessed are protected by the exonode), the requested data is read or written. If the capabilities do not match, the exodisk system returns an error code to the application and terminates the request. Applications can not access exonodes directly, but only through system calls.

Figure 2-2 illustrates the use of exonodes in the exodisk system. It shows an exonode in Sector 99 that is protected by the *read* capability A and the *write* capability B. The exonode entries point to an extent from Sector 103 to Sector 106 and to an extent in Sector 101. The exonode also contains a region of application-defined data.

The second level of the exodisk system is not strictly necessary to securely multiplex raw disk storage among applications. However, it presents an important optimization to effectively interleave file systems and other storage managers on top of the exodisk system. Exonodes provide a useful and efficient mechanism (applications do not need to pay the overhead of per-extent protection) to group several extents in a larger unit while allowing for finely interleaved file systems. Furthermore, the ability to augment exonodes with arbitrary-length application-defined data enables applications to co-locate metadata associated with their own storage abstractions with exodisk system metadata. Since the metadata of all storage system layers can be stored in close proximity, it is possible to avoid maintaining several separate tree-like mappings from disk storage to metadata as described by Stonebraker [50]. Exonodes also support file system optimizations such as immediate files [33] which store the data of small files in the *i*-node.

Exonodes support references to *descendant exonodes*, which are protected by their own, possibly different, capabilities. Their intended use is to create hierarchical protection spaces, possibly shared by several applications. If an application has *read* or *write* access to an "ancestor" exonode that (through any number of references to descendant exonodes) is connected to a descendant exonode, it also has the same access right to the descendant exonode. However, a descendant exonode does not validate access for an ancestor exonode. The exodisk system lets the application discover the corresponding capabilities for the descendant exonode, even if the application does not know them originally. Descendant exonodes also allow the integration of conventional Unix-style *i*-nodes (using descendant exonodes with the same capabilities as the ancestor exonode to represent single, double and triple indirect blocks) with the exonode design. The use of descendant exonodes is expanded on in Section 4.3.

Figure 2-3 illustrates the use of descendant exonodes in the exodisk system. The exonode in Sector 99 is the same as the exonode in Figure 2-2, except that it has another entry pointing to a descendant exonode in Sector 107. The descendant exonode points to an extent in Sector 111 and is protected by the *read* capability C and the *write* capability D.

The exodisk system also supports another type of exonode, called a *proxy exonode.* Proxy exonodes are not persistent over system reboots. Their intended use is for the

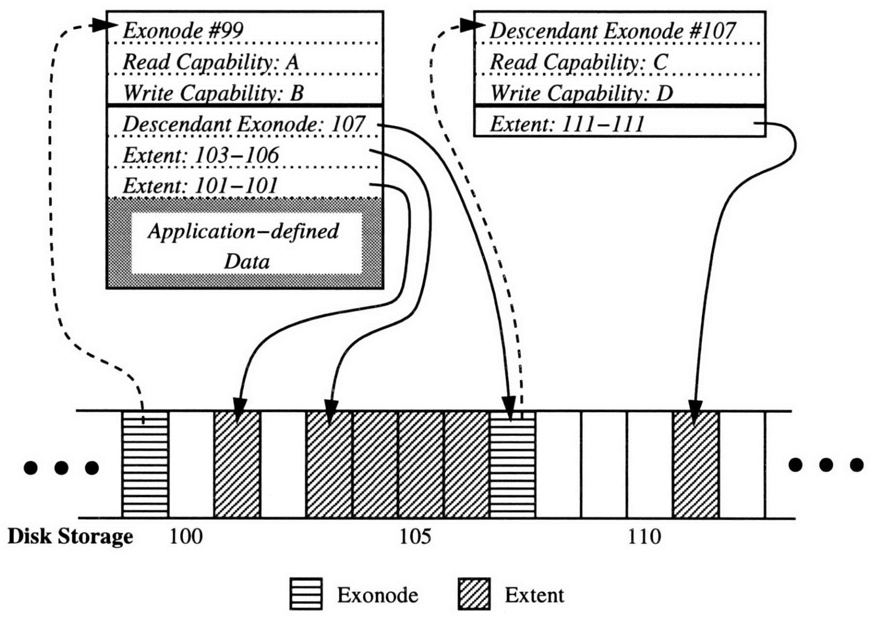

Figure 2-3: An Example Use of a Descendant Exonode.

temporary transfer of access rights to (full or partial) extents or exonodes. By using a proxy exonode to transfer access rights, the original owner of the extent or exonode can avoid revealing the on-disk capabilities to other applications and can restrict access to parts of the on-disk data described by the original extent or exonode. For example, if an application maintains a large database on disk, it can make selected parts of this database accessible to clients using a proxy exonode. Proxy exonodes are part of the exodisk system to simplify the (temporary) sharing of on-disk data between applications (see Section 4.2.2).

Figure 2-4 illustrates the use of proxy exonodes in the exodisk system. The exonode in Sector 99 is the same as the exonode in Figure 2-3, except that, instead of pointing to a descendant exonode in Sector 107, it points to another extent from Sector 108 to Sector 111. The proxy exonode is not associated with any sector on disk, but is only maintained within non-persistent state of the exodisk system. It has one entry that points to an extent from Sector 109 to Sector 111, and thus protects a subset of the sectors protected by the exonode in Sector 99. The proxy exonode is protected by the *read* capability C and the *write* capability D. The back-pointer from the proxy exonode to the exonode in sector 99 can be used by applications to determine from which exonode the proxy exonode was derived. It is also used by the exodisk system to ensure the invariant that a proxy exonode can only point to a subset of the data described by the original extent or exonode (see Section 2.2).

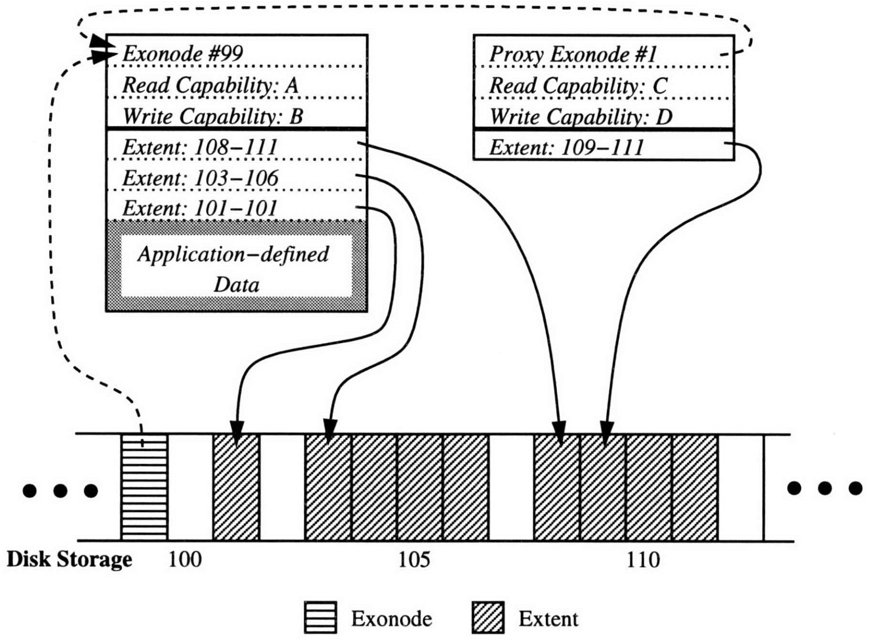

Figure 2-4: An Example Use of a Proxy Exonode.

2.1.2 Expose Disk Allocation

A sector must be explicitly allocated before it can be added to a self-protected extent or exonode. When a sector is no longer required to store data, it needs to be deallocated. To allocate disk space, an application asks the exodisk system for specific disk locations. If any of the requested sectors are already allocated, the exodisk system fails the allocation request. Otherwise, the disk space is associated with a (possibly new) self-protected extent or exonode, which establishes a *secure binding* [10]. The capabilities used to protect the self-protected extent or exonode are provided by the application.

The exodisk system uses a free-list to make the current status of disk allocation visible to all disk system clients. The free-list is simply a map of tags, with one tag corresponding to each physical sector on disk. Each tag indicates whether the sector is free or allocated, where allocated sectors may additionally be distinguished between data and special sectors, i.e. extent headers and exonodes. The prototype implementation of the exodisk system uses a free-list with one bit per sector, indicating whether or not the sector is currently allocated.

2.1.3 Expose Physical Name-Space

The exodisk system uses physical sector numbers, that is the name-space exported by the disk hardware. Self-protected extents and exonodes are addressed by their starting sector numbers on disk. More precisely, because most modern disk drives use a complicated internal mapping, the sector numbers used are those provided by the disk drive interface. The use of the exported name-space avoids introducing another level of indirection with logical block numbers. The current status of the name-space is application-visible through the free-list. The exodisk system also makes its view of the current disk position (and the queue of pending disk operations) visible to applications.

The physical name-space of the exodisk system in combination with explicit allocation enables applications to set their own storage allocation policies and optimizations. This approach prevents the exodisk system from transparently reorganizing data on disk or avoiding block-overwrite semantics, as is done in some logical representations of disk storage (for example, in Mime [7] or the Logical Disk [8]). But it also does not suffer from the overheads associated with such logical-to-physical mappings, and it gives applications the flexibility to utilize the policies that are best suited for the given application. Furthermore, if desired, logical representations of disk storage can be implemented on top of the exodisk system (in user-space).

2.1.4 Avoid Resource Policy Decisions

The exodisk system lets applications determine their own allocation, prefetch, access and cache policies. Through the use of the physically-named free-list, disk position information and explicit allocation, applications can determine the layout of data on disk which is optimal for that particular application's workload. Applications can further decide when and how to prefetch and cache data. If applications want to share on-disk data, they can agree on a common layout of the on-disk data. At the same time, they can still implement their own, individual storage management policies (for example, what data to cache) on top of the storage protection provided by the exodisk system without relying on a storage server. The unit of allocation/access in the exodisk system is the same as that exported by most modern disks, that is the size of an individual sector, avoiding the policy decision inherent in other block sizes. It is therefore possible to efficiently interleave several file systems on disk as well as to stripe over several disks, thus allowing for multiple file implementations as suggested in [49] and for separation of control and data storage as suggested in [34].

In order to efficiently map disk sectors to capabilities, the exodisk system needs to maintain a cache of extent headers and exonodes *within* kernel memory. The exodisk system gives applications a high degree of control over this *exodisk cache*: Applications determine which extent headers and exonodes are cached and when they are written back to disk storage. The in-kernel exodisk cache in the prototype implementation is a very simple, statically-sized cache with least recently used (LRU) replacement, and it is shared among all applications. It allows an application to mark an extent header or exonode as least recently used, so as to have some limited control over the cache replacement policy. The lack of full control over the exodisk cache can cause difficulties for application-writers: Since the exodisk cache is shared among applications, an application must test if a specific extent header or exonode is in the exodisk cache before using it (and hope that it does not get evicted between test and use). Furthermore, an application may be forced to write out a dirty extent header or exonode (before it wants to write it out) to allow for replacement. A future design should thus shift to an architecture where applications have *full* control over when to cache which extent headers and exonodes (independent of the caching needs of other applications) and over how much space to devote to the cache. Such a design is described in more detail in Section 2.3.3.

2.1.5 Design Summary

The four design principles and their application in the exodisk system are summarized in Table 2.1. The exodisk system protects disk space using self-protected extents and exonodes. Both self-protected extents and exonodes need to be explicitly allocated and deallocated by applications. The current global state of allocation is exposed through a free-list. Sectors are addressed by their physical sector numbers. Based on the resource protection provided by the exodisk system, applications determine their own allocation, prefetch and cache policies.

2.2 Persistence of Secure Bindings

The design of the exodisk system faces a problem unique (among computer system components) to storage management: persistence. Other physical resources in a computer system are typically allocated for either the lifetime of the application or even smaller periods of time. If the application exits or the system shuts down, it also relinquishes all access to such resources (for example, pages in memory or connections through the network). The secure bindings are revoked and the resources may be allocated by other applications. If the ap-

Design Principle	Application in Exodisk System
Securely expose hardware.	Two-level model: On the first level, disk sectors and extents of contiguous disk sectors are associated with capabilities. On the second level, several such extents can be combined in a lightweight *i*-node, called an *exonode*, which can also contain arbitrary-length application-defined data.
Expose allocation.	All extents and exonodes need to be explicitly allocated and deallocated. Allocation establishes a *secure binding*, that is a mapping between extent or exonode and protection unit (capability). The exodisk system exports a free-list of disk storage representing the current status of allocation.
Expose Names.	The exodisk system uses physical sector numbers. It also exports the last-known disk position and the queue of pending disk operations.
Avoid policy decisions.	The exodisk system minimizes policy decisions and lets applications determine their own allocation, prefetch, access and cache policies. It uses an in-kernel cache for extent headers and exonodes, called the *exodisk cache*, in order to efficiently support its protection model. Caching and cache replacement should be application controlled.

