Design Evolution of the EROS Single-Level Store*

Jonathan S. Shapiro
Systems Research Laboratory
Johns Hopkins University
shap@cs.jhu.edu

Jonathan Adams
Distributed Systems Laboratory
University of Pennsylvania†
jonathan-adams@ofb.net

Abstract

File systems have (at least) two undesirable characteristics: both the addressing model and the consistency semantics differ from those of memory, leading to a change in programming model at the storage boundary. Main memory is a single flat space of pages with a simple durability (persistence) model: all or nothing. File content durability is a complex function of implementation, caching, and timing. Memory is globally consistent. File systems offer no global consistency model. Following a crash recovery, individual files may be lost or damaged, or may be *collectively* inconsistent even though they are individually sound.

Single level stores offer an alternative approach in which the memory system is extended all the way down to the disk level. This extension is accompanied by a transacted update mechanism that ensures globally consistent durability. While single level stores are both simpler and potentially more efficient than a file system based design, relatively little has appeared about them in the public literature. This paper describes the evolution of the EROS single level store design across three generations. Two of these have been used; the third is partially implemented. We identify the critical design requirements for a successful single level store, the complications that have arisen in each design, and the resolution of these complications.

As the performance of the EROS system has been discussed elsewhere, this paper focuses exclusively on *design*. Our objective is to both clearly express how a single level store works and to expose some non-obvious details in these designs that have proven to be important in practice.

1 Introduction

Single level stores simplify operating system design by removing an unnecessary layer of abstraction from the system. Instead of implementing a new and different semantics at the file system layer, a single level store extends the memory mapping model downwards to include the disk. Where conventional operating systems use the memory mapping hardware to translate virtual page addresses to physical pages, single level stores map virtual page addresses to *logical* page addresses, using physical memory as a software-managed cache to hold these pages.

The most widely-used single level store design is probably the IBM System/38, more commonly known as the AS/400 [IBM98]. At the hardware level, the AS/400 is a capability-based object system. The AS/400 design treats the entire store as a unified, 64-bit address space. Every object is assigned a 16 megabyte segment within this space. Persistence is managed explicitly – changes to objects are rewritten to disk only when directed by the application. While the protection architecture and object structure of the AS/400 is described by Soltis [Sol96], key details of its single-level store implementation are unpublished.

Like the AS/400, EROS is a capability-based single level store design. Unlike AS/400, EROS manages persistence transparently using an efficient, transacted checkpoint system that runs periodically in the background. Applications rely on the kernel to transparently handle persistence, leaving the applications free to build data structures and algorithms without regard to disk-level placement or the need to protect recoverability through careful disk write ordering. The EROS system and its performance have been described elsewhere [SSF99]. This paper describes how the EROS single level store design is integrated into the system and its key components, and the evolution of this design over the last decade.

As an initial intuition for single level stores, imagine a system design that begins by assuming that the machine never crashes. In such a system, there would be no need for a file system at all; the entirety of the disk is used as a large paging area. Such a design would clearly eliminate a large body of code from a conventional operating system. The EROS system, including user-mode applications that implement essential functions, is currently 103,712 lines of code.[1] Excluding drivers, networking protocols, and include files, the Linux 2.4 kernel contains 383,698 lines of code, of which 283,956 implement support for various file systems and file mapping. While the two systems implement very different semantics, their functionality is comparable, and the EROS code provides features

* This research was supported by DARPA under contract #N66001-96-C-852. Additional support was provided by Panasonic Information Technology Laboratories, Inc. and VMware, Inc.

† Author now with Sun Microsystems, Inc.

[1] EROS drivers and network stack are implemented outside the kernel. Driver and network code size is not included in either estimation of code size.

that Linux lacks: on restart, EROS recovers processes and interprocess communication channels in addition to data objects.

The design challenges of a single-level store are (1) to address the problem that systems actually *do* crash, and ensure that consistency is preserved when this occurs, (2) to devise some efficient means of addressing this very large paging area, and (3) to provide some means for specifying desired locality, preferably in a fashion informed by application-level semantic knowledge. This paper describes three designs that meet these challenges in two different EROS kernel designs. Other potential applications of these design ideas include database storage managers, storage-attached networks, and logical volume systems.

The first design presented is the one used by KeyKOS, which was inherited by the original EROS system in 1991. This design suffered from minor irritations that caused us to revise the design in 1997. In 2001, it was decided to remove drivers from the EROS kernel entirely, which forced us to rethink and partially rebuild the single level store yet again. Aside from the storage allocator itself, all of these designs present identical external semantics to applications.

The balance of this paper proceeds as follows. We first provide a brief overview of the EROS object system, its storage model and the mechanism used to ensure global consistency. This discussion introduces the critical requirements that must be satisfied by a single-level store design. We then describe the user-level storage allocator, which bears responsibility for locality management and storage reclamation. We then describe each of the existing design generations in turn, and the motivation behind each revision. The paper concludes with related work, lessons learned, and some hints as to our future plans.