Table 2.1: The Design Principles and Corresponding Components of the Exodisk System.

plication restarts, it consequently has to re-acquire these resources. In contrast, when using disk storage, an application typically expects to continue to have access to the on-disk data, even if it exits and restarts and even if the whole computer system shuts down and restarts. The secure bindings between on-disk data and protection information consequently need to be maintained persistently. Extent headers and exonodes thus have to be stored on disk as well. While this insight may not be very surprising, it has important implications on persistence, integrity, and security.

A new allocation does not become persistent until it is reflected on stable storage. Applications control when the secure bindings (i.e., the extent headers and exonodes) are written to disk storage and thus control when allocation becomes persistent. Analogously, applications also control when deallocated extent headers and exonodes are written to disk storage and thus determine when deallocation becomes persistent. Applications can thus implement their own persistence semantics which usually represent some trade-off between persistence guarantees and application performance (see 5.4.3).

Storage system integrity obviously requires that an allocated disk sector is either contained in exactly one self-protected extent or it has exactly one exonode pointing to it (proxy exonodes are excluded from this invariant). Conversely, no un-allocated disk sector can be part of a self-protected extent or can have any exonodes pointing to it. Also, all self-protected extents can only contain allocated sectors and all exonodes can only point to allocated sectors. The exodisk system implementation must enforce these invariants.

Storage system integrity also requires that the set of bindings and the free-list agree on which blocks are available for allocation and which are not. To avoid complex metadata dependencies in the exodisk system, the free-list provides a conservative approximation of the actual (persistent) state of allocation. Sectors are marked as allocated in the free-list during allocation (before the extent header or exonode is written to disk storage) and are only marked as free once a deallocation has become persistent (after the extent header or exonode has been written to disk storage). Differences are reconciled during system initialization. This mechanism represents a necessary policy decision in the exodisk system design that favors performance and simplicity but may have a negative impact on recovery time.

Since proxy exonodes are not persistent, several proxy exonodes can point to the same self-protected extent or exonode. However, proxy exonodes can only point to a set of sectors which is either the same as that of the originating extent/exonode or smaller than it. As a consequence of this invariant, only modifications to a proxy exonode that do not add new sectors or modify the application-defined data are allowed. Also, reductions to the originating exonode (or extent) must be reflected in any corresponding proxy exonodes.

Finally, because secure bindings (i.e., access control information) are kept on stable storage (and maintained for long periods of time), they represent a potential security hole, even though they are maintained within the kernel and applications can only access them through system calls. For example, an attacker could remove the disk drive from the computer system, connect it to a different system that does not used an exodisk system and then analyze and modify the on-disk data. The prototype exodisk system addresses this problem by encrypting capabilities in extent headers and exonodes as well as using encrypted "magic" numbers that uniquely identify extent headers and exonodes on disk. The encryption is dependent on the sector number of the extent header or exonode which ensures that the same capability or magic number results in different encrypted representations, depending on their on-disk locations. This scheme thus provides some level of security.

However, an attacker could still circumvent the exodisk system (by accessing the disk through a computer system that does not use the exodisk system, as described above) and modify a known extent or exonode (i.e., an extent or exonode to which it has both *read* and *write* capabilities) to also validate access to parts of the disk the extent or exonode did not originally include. This scheme could then be used to gain access to any part of the on-disk data, even through the exodisk system. Other implementations may thus choose to provide a higher degree of security and use digital signatures, encrypt all disk data, or, instead of using simple capabilities, use an authentication system such as Kerberos [32] for authentication and authorization. An implementation must thus make a trade-off between security of the on-disk data and the complexity of the authentication scheme.

2.3 Issues in Multiplexing Disk Storage

A number of issues other than access control arise when multiplexing disk storage among applications. Issues include the notification of completion of disk operations, the scheduling of pending exodisk system requests, an improved but simple exodisk cache design that gives applications *full* control over cache management, and the interaction between the exodisk system and buffer cache management.

2.3.1 Asynchronous I/O

For disk I/O, the exodisk system provides asynchronous *read* and *write* operations. That is, the exodisk system immediately returns control to the calling application, without waiting for the disk request to complete, as soon as a *read* or *write* operation has been handed to the disk driver. Applications may choose to wait for requests to complete or not. On completion of a *read* or *write* operation, the exodisk system needs to notify the requesting application that the operation has finished. Two solutions are interrupts, using a reserved disk interrupt that must be handled by applications, and polling, using an integer-size message through application memory. The prototype implementation of the exodisk system uses polling.

2.3.2 Disk Scheduling

Several disk accesses from each of several applications can be outstanding at any time. As a result, a mechanism for scheduling disk operations is needed. The prototype exodisk system is implemented on top of a disk driver that uses the cyclical scan algorithm (C-LOOK), which always schedules requests in ascending order and achieves very good performance when compared to other seek-reducing algorithms [53]. Since the queue of disk operations is visible to applications, they can use this information to determine when to initiate disk operations. Future implementations may replace this global default with mechanisms for allowing application-control over disk scheduling. For example, applications could group several disk operations (up to a certain maximum number of disk operations) that are guaranteed to be scheduled in the given order without other applications being able to schedule operations in between the operations in this group.

2.3.3 An Improved Exodisk Cache Design

The current design includes a global exodisk cache, shared by all applications, which maps addresses to extent headers or exonodes. Applications can read or write extent headers and exonodes, as well as override the standard LRU replacement policy by marking some other extent header or exonode as least recently used. This design poses a number of problems: First, a greedy application could use this application control to effectively use almost all entries in the exodisk cache for its own protection data. Second, while exonodes are expected to be small (smaller than a disk sector), some storage manager implementations (for example, a logical disk system) may use very large exonodes. Again, devoting most of the exodisk cache to the protection data used by one application could harm other applications. Third, applications have only limited control over the exodisk cache size and replacement policy, allowing for no guarantees about when extent headers or exonodes are written back or replaced.

A future implementation may thus choose to introduce a more sophisticated exodisk cache design: A mechanism similar to the LRU-SP policy suggested by Cao *et al.* in [3] can be used to fairly distribute exodisk cache entries among applications and to offer protection against greedy applications. Alternatively, a cost-benefit model in connection with application supplied hints as suggested by Patterson *et al.* in [39] can be used to regulate the caching and prefetching within the exodisk cache. While both mechanisms ensure better management of the shared cache and show good application performance, they clearly limit application control over the shared cache and impose complex in-kernel policies.

A simple design[1], that maximizes application control over the exodisk cache, could work as follows: Applications explicitly determine when to read extent headers and exonodes into

[1]This design results from discussion with Dawson Engler, Frans Kaashoek and Greg Ganger.

the cache, and when to write them back to disk. On a *read* operation, the application chooses an extent header or exonode already in the cache to be replaced by the extent header or exonode to be read in. The application can only select extent headers or exonodes that it *itself* has read in, but not extent headers or exonodes read in by other applications. If the application does not want to replace any of the extent headers or exonodes, it can alternatively "donate" physical memory to the exodisk cache, which will then be used to cache the extent header or exonode to be read in and is exclusively managed by the donating application. After a *write* operation, the application can either leave the extent header or exonode in the exodisk cache, or have the memory used by the extent header or exonode returned to it.

If, on a *read* operation, the extent header or exonode is already in the exodisk cache, but under control of another application, the exodisk system will not copy it (thus avoiding cache consistency problems) but rather use the other application's copy. The memory that would have been used to store the extent header or exonode is marked as reserved. If the other application wants to replace its copy of the extent header or exonode, the extent header or exonode is copied into the reserved memory and continues to remain within the exodisk cache. If this application wants to replace the extent header or exonode the reserved memory is simply reused. This scheme ensures that, if several applications concurrently cache the same extent header or exonode, each application has the same view of the extent header or exonode and it avoids consistency problems in the exodisk cache.

This improved design has not been implemented (due to time constraints) and is only proposed here.

2.3.4 Buffer Cache

Applications can cache disk blocks (to which they have access) at the application-level. While this approach gives applications complete control over the management of their private buffer cache, it has two major short-comings: First, cached blocks can not be easily shared among applications. Sharing blocks among applications requires that applications trust each other and implies some shared protocol to locate where a particular data block is cached. Second, the lifetime of a cached block is limited by the caching application. One possible solution would use a global, trusted, user-level cache server acting as an intermediary between applications and the disk system. However, such a solution severely limits the flexibility of applications. More importantly, this solution also requires that the cache server has access rights to *all* cached disk blocks and enforces the appropriate access protections. The cache server would thus duplicate the storage protection of the exodisk system.

A better solution would be a kernel-level buffer cache. Its design is proposed here, but is has not been implemented. All *read* and *write* access, even to data already in the buffer cache, would have to be through the exodisk system to enforce storage protection. The buffer cache could be managed in a similar way as suggested above for the exodisk cache: Applications explicitly determine when to read a data block into the cache, and when to write it back to disk. On a *read* operation, the application chooses a data block already in the cache to be replaced by the data block to be read in. Again, as in the suggested exodisk cache design, the application can only select a data block it *itself* has read in. If the application does not want to replace any data block, it can alternatively donate physical memory to the buffer cache, which will then be used to cache the data block to be read in.

If, on a *read* operation, the data block to be read in already is in the buffer cache, but under control of another application, the data is not copied and the application is notified that another application's copy is used. The data block is marked as also being used by this application. If the other application wants to replace its copy of the data, this application is notified by an interrupt that the data block is about to be replaced. This application can then choose to provide memory (by either donating physical memory or by replacing another data block it owns) for the data block to remain in the buffer cache, or to allow the data to be removed from the buffer cache. Since a buffer cache will typically cache more and larger data than the exodisk cache, this scheme avoids large areas of physical memory being reserved for future use in the cache. At the same time, through visible resource revocation [11], applications can still completely control their buffer cache management.

2.4 Limitations

The exodisk system *space*-multiplexes disk storage but does not provide a mechanism to *time*-multiplex the available disk bandwidth. It seems that, for all but a few applications (such as digital video), this is best left to the storage subsystem anyway. Disk storage in the exodisk system is allocated on a first-come-first-serve basis. The exodisk system does not enforce any (other) allocation policies or quotas and it does not provide a mechanism for storage revocation. Since allocation policies, quotas and revocation require high-level policy decisions, they are best implemented on top of the exodisk system. For example, a central exonode server (see 4.3) can be used to allocate all storage (using descendant exonodes) and enforce the high-level allocation and revocation policies (since it can access the allocated data through the use of descendant exonodes). Finally, the exodisk system can not determine which on-disk data is live and which data is not used anymore by any application. This may make it hard to reclaim all unused disk space.

2.5 Summary

The design of the exodisk system attempts to maximize application control over disk storage management. It is guided by four design principles: (1) To securely expose the disk hardware, (2) to expose the disk allocation, (3) to expose the physical name-space, and (4) to avoid resource policy decisions. The resulting design uses self-protected extents and exonodes as its basic units of protection, requires explicit allocation and deallocation, uses a free-list to indicate the current state of allocation, and lets applications control most exodisk cache operations. Furthermore, the exodisk system uses asynchronous I/O, and leaves the caching of disk blocks to applications.

Chapter 3

Implementation

The prototype implementation of the exodisk system is implemented as a component of the Aegis exokernel [11]. Since it is independent of the rest of the exokernel, it could easily be incorporated into other operating systems, with the same benefits. The prototype runs on DECstation 3100 and 5000 computers. It consists of approximately 4300 lines of well-commented C-code. The underlying disk drivers for both the DECstation 3100 and 5000 were derived from the NetBSD disk drivers (using the drivers available in May 1995). They feature several bug fixes over the NetBSD versions (including a performance bug that limited the available *read* throughput) and have been modified to use a C-LOOK scheduling algorithm and to support scatter-gather disk I/O. The prototype implementation does not yet support proxy exonodes or mapping exonodes into application-space. Furthermore, exonodes can be at most 512 bytes in size (the sector size). However, several exonodes of the same size can be packed into a single sector. The exodisk cache is statically sized, shared among all applications and caches entire sectors, i.e. the first sector of each self-protected extent (including the extent header) and the entire sector for exonodes (which may or may not contain several exonodes).

This chapter is structured as follows. Section 3.1 gives an overview of the interface used in the prototype implementation and defines the data types used in the interface. Section 3.2 gives an overview over how the different operations are implemented.

```
SYSCALL int exodisk_request( int op,
                             struct ed_protector *prot,
                             struct buf *reqbp,
                             struct ed_operation *stuff );
```

Table 3.1: The Exodisk Prototype Interface.

3.1 Prototype Interface

The prototype implementation of the exodisk system uses a single system call, called `exodisk_request`, that takes four parameters and returns an error code. Its specification is given in Table 3.1. `op` determines which operation is to be executed by the exodisk system. The operations currently supported by the prototype implementation are listed in Table 3.2. Whether the `prot`, `reqbp` and `stuff` pointers are actually dereferenced by the exodisk system depends on the individual operation. `prot` identifies the self-protected extent or exonode to be used in an operation. Its base type, `struct ed_protector`, is defined in Table 3.3. `reqbp` describes disk I/O operations, and its base type, `struct buf`, is defined in Table 3.4. `stuff` provides various arguments to some exodisk system operations. Its base type, `struct ed_operation`, is defined in Table 3.5.

Most operations in the exodisk system (besides `ED_OP_POLL`, `ED_OP_GET_FREE_MAP` and `ED_OP_SHUTDOWN`) require an extent header or exonode. For all these operations, the `prot` pointer must be supplied. Extent headers are identified by providing the disk unit (`unit`) and the start sector (`blkno`). Exonodes are identified by the same fields in addition to the index within the sector (`xn_num`—to allow several exonodes to be packed into a single sector). Capabilities (not necessarily both for all operations) are provided in the `rc` and `wc` fields. They are used for disk I/O (`ED_OP_IO`), for allocation and deallocation of extents and exonodes, and for all operations that modify a self-protected extent or exonode. Capabilities are not needed to read or write an extent header or exonode.

The length field (`length`) is only used during allocation. It defines the total length in bytes of the self-protected extent or of the exonode. It must be a multiple of the sector size for self-protected extents and a multiple of the word size (4 bytes—to guarantee the correct alignment of data) for exonodes. The exonode entry number (`xn_entry`) indicates which exonode entry to use when adding an extent, descendant exonode or application-defined data to an exonode (`ED_OP_EXONODE_ADD_EXTENT`, `ED_OP_EXONODE_ADD_EXONODE` and

```
enum {
    ED_OP_IO,                     /* Read/Write Data                           */
    ED_OP_EXTENT_ALLOCATE,        /* Allocate Self-Protected Extent            */
    ED_OP_EXTENT_DEALLOCATE,      /* Deallocate Self-Protected-Extent          */
    ED_OP_EXTENT_GROW,            /* Grow Self-Protected Extent                */
    ED_OP_EXTENT_SHRINK,          /* Shrink Self-Protected Extent              */
    ED_OP_EXTENT_READ,            /* Read Extent Header into Exodisk Cache     */
    ED_OP_EXTENT_WRITE,           /* Write Dirty Extent Header                 */
    ED_OP_EXONODE_ALLOCATE,       /* Allocate Exonode                          */
    ED_OP_EXONODE_DEALLOCATE,     /* Deallocate Exonode                        */
    ED_OP_EXONODE_READ,           /* Read Exonode into Exodisk Cache           */
    ED_OP_EXONODE_WRITE,          /* Write Dirty Exonode                       */
    ED_OP_EXONODE_ADD_EXTENT,     /* Allocate and Add Extent to Exonode        */
    ED_OP_EXONODE_ADD_EXONODE,    /* Allocate and Add Descendant Exonode       */
                                  /*    to Exonode                             */
    ED_OP_EXONODE_ADD_DATA,       /* Add or Replace Application-Defined Data */
                                  /*    to/in Exonode                          */
    ED_OP_EXONODE_READ_DATA,      /* Read Data from Exonode                    */
    ED_OP_EXONODE_REMOVE_ENTRY,   /* Remove Entry from Exonode                 */
    ED_OP_EXONODE_COMBINE,        /* Combine Entries (Sector or Extent)        */
                                  /*    in Exonode                             */
    ED_OP_EXONODE_SPLIT,          /* Split Entry (Extent) in Exonode           */
    ED_OP_EXONODE_MAP,            /* Map Exonode (Not Yet Supported)           */
    ED_OP_EXONODE_PROXY,          /* Make Proxy Exonode (Not Yet Supported)    */
    ED_OP_EXONODE_CAP_DISCOVER,   /* Discover capabilities                     */
                                  /*    of Descendant Exonode                  */
    ED_OP_MARK_REPLACEMENT,       /* Mark Extent Header or Exonode             */
                                  /*    for Replacement                        */
    ED_OP_POLL,                   /* Number of Pending Operations              */
    ED_OP_IN_CACHE,               /* Extent Header or Exonode In Cache?        */
    ED_OP_GET_FREE_MAP,           /* Get Physical Address of Free-Map          */
    ED_OP_SHUTDOWN                /* Shut Down Exodisk                         */
};
```

Table 3.2: The Exodisk Prototype Operations.

```
struct ed_protector {
    uint16         unit;          /* Disk unit                                 */
    uint16         xn_num;        /* Exonode index                             */
    daddr_t        blkno;         /* Block number                              */
    uint32         length;        /* Length                                    */
    int32          xn_entry;      /* Entry in exonode, -1 if invalid           */
    dauth_t        rc;            /* Read capability                           */
    dauth_t        wc;            /* Write capability                          */
};
```

Table 3.3: The struct ed_protector Data Structure.

```
struct buf {
    struct  buf *b_next;        /* Used internally.                        */
    volatile long   b_flags;    /* Operation: B_READ or B_WRITE.           */
    dev_t   b_dev;              /* Disk unit.                              */
    daddr_t b_blkno;            /* Block number.                           */
    int     b_bcount;           /* Valid bytes in buffer.                  */
    caddr_t b_memaddr;          /* Corresponding main memory location.     */
    int     *b_resptr;          /* Desired location of result.             */
    int     b_resid;            /* Used internally.                        */
};
```

Table 3.4: The `struct buf` Data Structure.

```
struct ed_operation {
    caddr_t        buf;         /* Application-defined data.               */
    uint32         one;         /* First parameter.                        */
    uint32         two;         /* Second parameter.                       */
    uint32         three;       /* Third parameter.                        */
    dauth_t        rc;          /* Read capability of Descendant Exonode.  */
    dauth_t        wc;          /* Write capability of Descendant Exonode. */
};
```

Table 3.5: The `struct ed_operation` Data Structure.

`ED_OP_EXONODE_ADD_DATA`). If it is `-1`, the exodisk system picks the first available entry. The exonode entry number must also be supplied for operations that access a specific entry of an exonode (`ED_OP_EXONODE_READ_DATA`, `ED_OP_EXONODE_REMOVE_ENTRY`, `ED_OP_EXONODE_COMBINE`, `ED_OP_EXONODE_SPLIT` and `ED_OP_CAP_DISCOVER`). It may also be used for disk I/O, where it serves as a hint as to which entry in the exonode actually covers the sectors to be accessed.

The `buf` pointer is only used for disk I/O (`ED_OP_IO`). The flag (`b_flags`) determines whether to read from disk or to write to disk. Disk I/O also requires the disk unit (`b_dev`), the start sector on disk (`b_blkno`), the length of the data to be accessed (`b_bcount`—which must be a multiple of the sector size) and the main memory address of the application buffer (`b_memaddr`). If the result pointer (`b_resptr`) does not equal `NULL`, the exodisk system will write `0` (or a negative error code) into the location pointed to by `b_resptr` on completion of the disk request. This allows applications to implement polling. The `buf` pointer can be overloaded with a pointer to an integer (for the same polling purposes) when reading and writing extent headers and exonodes.

Various operations need additional information which is supplied through the stuff pointer. For growing and shrinking self-protected extents (ED_OP_EXTENT_GROW and ED_OP_EXTENT_SHRINK), the length (in bytes) by which the extent is to be modified is contained in the first field (one). The length needs to be a multiple of the sector size. For adding an extent to an exonode (ED_OP_EXONODE_ADD_EXTENT), the start sector is contained in the first field (one) and the length (in bytes) of the extent in the second field (two). The length, again, needs to be a multiple of the sector size. For adding a descendant exonode to an exonode (ED_OP_EXONODE_ADD_EXONODE—which also allocates the descendant exonode), the first field (one) defines the sector number of the descendant exonode, the second field (two) defines the index of the exonode within the sector (as the xn_num field does in struct ed_protector), the third field (three) defines the length (in bytes) of the descendant exonode, and the capability fields (rc and wc) define the capabilities of the descendant exonode. The length of descendant exonodes, as for exonodes, needs to be a multiple of the word size (4 bytes).

For writing and reading application-defined data (ED_OP_EXONODE_ADD_DATA and ED_OP_EXONODE_READ_DATA), the first field (one) specifies the length (in bytes) of the application-defined data. The length must be a multiple of the word size (4 bytes—to guarantee the correct alignment of data in the exonode). The pointer (buf) points to the main memory location of the data buffer. For combining two entries in an exonode (ED_OP_EXONODE_COMBINE), the first field (one) specifies the second entry to be combined (the first entry is specified by the xn_entry field of prot). For splitting one extent entry in an exonode (ED_OP_EXONODE_SPLIT), the first field (one) specifies where to split the extent (as an offset in bytes from the beginning of the extent) and the second field (two) specifies the entry number for the split-off part of the original extent. The offset needs to be a multiple of the sector size. Finally, for discovering the capabilities of a descendant exonode (ED_OP_EXONODE_CAP_DISCOVER), the capabilities of the descendant exonode are written into the rc and wc fields.

3.2 Prototype Operation

After having been invoked through the exodisk_request system call, the exodisk system first verifies that any application-supplied data (the data structures pointed to by the prot, reqbp and stuff pointers, as well as buffer space for disk I/O and for application-defined data) have virtual-to-physical mappings in the processor's TLB (or the exokernel STLB [11]). If this is not the case, a TLB fault is signaled to the application, and the system call is restarted after the mapping has been installed in the TLB.

If the requested operation does not affect self-protected extents or exonodes (i.e., it is `ED_OP_POLL`, `ED_OP_GET_FREE_MAP` or `ED_OP_SHUTDOWN`) it is simply executed. Otherwise, the exodisk system dispatches control based on the individual exodisk operations:

- If the operation allocates a new self-protected extent or a new exonode, the exodisk system first verifies if the requested sector(s) are valid and not already allocated. If this is the case, it creates a new extent header or exonode in the exodisk cache, marks the exodisk cache entry as dirty and the requested sector(s) as allocated. On allocation of an exonode in a sector with several exonodes, if the sector is allocated but in the exodisk cache (and this particular exonode is not used in the sector), the exodisk system simply adds the exonode to the sector. In all other cases, an appropriate error code is returned.

- If the operation reads a sector with exonodes or an extent header into the exodisk cache, the exodisk system verifies that the sector is allocated and, if so, issues a disk request. Control is then returned to the application. On completion of the disk request, the exodisk system verifies that the sector indeed contains an extent header or exonode (using the magic number) and adds it to the exodisk cache. If the sector does not contain an extent header or exonode, it is simply discarded from the exodisk cache. If a result pointer was supplied, the exodisk system will post a result code on completion of the disk request.

- If the operation modifies a self-protected extent or exonode (including deallocating the extent or exonode), the exodisk system verifies that the corresponding extent header or exonode is in the exodisk cache (and not currently being written to disk), that the application presented the correct *write* capability for the modification, and other conditions necessary to complete the modification (for example, whether the disk space is free when adding disk space to a self-protected extent or exonode). The exodisk system then executes the modification and marks the exodisk cache entry as dirty.

  If the operation deallocates disk space, and the extent header or exonode has not been written to disk since allocation, the exodisk system immediately releases the space in the free-map (and the exodisk cache entry if the entire self-protected extent or all exonodes in a sector are deallocated). If the extent header or exonode has been written to disk since allocation, the exodisk system notes this in the exodisk cache entry and the space will be released in the free-map once the extent header or exonode has been written to disk (if the entire self-protected extent or all exonodes in a sector are deallocated, the exodisk cache entry is marked as deallocated but not yet written back to disk).

- If the operation reads data from disk or writes data to disk, the exodisk system verifies that the corresponding extent header or exonode is in the exodisk cache, that the application presented the correct capability, and that the self-protected extent or exonode actually covers the disk space to be accessed. If this is the case, the exodisk system issues the disk request and returns control to the application. If a *write* request includes the first sector of a self-protected extent, the exodisk system copies the data (except the space taken by the extent header) of the first sector into the exodisk cache and sets up a scatter-gather disk request. On completion of the disk request, the exodisk system posts a result code to the application (if a result pointer was supplied).