2 Object System Overview

EROS is a microkernel design. The kernel implements a small number of object types, but leaves storage allocation, fault handling, address space management, and many other traditional kernel functions to user-level code. For this paper, the most important function implemented by user-level code is the *space bank* (Section 5). The space bank is responsible for all storage allocation, for storage quota enforcement, for bulk storage reclamation, and for disk-level object placement. At the kernel interface, all of this is accomplished by allocating and deallocating objects with appropriately selected unique object identifiers. Every object has a unique object identifier (OID). OIDs directly correlate to disk locations, which enables the space bank to perform object placement.

The EROS kernel design consists of three layers (Figure 1). The *machine layer* stores process and memory mapping information using a representation that is convenient to the hardware. For process state, this representation is determined by the design of the hardware context switch mechanism. For memory mapping state, it is determined by the design of the hardware memory management unit. These layers are managed as a cache of selected objects that logically reside in the object cache. When necessary, entries in the process cache or memory mapping tables are either written back or invalidated. Entries in the process cache correspond to entries in the process table of a more conventional design, but processes may be moved in and out of the process cache several times during their lifespan.

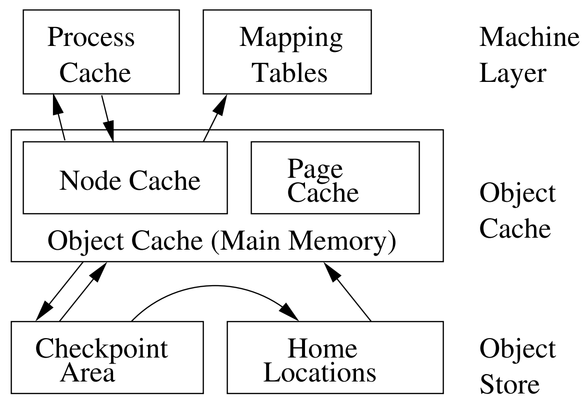

Figure 1: EROS design layers.

The *object cache* occupies the bulk of main memory. At this layer there are only two types of objects: pages and nodes. Pages hold user data. Nodes hold capabilities. Every node and page has a corresponding location in the home location portion of the object store. As with the machine layer, the object cache is a software-managed cache of the state on the disk.

As is implied by Figure 1, all higher-level operating system abstractions are composed from these two fundamental units of storage. Process state is stored in nodes, and is loaded into the process cache at need. Address space mappings are likewise represented using trees of nodes. These are traversed to construct hardware mapping data structures as memory faults occur. Details of these transformations can be found in [SSF99, SFS96].

The *object store* layer is the object system as it exists on the disk. At this layer the system is divided into two parts: the "home locations," which provide space for every object in the system, and the "checkpoint area," which provides the means for building consistent snapshots of the system. All object writes are performed to the checkpoint area. Revised objects are migrated to their home locations only after a complete, system-wide transaction has been successfully committed. The checkpoint mechanism is described in Section 3.2.

Collectively, these layers implement a two-level caching design. At need, the entire user-visible state of the system can be reduced to pages and nodes, which can then be written to disk.

3 Storage Model

The main reason for having two types and sizes of objects is to preserve a partition between data and capabilities. Data resides in pages and capabilities reside in nodes. It is certainly possible to design a single level store in which all objects are the size of a page, but it proved inconvenient to do so in EROS for reasons of storage efficiency. In the current implementation, capabilities occupy 16 bytes and nodes 544 bytes, but these sizes are not exposed by any external system interface. This leaves us free in the future to change the size of capabilities compatibly, much as was done in the AS/400 transition from 48-bit to 64-bit addresses.

Every object in the store has a unique object identifier (OID). Objects on the disk are named by operating-system protected capabilities [Dv66], each of which contains an object type (node, page, or process), the OID of the object it names, and a set of permissions. An EROS object capability is similar to a page table entry (PTE) that contains the swap location of an out-of-memory page.[2] When an object capability is actively in use, the EROS kernel rewrites the capability internally to point directly at the in-memory copy of the object.

Pages contain user data. Nodes contain a fixed-size array of capabilities, and are used as indirect blocks, memory mapping tables, and as the underlying representation for process state. Within the store, these nodes are packed into page-sized containers called "node pots." All I/O to or from the store is performed in page-sized units.

3.1 Home Locations

Every object in the EROS store has a uniquely assigned "home location" on some disk. Some versions of EROS implement optional object mirroring, in which case the same object may appear on the disk at multiple locations and is updated (when needed) at all locations. Additional mirroring or RAID storage may be performed by the storage controller. This is invisible to the EROS kernel.