- If the operation writes an extent header or exonode to disk, the exodisk system verifies that the corresponding extent header or exonode is in the exodisk cache (and not currently being written to disk), and the extent header or exonode is in fact dirty. It then issues the disk request and returns control to the application. On completion of the disk request, the exodisk system marks sectors as free in the free-map (if appropriate), marks the extent header or exonode as clean and posts a result code to the application (if a result pointer was supplied).

Chapter 4

Exodisk System Usage

This chapter describes how storage managers can be implemented on top of the exodisk system, how applications can share on-disk data, and how unused disk space can be reclaimed. The discussion is intended to highlight the flexibility of the exodisk system. The storage managers discussed in this chapter have not been implemented. Furthermore, the storage management policies suggested in this chapter are not enforced by the exodisk system. Their effectiveness largely depends on how closely applications follow an agreed-on convention. Also, other, possibly more effective storage management policies can be implemented as well.

This chapter is structured as follows. Section 4.1 gives examples of storage managers on top of the exodisk system. Section 4.2 discusses mechanisms for applications to share on-disk data. Section 4.3 proposes policies that facilitate reclaiming unused disk space.

4.1 Example Storage Managers

This section gives examples of how applications might utilize the exodisk system interface to implement their own storage managers. It focuses mostly on conventional storage managers and possible extensions to them. The on-disk images of file systems implemented above the exodisk system will generally differ from their original on-disk images above raw disk storage.

An Unix file system implementation on top of the exodisk system can use exonodes as *i*-nodes. The file metadata (such as the file size and the access rights) of an Unix *i*-node can be stored as application-defined data inside the exonode. Direct blocks are extent entries of

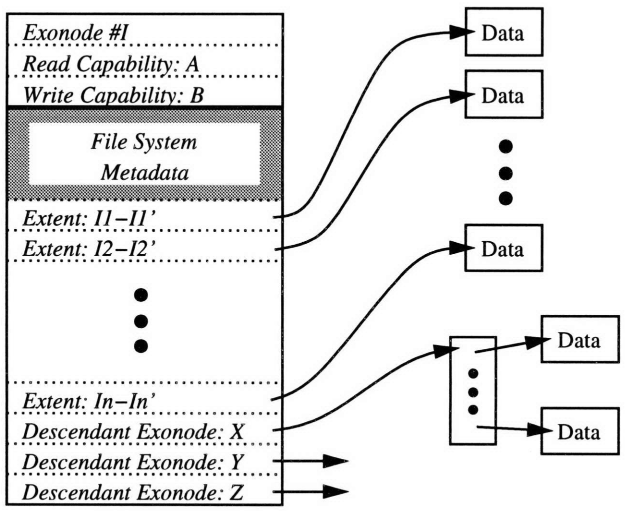

Figure 4-1: An Exonode as an Unix File System I-node. Exonode I with *read* capability A and *write* capability B serves as an *i*-node. The file size, access rights and other file system metadata are stored in the exonode as application-defined data. They are followed by a list of n extent entries that represent the direct blocks. Finally, pointers to three descendant exonodes X, Y and Z represent the single, double and triple indirect block pointers.

a fixed size (e.g., 4 KByte). The single, double and triple indirect entries in an *i*-node are descendant exonode entries, with the descendant exonodes using the same capabilities as the root exonode and consisting of fixed-size extent entries and further descendant exonode entries (and the exodisk system metadata). Figure 4-1 illustrates this use of an exonode to support Unix File System *i*-nodes. Since directories are simply special files based on the same *i*-node representation, they do not require special mechanisms.

The in-core *i*-node table of an Unix file system can be maintained within the exonode cache of the exodisk system. Since applications have tight control over caching and replacement of the exonode cache (using the exodisk cache design proposed in Section 2.3.3), an user-level Unix file system can dynamically regulate its caching and prefetching policies depending on the specific workloads. Furthermore, applications can define their own persistence models and implement them on a per-file basis (see Section 5.4.3).

Several extensions to the conventional Unix file system structure can be easily implemented on top of the exodisk system. The data of very small files can be stored within the exonode, thus avoiding one disk operation for both *read* and *write* access to such a file [33]. The use of extents in the exodisk system makes it possible to allocate file blocks in larger contiguous units than single blocks, resulting in a (partial) extent-based file system structure. Such a modification will result in improved performance because more data can

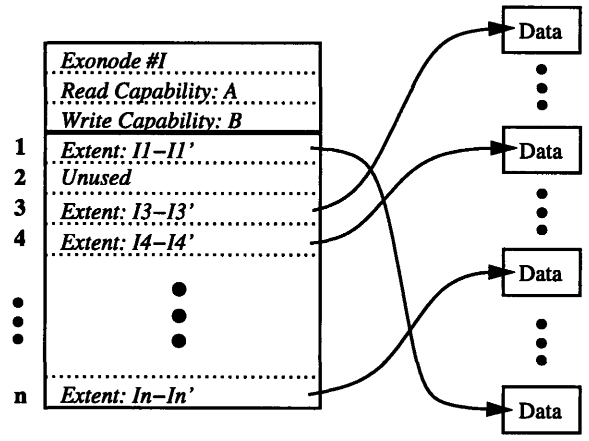

Figure 4-2: Logical-To-Physical Mapping in an Exonode. Exonode I is protected by the *read* capability A and the *write* capability B. It has n entries to represent the n logical blocks. Each entry either contains an extent to point to the corresponding physical data block, or is unused to indicate that the logical block is not currently allocated (entry 2).

be referred to by direct pointers and larger data transfer sizes can be used [40, 31]. Since the size of exonodes is not fixed, it is possible to use *i*-nodes of different sizes and avoid the use of indirect *i*-nodes for medium-sized files. Since applications have tight control over the bottom layer of metadata, they can implement new file system operations (such as file insertion, see Section 5.4.1) that are hard to implement in conventional file systems (i.e., only with a considerable performance hit). Finally, applications can create user-augmented file systems by adding new functionality (such as encryption and compression) on top of given storage layouts by expanding the storage manager (similarly to stackable file systems [21]).

The MS-DOS file system does not use *i*-nodes to organize disk storage but rather uses a so-called *File Allocation Table* (FAT) to map disk blocks to successors [51]. The resulting singly-linked lists indicate the blocks and the order of blocks belonging to a particular file while the file itself is simply represented by the initial index into the FAT. An MS-DOS file system implementation on top of the exodisk system can use one large self-protected extent for the FAT and several, fixed-size self-protected extents to represent data blocks (causing additional overhead because each block, or self-protected extent, has its own protection information). Alternatively, it can use one exonode, storing the FAT in the exonode as application-defined data and using fixed-size extent entries for the individual blocks.

Logical representations of disk storage such as Mime [7] or the Logical Disk [8] map logical block numbers to physical disk locations. Such a mapping can be efficiently implemented using an exonode: The logical block number can be an index into the array of exonode entries and the physical block information (again, a fixed-size extent) is stored in that exonode entry. Exonode entries that represent logical blocks with no mapping into disk storage are simply not used. Figure 4-2 illustrates the use of an exonode to represent the logical-to-physical mapping of a logical disk system. This implementation technique has little protection overhead (every extent is protected by the capabilities of the exonode) and allows fine-grained interleaving with other file systems. Descendant exonodes can be used to divide the logical name-space into several smaller units and to thus avoid having an overly large exonode.

Log-structured file systems such as Sprite LFS [44] and BSD LFS [46] can use an exonode with several fixed-size extent entries, where each extent represents a segment of the file system log. While generally all file data is accessed through the log-structured file system interface, an application can use proxy exonodes to give other applications *read* access to individual files without these applications being aware of the log-like organization of file data. The proxy exonode for a file would consist of several (ordered) extent entries that represent that particular file's data.

Temporary disk storage can be allocated very efficiently in the exodisk system by simply using self-protected extents. Applications can thus directly use the exodisk system interface to read and write their temporary data (without incurring the overhead associated with an additional file system layer) and implement, for example, their own paging system.

Database managers usually store their records or objects in large files and utilize separate indices (often implemented as a b-tree) to efficiently access particular records or objects within a data file. The database indices, the file system metadata, and the exodisk system protection information all represent a mapping from disk storage to metadata, each associated with different layers of storage management. However, maintaining these mappings in separate disk locations may have a large impact on database performance [50]. The exodisk system presents an elegant solution to this problem, since database managers can use application-defined data entries in exonodes to store the database index together with the file system and exodisk system metadata.

All of the above examples avoid dependencies on fixed partitions. Several file systems can thus be finely interleaved, and a particular application can use several file systems at the same time without incurring the overhead associated with seeks from partition to partition (see Section 5.3). For example, an application can store temporary data (organized in self-protected extents) in proximity to its source data (organized using a Unix file system) and thus avoid both file system overhead for access to temporary data and overhead for seeks between the regular Unix file system partition and a scratch partition.

4.2 Data Sharing

In order to share on-disk data, applications need to agree on a common storage format (for example, what part of a file contains what data). This is true for conventional storage managers such as file systems as well as for the exodisk system. However, since the exodisk system gives applications tight control over the bottom layer of metadata (which is usually hidden from applications through the high level of abstraction provided by the storage manager), applications have a much larger degree of flexibility in choosing which format to use. A common storage format must thus include a description of this metadata as well (for example, the exact structure of the exonodes used to describe files).

4.2.1 Simple Sharing

The *read* and *write* capabilities associated with self-protected extents and exonodes provide a basic mechanism for sharing on-disk data between applications. An application that creates data can give either capability or both to another application that can then access the same on-disk data. This scheme faces two serious problems: First, since secure bindings remain persistent for the lifetime of the on-disk data, a transfer of access rights (by giving out a capability) authorizes the recipient to access the data for its lifetime. Second, since capabilities grant access to *all* data protected by a self-protected extent or exonode, a transfer of access rights always authorizes the recipient to access all data protected by a self-protected extent or exonode. Both problems can be addressed by using proxy exonodes.

4.2.2 Sharing Using Proxy Exonodes

Proxy exonodes provide a convenient means to allow for the *temporary* sharing of *partial* self-protected extents or exonodes. An application that creates data can create a proxy exonode with different capabilities from the original self-protected extent or exonode to authorize access to part of the on-disk data. It can then give the proxy exonode identifier and the capabilities of the proxy exonode to another application. The recipient will only have access to the part of the original self-protected extent or exonode described by the proxy exonode and only for the lifetime of the proxy exonode (which is always limited by system reboots). Once the proxy exonode has been removed from the exodisk cache, the recipient can not access the data protected by the proxy exonode anymore and has to re-request access from the application that created the data.

4.3 Reclaiming Disk Space

Disk storage raises the problem of how to reclaim disk space that is properly allocated but not used anymore (for example, because the data is outdated or because the user that stored this data on disk does not use the computer system anymore). The exact policy of how to determine what data will not be used anymore and when to reclaim the unused but allocated disk space is usually determined by some human agency and enforced by a specially privileged person or superuser (such as a system administrator). This problem also occurs in traditional storage managers. However, it is exacerbated in the exodisk system since each application can utilize its own storage manager (with a different on-disk data layout) and since any application that wants to deallocate an extent or exonode would need to know the correct *write* capability.

Providing a special mechanism for a superuser to access all self-protected extents or exonodes would fix a specific policy within the exodisk system and should thus be avoided. Since it would have to rely on some form of authorization that overrides the individual capabilities, it could also introduce a potential loophole in the protection scheme of the exodisk system. A hierarchical protection space has the potential of providing a (partial) solution that avoids the problems of a special mechanism within the exodisk system.

4.3.1 Exonode Hierarchy

Hierarchical protection spaces[1] can be built using a trusted server to allocate (some) exonodes. However, applications only use the server to allocate at most one exonode and thus interaction with this server is minimized. In this scheme, an application requests only its first exonode from a central exonode server (if several applications share the same library storage manager, the manager may only allocate one exonode for all these applications

[1]The idea of hierarchical protection spaces was originally proposed by David Mazieres. Marc Fiuczynski realized the full potential of the descendant exonodes used to build hierarchical protection spaces (the original exodisk system design only proposed indirect exonodes, protected by the same capabilities as the ancestor).

through the server). This first exonode is a descendant exonode, referenced from within a "root" exonode to which only the exonode server itself has access. The application agrees to allocate all data within descendant exonodes that are (through any number of references to descendant exonodes) referencable from the first exonode. Since all data is allocated within descendant exonodes with a common root and thus forms a hierarchical protection space, a properly authorized superuser can access the data of all applications through the root exonode.

In the current implementation of the exodisk system, applications can not be forced to allocate all their data in descendant exonodes. However, by changing the interfaces to require special authorization for allocating exonodes (as opposed to descendant exonodes), the current implementation can be easily changed to enforce hierarchical protection spaces while only slightly limiting the flexibility of applications.

Descendant exonodes provide hierarchical protection spaces through a level of indirection since a capability may grant access to a descendant exonode *through* an ancestor exonode. An alternative solution, which avoids the level of indirection, places the capabilities *themselves* into a hierarchical space. David Mazieres, who proposed this idea, is currently exploring it in an implementation of an exokernel operating system on the 80x86 architecture [28].

4.4 Summary

The examples in this chapter illustrate that the exodisk system provides applications with the flexibility to conveniently implement many different storage abstractions and optimization that best meet their specific performance and functionality needs. Several different storage abstractions can be finely interleaved on the same disk without requiring partitions to be set up in advance. The sharing of on-disk data is simplified through the use of proxy exonodes, which support both temporary and partial sharing. Descendant exonodes can be used to support hierarchical protection spaces. Note that none of the examples and mechanisms discussed in this chapter are enforced by the exodisk system which gives application designers the freedom to create and implement novel storage abstractions.

Chapter 5

Evaluation

This chapter shows that the overhead of protection in the exodisk system is minimal (less than 10%, and often in the noise) for most operations and storage organizations. It shows that the performance of concurrently running applications can benefit from fine-grained interleaving (by up to 45% when compared to allocating the data of two applications 250 MByte apart). Furthermore, it shows that a fast *insert* operation performs up to a factor of six better than insertion by copying (for large files), that disk-directed I/O [26] can improve the request-processing rate in an issue-wait-process cycle (by 39% when issuing 128 requests at the same time), and that applications can make their own trade-offs between persistence and performance by providing their own persistence models.

This chapter is organized as follows. Section 5.1 describes the experimental environment used in my experiments. Section 5.2 describes the experiments used to determine the base overhead of storage protection. Section 5.3 describes the experiments used to determine the benefits of fine-grained interleaving. Section 5.4 describes some experiments intended to illustrate the benefits of application-specific storage management. Section 5.5 summarizes the evaluation of the exodisk system.

5.1 Experimental Environment

The experiments described in this Chapter were conducted on a DECstation 5000/133 using a RZ25 disk drive. The characteristics of the computer system and the disk drive shown in Table 5.1 are taken from Digital Equipment's specifications [13, 14]. The exodisk cache for experiments using the exodisk system contains up to 281 entries, i.e. up to 281 extent

DECstation 5000/133: 33 MHz MIPS R3000A. 64 KByte I-cache, 128 KByte D-cache. 48 MByte main memory. 26.5 SPECmark89.
RZ25: 426 MByte formatted, 3.5-inch, SCSI-2. 4412 ± 0.5% RPM. 512 byte sectors. 60 KByte buffer. 14 ms average seek time. 20.8 ms average access time. 2110 spiral KByte per second maximum bandwidth. 42 random I/O per second maximum request rate.

Table 5.1: Experimental Environment.

headers or up to 281 sectors with exonodes. The size of the exodisk cache has been chosen somewhat arbitrarily: It had to be a prime number (to produce good results for the hash function used to hash into the cache). It was also chosen so that the working set of some experiments would fit into the cache, while it would not fit for all experiments. Since the cache size is a compile time constant, it can be easily changed. Cache replacement for the exodisk cache is least recently used (LRU). Experiments using the Ultrix file system were conducted in single user mode under Ultrix version 4.3, revision 44.

5.2 Base Overhead of Protection

This section describes a set of micro-benchmarks used to determine the base overhead of storage protection for both small "file" I/O and large "file" I/O. The performance of the same benchmarks using self-protected extents in the exodisk system, exonodes in the exodisk system, the Aegis disk driver, the Ultrix raw disk interface, and the Ultrix file system is compared. For convenience, the term "file" is used throughout this discussion, independent of how data is organized. The term, "file," thus generally denotes a collection of data and its associated exodisk system or Ultrix file system metadata. For experiments that use the Aegis disk driver or the Ultrix raw disk interface, no on-disk metadata is associated with the file data.

This section is structured as follows. Subsection 5.2.1 describes the small and large

Experiments	Operations
Small File I/O	Either 1,000 10 KByte or 10,000 1 KByte files: 1. Allocate and Write 2. Read 3. Deallocate
Large File I/O	One 80 MByte file: 1. Allocate 2. Sequentially Write 3. Sequentially Read 4. Randomly Write 5. Randomly Read 6. Deallocate

Table 5.2: The Small File I/O and Large File I/O Experiments.

file I/O experiments. Subsection 5.2.2 discusses the various file organizations and their disk layout (if known). Subsection 5.2.3 presents the experimental results for small file I/O. Subsection 5.2.4 presents the experimental results for large file I/O. Subsection 5.2.5 presents the conclusions for the the base overhead of storage protection.

5.2.1 Experiments

Each small file I/O experiment first allocates and writes, then reads and finally deallocates either 1,000 10 KByte files or 10,000 1 KByte files. Each file is read or written in one request. The large file I/O experiments first allocate, then sequentially write, then sequentially read, then randomly write, then randomly read, and finally deallocate one 80 MByte file. For the sequential *write* and *read* operations file data is transfered in 64 KByte blocks and for the random *write* and *read* operations file data is transfered in 4 KByte blocks. The large transfer size for the sequential *write* and *read* operations mimics the block clustering used in some file systems [31]; the transfer size for the random *write* and *read* operations reflects the typical block size in Unix file systems [29]. The experiments are summarized in Table 5.2.

The above experiments are modeled after similar experiments used to evaluate the performance of the Sprite log-structured file system [44] and the Logical Disk [8]. The file sizes of 1 KByte and 10 KByte are also consistent with [1, 9, 23, 37] which show that roughly between 70% and 80% of all files in the measured Unix, Sprite and AFS file systems are less than 10 KByte in size.

5.2.2 File Organizations

The following file organizations and interfaces are used in the small and large file I/O experiments. The labels in front of each file organization are used in Subsection 5.2.3 and Subsection 5.2.4 to represent the different organizations.

Extent. A file is represented as a self-protected extent. The file length is the length of all data, including the extent header. All files are allocated in one contiguous area of the disk drive, except that for every ten files in small file I/O one sector is skipped. This layout allocates file data for small file I/O in the same sectors as for the file organization using exonodes and thus ensures that the performance results are directly comparable (i.e., they just measure the overhead of storage protection). This organization measures the overhead of storage protection using self-protected extents.

Exonode. A file is represented as an exonode. Each exonode consists of a pointer to an extent that contains the file data. For small file I/O, ten exonodes are stored in one disk sector, followed by the data of the ten files. All files and their exonodes are allocated in one contiguous area of the disk drive. This organization measures the overhead of storage protection using exonodes.

Disk Driver. File data is written and read directly through the Aegis disk driver interface. File data is allocated in one contiguous area of the disk drive, except that for every ten files in small file I/O one sector is skipped. Again, this layout allocates file data for small file I/O in the same sectors as for the file organization using exonodes. This organization measures the basic performance of the disk in combination with the disk driver, providing a basis for evaluation of the other organizations.

Raw Ultrix. File data is written and read through the Ultrix raw disk interface. File data is allocated in one contiguous area of the disk drive, except that for every ten files in small file I/O one sector is skipped. Again, this layout allocates file data for small file I/O in the same pattern as for the file organization using exonodes. This organization measures the basic performance of the disk and Ultrix disk drivers (with some overhead due to the raw disk interface), providing a basis for comparing the Extent, Exonode and Disk Drivers organizations with the Ultrix file system (see below).

Ultrix. Files are accessed through the Ultrix `open`, `write`, `read`, `lseek` and `close` interfaces. For small file I/O, all files are equally distributed over 100 directories to reduce the file system overhead associated with directory operations (such as lookup). The directories are created as part of file allocation and are deleted as part of file deallocation. No control over the file layout on disk is possible. This organization measures the performance of a fully featured Unix file system.

The on-disk layout for the small file I/O experiments using the Extent, Exonode, Disk Driver and Raw Ultrix organizations uses the same sectors to store file data to ensure that differences in performance are a result of data organization and storage management, and not a result of artifacts of the disk system (such as remapped sectors). The differences in performance between the Extent, Exonode and Disk Driver organizations can thus be used to determine the overhead of storage protection in the exodisk system. The one sector every 10 files (which is skipped in the Extent, Disk Driver and Raw Ultrix organizations) reduces the available bandwidth by less than 5% for 1 KByte files and 0.5% for 10 KByte files.

5.2.3 Small File I/O

The results of the small file I/O experiments using 1,000 10 KByte files are shown in Figure 5-1. The numerical results are listed in Appendix A. In general, the Extent, Exonode, and Disk Driver organizations show comparable performance. The Exonode organization shows a small overhead for allocating and writing the 1,000 files, and the Extent organization shows a significant overhead for reading the 1,000 files.

Small file I/O using exonodes shows little overhead when compared to small file I/O through the Aegis disk driver. The 7.7% overhead for allocating and writing 1,000 10 KByte files is caused by the fact that the exodisk system needs to issue one extra disk request for every ten files (to write a sector with ten exonodes). For reading the 1,000 10 KByte files, no extra disk requests are required since all exonodes fit in the exodisk cache (1,000 exonodes organized as ten exonodes per sector use 100 exodisk cache entries). Also, file deallocation using exonodes is efficient (1.4 milliseconds per file).

Small file I/O using self-protected extents shows negligible overhead for allocation and writing when compared to the Aegis disk driver, because the exodisk system uses scatter-gather I/O to write the extent header and the user-data in one single disk request. So, for allocation and writing of self-protected extents, the exodisk system issues the same number

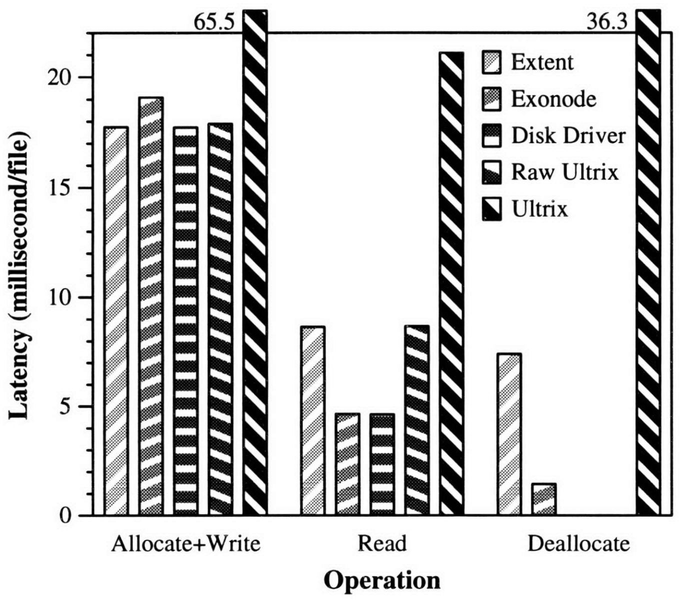

Figure 5-1: Small File I/O Performance: 1,000 10 KByte Files. Bars represent average latency per file in milliseconds. Smaller bars represent better results. Deallocate for Disk Driver and for Raw Ultrix shows no results, since nothing needs to be deallocated.

of disk requests as the raw disk driver. Reading shows considerable overhead (86.6%) for small file I/O using self-protected extents. Since each extent header takes up one exodisk cache entry, the working set of the experiment does not fit into the exodisk cache. The exodisk system thus needs to synchronously read the extent header into the exodisk cache, and only then can it read the rest of the data. Extent deallocation takes 5.1 times the time needed for deallocation using exonodes. Again, the need to synchronously read the extent headers into the exodisk cache and then write the extent headers of the deallocated files back to disk causes considerable overhead.

Scatter-gather I/O should be able to eliminate the overhead for reading extents by reading the extent header into the exodisk cache and the rest of the extent into application memory in one disk request (similarly to writing the extent header and the user-data in one disk request). Since the exodisk system needs to verify the *read* capability, the disk request has to be interrupted after reading the extent header into the exodisk cache (which requires modification of the disk driver). If the capabilities match, the request can complete as scheduled, but if they do not match, it is necessary to redirect the user-data into an in-kernel scratch buffer (or to terminate the disk request if supported by the disk driver). The size of this scratch buffer may fix a policy decision within the exodisk system since it limits the maximum transfer size for reading from self-protected extents (if the extent header is not already in the exodisk cache). Furthermore, the user-data in the first sector of the extent needs to be copied into application memory as well.

Small file I/O using the Ultrix raw disk interface shows negligible overhead for allocation and writing when compared to small file I/O through the Aegis disk driver. This result suggests that the Ultrix disk driver shows approximately the same performance as the Aegis disk driver (which is based on the NetBSD disk driver). Reading incurs considerable overhead (87.1%), which could be due to a performance bug in the Ultrix disk drivers (the original NetBSD disk driver did unnecessary *copy* operations on *reads* and thus showed decreased throughput, which was fixed in the Aegis disk driver) or, alternatively, in the way Ultrix implements the raw disk interface. Deallocation takes no time when using the Ultrix raw disk interface as well as when using the Aegis disk driver, since nothing needs to be deallocated (i.e., since both organizations do not use any metadata, no metadata needs to be written to disk).

Small file I/O using the Ultrix file system is 3.7 times slower than the Aegis disk driver for writing 1,000 10 KByte files and 4.5 times slower for reading 1,000 10 KByte files. Deallocation is 4.9 times slower than the experiments using self-protected extents in the exodisk system (which show the higher overhead in the exodisk system). These results show the high overhead of a full-blown file system.

The results of the small file I/O experiments using 10,000 1 KByte files are shown in Figure 5-2. The numerical results are listed in Appendix A. In general, the Extent and Exonode organizations show little overhead when compared to the Disk Driver organization. The Extent organization shows a significant overhead for reading the 10,000 files.

Small file I/O using exonodes, again, shows little overhead (less than 10%) when compared to small file I/O through the Aegis disk driver. This overhead includes synchronous *reads* to read the sector containing the exonodes for 10 files into the exodisk cache. However, since this *read* is only necessary for every 10 file accesses, the overhead is only slightly larger than for 1,000 files using exonodes. Again, file deallocation using exonodes is efficient (1.4 milliseconds per file).

Small file I/O using self-protected extents, again, shows negligible overhead for allocating and writing the 10,000 files when compared with the Aegis disk driver. Reading shows considerable overhead (68.1%) for small file I/O using self-protected extents. As mentioned earlier, this cost can be eliminated with a clever scatter-gather I/O approach. Deallocation

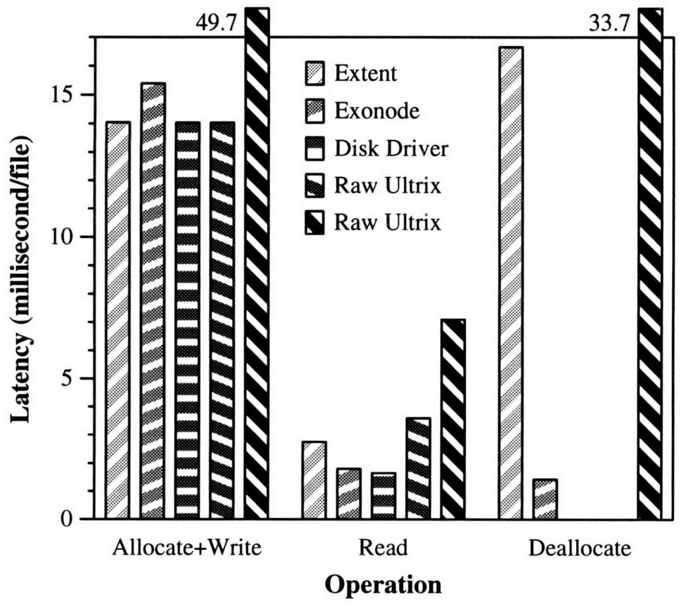

Figure 5-2: Small File I/O Performance: 10,000 1 KByte Files. Bars represent average latency per file in milliseconds. Smaller bars represent better results. Deallocate for Disk Driver and for Raw Ultrix shows no results, since nothing needs to be deallocated.

takes 11.7 times the time needed for deallocation using exonodes. As before the need to synchronously read the extent headers into the exodisk cache before each access causes considerable overhead.

Small file I/O using the Ultrix raw disk interface, again, shows negligible overhead for allocation and writing when compared to small file I/O through the Aegis disk drivers. Reading shows more than twice (2.2 times) the latency of the Aegis disk driver, which, again, indicates a performance bug in the Ultrix disk driver. As before, deallocation takes no time when using the Ultrix raw disk interface (or the Aegis disk driver).

Small file I/O using the Ultrix file system is between 3.5 and 4.3 times slower than the Aegis disk driver for allocating and writing and reading 10,000 1 KByte files, and two times slower than the experiments using self-protected extents in the exodisk system for deallocation. These experiments further underline the high overhead of a full-blown file system.

5.2.4 Large File I/O

The results for writing and reading one 80 MByte file in the large file I/O experiments are shown in Figure 5-3. The numerical results are listed in Appendix A. The results for file allocating and deallocation are not shown. They are minimal for allocation (less than 0.06

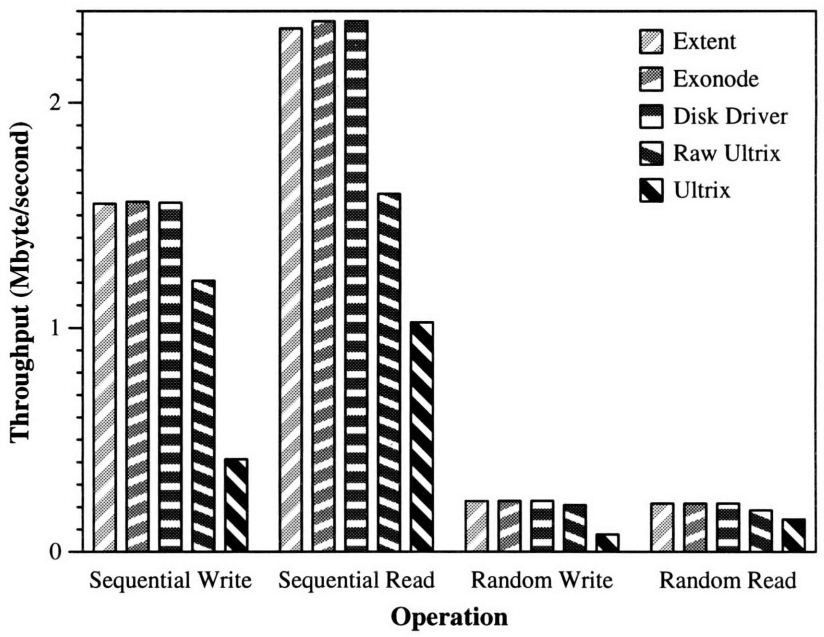

Figure 5-3: Large File I/O Performance. Bars represent throughput in MByte per second. Larger bars represent better results.

seconds for all file organizations) and negligible for deallocation (less than 0.035 seconds), with the exception of Ultrix files (1.39 seconds).

For large file I/O, the differences between file I/O using self-protected extents or exonodes in the exodisk system and file I/O using the Aegis disk driver is in the noise. Storage protection using self-protected extents or exonodes thus causes no overhead for large I/O when compared to direct access to the disk through the disk driver.

The experiments using the Ultrix raw disk interface have up to 32.4% less throughput than the experiments using self-protected extents, exonodes and the Aegis disk driver. The results show that the Ultrix raw disk interface does not fully exploit the available disk bandwidth. The experiments using the Ultrix file system show between 32.7% and 73.4% less throughput than the experiments using self-protected extents, exonodes and the Aegis disk driver. These results underline the high overhead of the Ultrix file system, even in the absence of frequent metadata manipulations (as incurred for the small file I/O experiments).

5.2.5 Conclusions for Base Protection Overhead

For small files, exonodes incur little overhead (less than 10%) when compared to the raw Aegis disk driver. Several exonodes can be clustered into a single sector which amortizes metadata *reads* over accesses to the data of several files. For large files, exonodes incur no overhead. Self-protected extents incur considerable overhead (up to 86.6%) for small files while not incurring any overhead for large files when compared to the experiments using the Aegis disk driver. Using scatter-gather I/O for reading self-protected extents as well as better caching and prefetching policies than the on-demand caching and LRU replacement policies used in above experiments, it should be possible to further reduce the overhead of storage protection in the exodisk system.

The performance of the Ultrix file system, which shows considerably worse performance than the Aegis disk driver and the exodisk system, underlines the observation that high-level storage abstractions severely limit the performance of applications. This is only partially caused by the seemingly poor performance of the Ultrix driver.

5.3 Benefits of Fine-grained Interleaving

This section uses a very simple set of micro-benchmarks to (roughly) quantify the performance benefits of fine-grained interleaving. The experiments run two applications at the same time, each of which first sequentially allocates and writes, then sequentially reads, then randomly writes (picking the file at random), then randomly reads, and finally sequentially deallocates 1,000 10 KByte files. As with the Exonode experiments in Section 5.2, ten exonodes are stored in one sector, followed by the data of the ten files. The data of one file is transfered in one disk request. The experiments use three different data layouts on disk:

Partitioned. The exonodes and data of each application are allocated in one contiguous area on disk, and the data of the two applications is 250 MByte apart.

Neighboring. The exonodes and data of each application are allocated in one contiguous area on disk, and the data of one application is allocated directly after the data of the other application.

Interleaved. The exonodes and data of both applications are interleaved with one exonode and the data for ten files of one application following the exonode and data for ten files of the other application.

The experimental results are shown in Figure 5-4 (except for Deallocate which is between 2.5 and 2.8 milliseconds). The numerical results are listed in Appendix A.

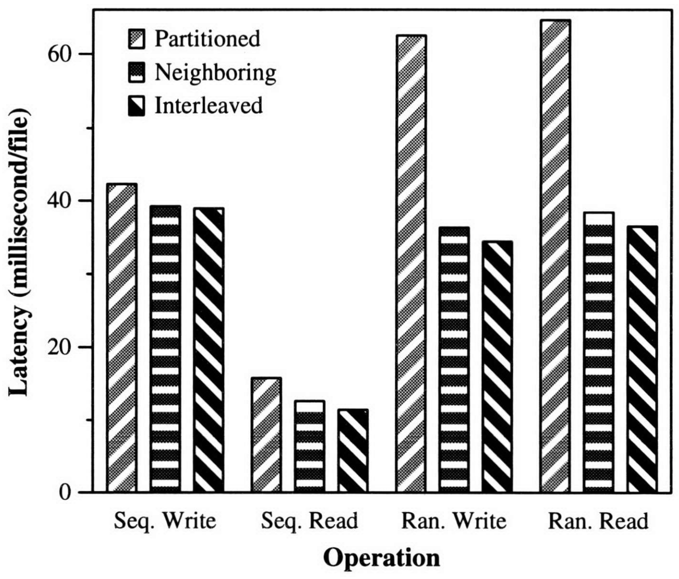

Figure 5-4: Small File I/O Performance: Partitioned vs. Interleaved. Bars represent average latency per file in milliseconds. Smaller bars represent better performance. Seq. Write includes file allocation. Deallocate is not shown (less than 0.1 for all organizations).

Both the Neighboring and Interleaved layouts show between 7.3% and 27.4% better performance for sequentially writing and reading the 1,000 10 KByte files than the Partitioned layout. They show between 40.6% and 45.0% better performance for randomly writing and reading the 1,000 10 KByte files. The Interleaved layout shows approximately the same performance as the Neighboring layout for sequentially writing the 1,000 files and between 4.9% and 9.5% better performance than the Neighboring layout for the other operations. The differences between the co-locating layouts (Neighboring and Interleaved) and the Partitioned layout are not as pronounced for sequentially accessing the files, because the micro-benchmarks issue disk requests for ten files at a time without synchronizing (only for sequential access). The disk scheduler can thus schedule the disk requests of both applications to reduce the seek times.

The results underline the importance of fine grained-interleaving: Fine grain interleaving avoids the inflexibility of fixed partitions and can also lead to improvements in application performance.

5.4 Application-Specific Storage Management

The main goal of the exodisk system is to allow for the construction of applications that are very difficult or very expensive (in terms of performance) to implement on top of traditional

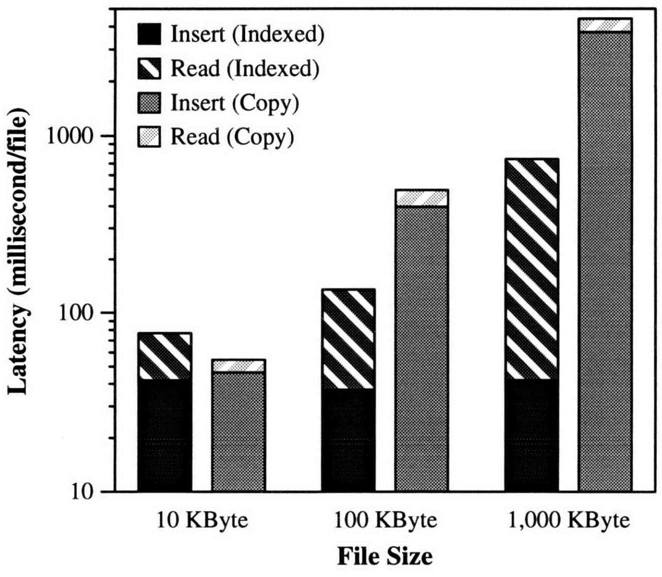

Figure 5-5: Performance of File Insertion. Bars represent average latency per file in milliseconds. Smaller bars represent better results.

file system interfaces. This section attempts to illustrate some of the provided flexibility and its potential impact with several examples. Each example exploits the exodisk system by utilizing application-specific metadata structures, data layouts, and/or persistence.

5.4.1 File Insertion

Conventional file system interfaces are not well-suited for inserting data into a file. To insert data into a file, an application must copy the entire file after the insertion point and then insert the data while copying the file. Alternatively, applications can implement their own index-management on top of the file system to map logical file positions into the file provided by the file system. While this method avoids the high overhead of copying the entire file, it forces the application to replicate functionality already present in the file system and may result in unfavorable performance since both mappings are maintained separately [50]. In contrast, the exodisk system allows applications to define and manipulate the bottom layer of metadata. One can exploit this freedom to efficiently implement a file *insert* operation.

The performance of a simple insert operation that exploits the flexibility of the exodisk system (Indexed) is compared with insertion by copying the entire file (Copy). The experiments insert 16 bytes into the middle of 100 10 KByte files, 100 100 KByte files and 10 1,000 KByte files, and then sequentially read the same files. The performance results are shown in Figure 5-5. The numerical results are listed in Appendix A.

The Indexed experiments use application-defined data in a file's exonode to represent the mapping of logical file position onto disk storage. The file data itself is stored in extents, protected by the same exonode. The mapping uses an ordered list of triples, where each triple indicates how many bytes of file data are stored in each extent. On an insert operation, the extent with the file data is split at the insert location and a new extent is allocated to contain the inserted data and the original file data after the insert location within the same 4 KByte block. The file's index-table is updated accordingly, and the new extent and the file's exonode are written to disk. On a *read* operation, the file's data is read in the order described in the index-table. The Copy experiments simply insert the new data while copying the entire file after the insertion point. The Indexed experiments show a constant latency for inserting the data into the file, independent of the size of the file. They show 40.7% more latency than the Copy experiments for 10 KByte files (due to the slower read performance), a factor of four better performance for 100 KByte files, and a factor of six better performance for 1,000 KByte files.

5.4.2 Disk-Directed I/O

Disk-directed I/O [26] is a straight-forward attack on the mechanical disk bottleneck. Instead of issuing one disk request at a time in an issue-wait-process cycle and waiting for it to complete (as is done, for example, when reading a list of files in a conventional file system), one issues all of the requests without synchronizing, allows the disk to service them in any order it wants (which hopefully minimizes the total disk access time) and processes the requests as they complete. Aggressive prefetching based on application hints (e.g., [39]) is very similar in behavior; the application (or, more precisely, the file system) issues requests in advance and allows the disk system to service them out of order, but the application still processes them in order. This approach can significantly improve performance in modern disk systems, which are characterized by (1) multiple disks disguised as a single disk (i.e., disk arrays [38]), or (2) disks that aggressively schedule requests based on information not available to the host system (e.g., both the seek time and rotational latency [47, 24]).

The low-level interface provided by the exodisk system makes using disk-directed I/O trivial. One can simply initiate multiple requests, wait for notification of each request's completion and process it. To illustrate this, a very simple disk-directed I/O application is used. It generates a list of N random disk locations to be read and processed (for this

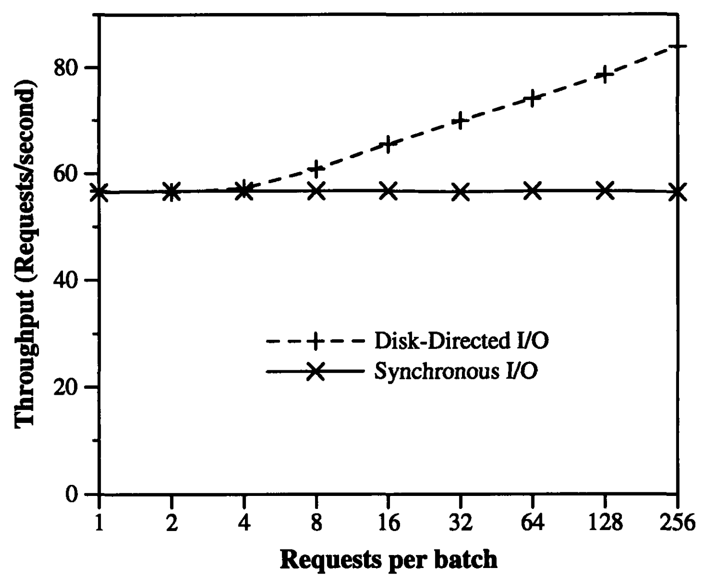

Figure 5-6: Performance of Disk-directed I/O. Each datum represents the average number of disk requests serviced per second. Larger numbers are better.

experiment, the processing of each request consists simply of copying the data to another buffer). In the base case, one sector at each of the N locations is read and processed (one at a time) in list order. For the disk-directed I/O case, the N requests are all initiated (without waiting for completion) and processed as they complete.

Figure 5-6 shows the experimental results. Each experiment issues a total of 10,240 random *read* requests to a 122 MByte file (implemented using a self-protected extent). As expected, disk-directed I/O outperforms the base case, and the improvements increase with the number of requests. The benefit comes from two sources: (1) the better overlapping of computation and I/O, and, more significantly, (2) the improved I/O efficiency because of disk scheduling (the device driver is using C-LOOK). The performance difference should be considerably larger when using modern disk drives that internally implement very aggressive scheduling algorithms.

Of course, this is a very simple example designed to illustrate the ease of using this approach with the exodisk system. Kotz [26] and Patterson *et al.* [39] much more thoroughly demonstrate the benefits of doing so.

5.4.3 Application-Specific Persistence

Because applications can implement their own file abstractions (or choose from a library of pre-developed ones), varying persistence models can be easily implemented on top of the exodisk system. Applications can thus decide how long to cache dirty file data (either in application space or in an in-kernel buffer cache, see Section 2.3.4) and how long to cache dirty extent headers and exonodes in the exodisk cache before writing them to disk. Furthermore, applications do not need to use the same persistence model for all files, but can use different persistence models for different files.

The concept of allowing different files to have different integrity and persistence characteristics is not new. For example, temporary and memory-based file systems [35, 30] were added to systems so that files that are known to be short-lived do not have to suffer the high cost of file creation and deletion imposed by conventional file systems. One major difference between these approaches and the exodisk system model is that they partition the files with specific characteristics into separate file systems, imposing restrictions on naming and data placement. The exodisk system allows any characteristics (except for violating the minimal exodisk system integrity rules) to be used for any disk data, independent of what name is associated with it and where it is stored.

A set of micro-benchmarks is used to quantify the performance of three different persistence models. The experiments allocate and write, then read, and finally deallocate 1,000 10 KByte files. As with the Exonode experiments in Section 5.2, ten exonodes are stored in one sector, followed by the data of the ten files. The data of one file is transfered in one disk request (if it is transfered, see below). The experiments use three different persistence models:

Memory. No data or metadata is written to disk. Exonodes are allocated and deallocated through the exodisk system but never written to disk. File data is stored in a buffer in application space.

Metadata. No data is written to disk. Exonodes are synchronously written to disk directly after allocation and deallocation. File data is stored in a buffer in application space.

Data. Data and metadata are written to disk. Exonodes are synchronously written to disk directly after allocation and deallocation. File data is synchronously written to and read from disk.

The experimental results are shown in Figure 5-7. The numerical results are listed in Appendix A. The Memory model writes all data into an application buffer (providing a very simple buffer cache) and all exodisk system metadata into the exodisk cache, giving

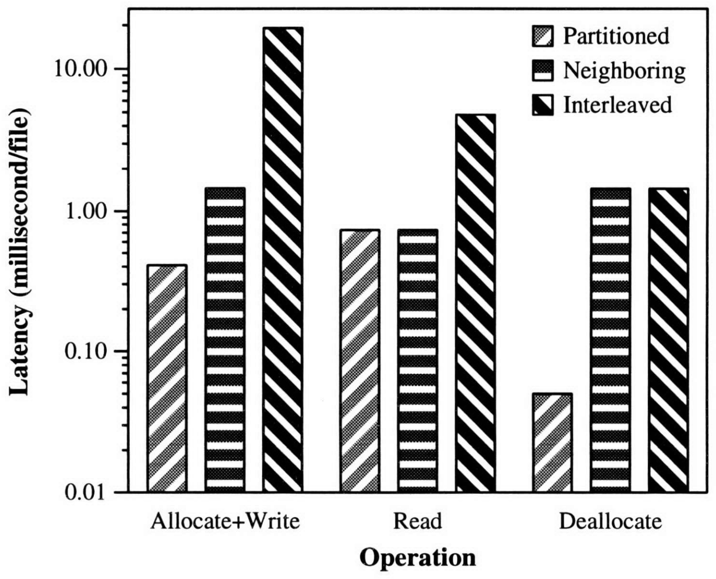

Figure 5-7: Performance of Persistence Models. Bars represent the average latency per file in milliseconds. Small bars represent better results.

no persistence guarantees and showing the best performance. The Metadata model writes all exonodes to disk and thus ensures that both file allocation and file deallocation are persistent. The performance of the Allocate+Write and Deallocate operations is determined by writing the sectors with exonodes to disk. Since both operations write the same sectors in the same order, they show the same latency. The Data model synchronously writes both data and metadata to disk directly after creation, giving the strongest persistence guarantees but paying the performance penalty of frequent synchronous writes. However, it does not use a buffer to store the file data. Memory shows a factor of 21 better performance than Data. This result is (roughly) comparable to performance results for the Andrew file system benchmark under the different persistence models reported by Chen *et al.* in [6] (which show a factor of 25 performance difference between the Memory File System [30] and the Unix File System with write-through after each write). Most applications will likely choose a persistence model between the two extremes of the Memory and the Data model and use a buffer cache to delay writes (similarly to the Unix Fast File System [29]).

The results show the impact of different persistence models on application performance. Applications can choose which model to use for each file and thus make their own trade-offs between the strength of the persistence model and application performance. They can also dynamically alter and fine-tune their choices, depending on their specific performance requirements and computing environment. For example, Chen shows in [5] that the optimal delay for writing dirty file system data from the buffer cache back to disk is dependent on the size of the file cache and the disk transfer rate. Any of these persistence models can be efficiently implemented on top of the exodisk system.

5.5 Summary

The micro-benchmarks in this chapter show that the cost of protection in the exodisk system is minimal in most cases (less than 10%, and often in the noise). Scatter-gather I/O can be used to significantly reduce the noticeable overhead for reading self-protected extents. The results also show the poor performance of a full-featured file system (the Ultrix file system) for some applications.

The exodisk system enables the fine-grained interleaving of disk storage, which avoids fixed partitions and can significantly reduce application running times. For example, for two applications accessing 1,000 10 KByte files each, fine-grained interleaving reduces the latency of some operations by up to 45% when compared to allocating the data of the two applications 250 MByte apart.

The exodisk system also enables new storage abstractions and operations. A simple file *insert* operation that exploits the freedom of the exodisk system performs up to a factor of six better than insertion by copying the entire file. In disk-directed I/O, applications issue several requests at the same time, let the disk system service them in any order, and process the requests as they complete. Disk-directed I/O processes 24% more requests per second than synchronous I/O when 32 requests are issued at the same time, and 39% more requests when 128 requests are issued at the same time. Finally, applications can implement their own persistence models (on a per-file basis) and can thus make their own trade-offs between persistence and performance.

Chapter 6

Related Work

A variety of previous work is related to the connection between applications and storage management. This work can be separated into three main categories: work related to giving applications control over storage management (Section 6.1), work focused on doing the exact opposite (i.e., raising the level of abstraction and doing the work "behind the scene"—Section 6.2), and work related to expanding the set of options available for applications to choose among (Section 6.3).

6.1 Giving Control to Applications

The storage subsystems of IBM mainframes (for example, the IBM 3990 I/O Subsystem as described in [22]) allowed users to construct I/O control programs to be executed directly on channel processors. This provided a great deal of freedom to applications, but did not provide for protection (e.g., applications could easily interfere with one another).

Recent work in application-controlled storage management has focused on allowing applications to direct caching and prefetching activity, either by giving applications direct control over caching and prefetching [3, 2], or by using a cost-benefit model in connection with application supplied hints [39]. This leads to significant performance improvements (e.g., over 40% reduction in the execution time of some applications), but addresses only one part of the spectrum of storage management issues. The work reported in this thesis builds upon this earlier work in suggesting a more radical and complete shift in responsibility for storage management from the kernel (or trusted servers) to applications.

The idea of allowing applications to control resource management in a protected way is not new and has been suggested, pursued and applied for a variety of other computer system resources. The exodisk system takes this philosophy directly from the exokernel operating system architecture [10, 11], which in turn builds upon a variety of other work. The contribution of this thesis is in applying this philosophy to the storage subsystem, which has some unique problems, in developing a system that addresses these problems and in demonstrating the potential value of this new storage architecture.

6.2 Raising the Level of Abstraction

An approach that is the exact opposite of the one taken in this thesis is to raise the level of abstraction by presenting some form of logical interface to disk storage. Loge [12], Mime [7], the Logical Disk [8, 20], the HP AutoRAID System [52] and Network-Attached Secure Disks (NASD) [17, 18] all use a logical representation of disk storage through a level of indirection. Such systems can be designed to significantly reduce the efforts of application-writers by isolating them from the complexities of storage management (the storage system itself decides on where to place data and can also transparently reorganize the data) and by providing transactional semantics (as is done in Mime and the Logical Disk). Such an approach also provides the storage subsystem implementer with greater design freedom, which can (in some circumstances) be translated into improved performance.

The main problem with this approach (ignoring the mapping costs that invariably seem to plague such systems) is that these high-level interfaces are thrust upon *all* applications, including those that would be better off without them. The exodisk system grew from the philosophy that the lowest-level that is common to all applications should be as simple and unobtrusive as possible. Higher-level abstractions, such as these logical disk systems, can be implemented on top of the common layer, allowing for the best of both worlds.

6.3 Supporting Varied Requirements

A variety of specialized file system implementations have been developed over the years to address particular application needs or behavior. Examples are immediate files [33], temporary and memory-based file systems [35, 30], log-structured file systems [42, 44, 46], clustered files [40, 31], semantic file systems [19], and stackable file systems [21]. Abstractions have been developed to allow these multiple storage management schemes to co-exist, ranging from common interface definitions (for example, vnodes as described in [25]) to file systems that propose to incorporate them all (for example, [34, 49]).

The large variety of file abstractions that are appropriate to different environments can be viewed as a powerful argument for constructing storage systems in such a way that application-writers can choose those abstractions that suit them best and can easily devise and employ new storage abstractions as their needs evolve. It does not seem possible to incorporate the superset of requirements in any storage subsystem, accessible through a common interface—and certainly not with any kind of reasonable performance.

Chapter 7

Conclusion

As many researchers have demonstrated, application control over various aspects of storage management significantly improves application performance. The exodisk system takes this approach to its logical conclusion by giving applications full control over *all* aspects of storage management in a secure way. The exodisk system proposed in this thesis securely multiplexes disk storage at a fine-grain (i.e., the disk sector). It uses self-protected extents and exonodes (application-configured *i*-nodes) to provide access control and leaves all other storage management decisions to applications. The protection overhead of the exodisk system is minimal (less than 10%, and often in the noise) for most operations and data organizations when compared to the performance of the disk driver.

To illustrate the flexibility of the exodisk system, three examples of specialized storage abstractions have been evaluated, namely fast file insertion, disk-directed I/O, and application-specific persistence. Each of these exploits some freedom provided by the exodisk system that is not found in conventional storage systems, improving performance substantially (up to a factor of six).

Appendix A

Numerical Results

This appendix lists the numerical results for the small file I/O experiments using 1,000 10 KByte files (Table A.1), the small file I/O experiments using 10,000 1 KByte files (Table A.2), the large file I/O experiments (Table A.3), the experiments comparing the partitioned, neighboring and interleaved layouts (Table A.4), the file insertion experiments (Table A.5) and the experiments comparing different persistence models (Table A.6). The results are discussed in Chapter 5.

	Allocate+Write	Read	Deallocate
Extent	17.8	8.7	7.4
Exonode	19.1	4.7	1.4
Disk Driver	17.7	4.6	0
Raw Ultrix	17.9	8.7	0
Ultrix	65.5	21.1	36.3

Table A.1: Small File I/O Performance: 1,000 10 KByte Files. Each datum gives the average latency per file in milliseconds. Each experiment was conducted five times. Standard deviation for Extent, Exonode, Disk Driver and Raw Ultrix is less than 0.25, for Ultrix less than 1.9. Deallocation for Disk Driver and Raw Ultrix takes no time since nothing needs to be deallocated.

	Allocate+Write	Read	Deallocate
Extent	14.0	2.8	16.7
Exonode	15.4	1.8	1.4
Disk Driver	14.0	1.6	0
Raw Ultrix	14.0	3.6	0
Ultrix	49.7	7.1	33.7

Table A.2: Small File I/O Performance: 10,000 1 KByte Files. Each datum gives the average latency per file in milliseconds. Each experiment was conducted five times. Standard deviation for Extent, Exonode, Disk Driver and Raw Ultrix is less than 0.5, for Ultrix less than 5.3. Deallocation for Disk Driver and Raw Ultrix takes no time since nothing needs to be deallocated.

	Seq. Write	Seq. Read	Ran. Write	Ran. Read
Extent	1.55	2.33	0.23	0.22
Exonode	1.56	2.36	0.23	0.22
Disk Driver	1.56	2.36	0.23	0.22
Raw Ultrix	1.21	1.59	0.21	0.19
Ultrix	0.41	1.03	0.08	0.15

Table A.3: Large File I/O Performance. Each datum gives the average throughput in MByte per second. Each experiment was conducted five times. Standard deviation is less than 0.030 for Sequential Write and Sequential Read, and less than 0.001 for Random Write and Random Read. Results for file allocation and deallocation are not shown. They are minimal for allocation (less than 0.06 seconds for all organizations) and negligible for deallocation (less than 0.035 seconds) with the exception of Ultrix files (1.39 seconds).

	Seq. Write	Seq. Read	Ran. Write	Ran. Read	Deallocate
Partitioned	42.3	15.7	62.5	64.6	2.8
Neighboring	39.2	12.6	36.3	38.4	2.6
Interleaved	38.9	11.4	34.4	36.5	2.5

Table A.4: Small File I/O Performance: Partitioned vs. Interleaved. Each datum gives the average latency per file in milliseconds. Each experiment was conducted five times. Standard deviation is less than 0.65 for all operations and organizations. Seq. Write includes file allocation.

	10 KByte	100 KByte	1,000 KByte
Insert (Indexed)	41.9	37.3	41.9
Read (Indexed)	34.9	97.7	698.4
Insert (Copy)	46.3	396.1	3,727.2
Read (Copy)	8.3	97.7	696.3

Table A.5: Performance of File Insertion. Each datum gives the average latency per file in milliseconds. Each experiment was conducted five times. Standard deviation is less than 2.7 for all operations, except for Copy Insert with 1,000 KByte files (26.3).

	Allocate+Write	Read	Deallocate
Memory	0.41	0.73	0.05
Metadata	1.45	0.73	1.44
Data	19.1	4.8	1.44

Table A.6: Performance of Persistence Models. Each datum gives the average latency per file in milliseconds. Each experiment was conducted five times. Standard deviation is less than 0.008 for all operations and persistence models, except for Read under the Data model (0.31).

Bibliography

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 198–212, Pacific Grove, California, October 1991.

[2] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. Technical report, Princeton University, 1994. CS-TR-94-493.

[3] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 165–177, Monterey, California, November 1994.

[4] D. L. Chaum and R. S. Fabry. Implementing capability-based protection using encryption. Technical Report UCB/ERL M78/46, University of California at Berkeley, July 1978.

[5] Peter M. Chen. Optimizing delay in delayed-write file systems. Technical report, University of Michigan, May 1996.

[6] Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani, and Christopher M. Aycock. The Rio file cache: Surviving operating system crashes. Technical Report CSE-TR-286-96, University of Michigan, March 1996.

[7] Chia Choa, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, Hewlett Packard, 1992.

[8] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 15–28, Ashville, North Carolina, December 1993.

[9] Maria R. Ebling and M. Satyanarayanan. SynRGen: An extensible file reference generator. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, Tennessee, May 1994. Also available as Technical Report CMU-CS-94-119.

[10] Dawson R. Engler. The design and implementation of a prototype exokernel operating system. Master's thesis, Massachusetts Institute of Technology, 1995.

[11] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[12] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing disk controller. In *Proceedings of 1992 Winter USENIX*, pages 237–251, San Francisco, California, January 1992.

[13] Digital Equipment. DECstation 5000/133 Product Description. `http://ftp.digital.com/pub/Digital/DECinfo/SOC/Feb94/ch-3-b.txt`.

[14] Digital Equipment. The RZ25: Digital's Highest Performing 3.5-Inch SCSI Disk - 426 MBF. S30108 Sales Update Dated 15-JUL-91.

[15] Gregory R. Ganger, Robert Grimm, M. Frans Kaashoek, and Dawson R. Engler. Application-controlled storage management. Technical Report TM-551, MIT Laboratory for Computer Science, 1996.

[16] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 49–60, Monterey, California, November 1994.

[17] Garth Gibson. Network-attached secure disks (NASD). `http://www.cs.cmu.edu/afs/cs/project/pdl/WWW/Rpt/NASD_Feb96.ps`, February 1996.

[18] Garth Gibson. Network-attached secure disks (NASD). `http://www.cs.cmu.edu/afs/cs/project/pdl/WWW/ftp/NASD_ARPA_Feb16_96.ps%`, February 1996.

[19] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Semantic file systems. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 16–25, Pacific Grove, California, October 1991.

[20] Robert Grimm, Wilson C. Hsieh, Wiebren de Jonge, and M. Frans Kaashoek. Atomic recovery units: Failure atomicity for logical disks. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[21] John Heidemann and Gerald Popek. Performance of cache coherence in stackable filing. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 127–142, Copper Mountain Resort, Colorado, December 1995.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, first edition, 1990.

[23] Gordon Irlam. Unix file size survey—1993. World-Wide Web, 1993. `http://www.base.com/gordoni/ufs93.html`.

[24] David M. Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7rev1, Hewlett-Packard, 1991.

[25] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of 1986 Summer USENIX*, pages 238–247, Atlanta, Georgia, June 1986.

[26] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 61–74, Monterey, California, November 1994.

[27] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[28] David Mazieres. Resource protection. Personal Communication, May 1996.

[29] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[30] Marshall Kirk McKusick, Michael J. Karels, and Keith Bostic. A pageable memory based filesystem. In *Proceedings of 1990 Summer USENIX*, pages 137–143, Annaheim, California, June 1990.

[31] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of 1991 Winter USENIX*, pages 33–43, Dallas, Texas, January 1991.

[32] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Project Athena technical plan, Massachusetts Institute of Technology, October 1988.

[33] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *Software—Practice and Experience*, 14(4):365–368, April 1984.

[34] Keith Muller and Joseph Pasquale. A high performance multi-structured file system design. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 56–67, Pacific Grove, California, October 1991.

[35] Masataka Ohta and Hiroshi Tezuka. A fast /tmp file system by delay mount option. In *Proceedings of 1990 Summer USENIX*, pages 145–150, Anaheim, California, June 1990.

[36] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[37] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, Washington, December 1985.

[38] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of SIGMOD*, pages 109–116, Chicago, Illinois, June 1988.

[39] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, Colorado, December 1995.

[40] J. Kent Peacock. The Counterpoint fast file system. In *Proceedings of 1988 Winter USENIX*, pages 243–249, Dallas, Texas, February 1988.

[41] John S. Quarterman, Abraham Silberschatz, and James L. Peterson. 4.2BSD and 4.3BSD as examples of the UNIX system. *Computing Surveys*, 17(4):379–418, 1985.

[42] Mendel Rosenblum. *The Design and Implementation of a Log-structured File System.* PhD thesis, University of California at Berkeley, 1992. Also available as Technical Report UCB/CSD 92/696.

[43] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995.

[44] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[45] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[46] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of 1993 Winter USENIX*, pages 307–326, San Diego, California, January 1993.

[47] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of 1990 Summer USENIX*, pages 313–324, Washington, DC, January 1990.

[48] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the 1995 USENIX Technical Conference*, pages 325, 249–264, New Orleans, Louisiana, January 1995.

[49] Raymie Stata. File systems with multiple file implementations. Master's thesis, Massachusetts Institute of Technology, February 1992. Also available as Technical Report MIT/LCS/TR-528.

[50] Michael Stonebraker. Operating system support for database management. *Readings in Database Systems*, pages 167–173, 1988.

[51] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall Inc., Englewood Cliffs, N.J. 07632, 1992.

[52] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 96–108, Copper Mountain Resort, Colorado, December 1995.

[53] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 241–251, Nashville, Tennessee, May 1994.