The bulk of the disk space in an EROS system is used to contain the "home locations" of the objects. The basic design requirements for this part of the store are:

- Object fetch and store should be efficient. As a result, there should be a simple, in-memory strategy for directly translating an OID value to the disk page frame (the home location) that contains the object.
- EROS does not expose the physical locations of objects outside the kernel; only OIDs are visible, and these only to selected applications. For purposes of locality management, there must be some well-defined relationship between OID values and disk locations.

Without an in-memory algorithm to translate an OID into a disk object address, the store would require disk-level directory data structures that would in turn require additional, sequentially dependent I/O accesses to locate and fetch an object. The elimination of such additional accesses is a performance-critical imperative. The encoding of object locations, the organization of the disk, and the locality management of home locations has been the main focus of evolution in the design of the storage manager.

3.2 Checkpointing

To ensure that global consistency for all processes and objects is maintained across restarts, it is sufficient for the kernel to periodically write down an instantaneous snapshot of the state of the corresponding pages and nodes. To accomplish this, EROS implements an efficient, asynchronous, system-wide checkpoint mechanism derived from the checkpoint design of KeyKOS [Lan92].

The checkpoint system makes use of a dedicated area on the disk. This area is also used for normal paging, and is conceptually equivalent to the "swap area" of a conventional paging system. Before any node or page is made dirty, space is reserved for it in the checkpoint area. As memory pressure induces paging, dirty objects are paged out to (and if necessary, reread from) the checkpoint area. Object locations in the checkpoint area are recorded in an in-memory directory.

Periodically, or when the checkpoint area has reached a predefined occupancy threshold, the kernel declares a "snapshot," in which every dirty object in memory is marked "copy on write." Simultaneously, a watermark is made in the checkpoint area. Everything modified prior to the snapshot will be written beneath this watermark; everything modified after the snapshot is written above it. The kernel now allows execution to proceed, and initiates background processing to flush all of the pre-snapshot dirty objects into the previously reserved space in the checkpoint area. Checkpoint area I/O is append-only. If an object is dirtied multiple times, no attempt is made to reclaim its

[2] In the PTE case, no object type is needed because PTEs can only name pages.

previously occupied space in the checkpoint area. This ensures that checkpoint I/O is mostly sequential.

Once all objects have been written to the checkpoint area, an area directory is written and a log header is rewritten to capture the fact that a consistent system-wide transaction has been completed. A background "migrator" now copies these objects back to their home locations in the store. Whenever a checkpoint transaction completes, the checkpoint area space occupied by the previous transaction is released. To ensure that there is always space in the checkpoint area for the next checkpoint, migration is required to complete before a new checkpoint transaction can be completed.

The net effect of the checkpoint system is to capture a consistent system-wide image of the system state. If desired, checkpoints can be run at frequencies comparable to those of conventional buffer cache flushes, making the potential loss of data identical to that of conventional systems. To support the requirements of database logs, there is a special "escape hatch" mechanism permitting immediate transaction of individual pages.

3.3 Intuitions for Latency

While EROS is not yet running application code, KeyKOS has been doing so since 1980, supporting both transaction processing and (briefly) general purpose workloads. The performance of the checkpoint design rests on two empirical observations from KeyKOS:

- Over 85% of disk reads are satisfied from the checkpoint area.
- Over 50% of dirty objects die or are redirtied before they are migrated. Such objects do not require migration.

These two facts alter the seek profile of the system, reducing effective seek latencies for reads. They also alter the rotational delay profile of the system, reducing effective rotational latencies for writes. Our goal in this section is to provide an intuition for why this is true. While the specific measurements obtained from KeyKOS probably will not hold for EROS twenty years later, we expect that the performance of checkpointing will remain robust. We will discuss the reasons for this expectation below.

It is typical for the amount of data included in a given checkpoint to be comparable to the size of main memory. The checkpoint area as a whole must be able to hold two checkpoints. Given a machine with 256 megabytes of memory, the expected checkpoint area would be 512 megabytes. On a Seagate Cheetah (ST373405LC, 29550 cylinders, 68 Gbytes), this region would occupy 0.7% of the disk, or 216 cylinders. As 100% of normal writes and 85% of all reads occur within this region, the arm position remains within a very narrow range of the disk with high probability.

Estimating disk latencies is deceptive, because computations based on the published minimum, average, maximum seek times have nothing to do with actual behavior. Seek time profiling is required for effective estimation. The seek time calculations presented here are based on profiling data collected by Jiri Schindler using DIXtrac [SG99]. We emphasize that these are computed, rather than measured results. We will assume a disk layout in which the checkpoint area is placed on the middle cylinders of the drive.

3.3.1 Expected Read Behavior

The disk head position in KeyKOS has a non-uniform distribution. To compute the expected seek time, we must consider the likely location of the preceding read as well as the current one (Table 1), giving an expected seek time for reads of 2.92ms on a drive whose *average* seek time is 6.94 ms. This expectation is robust in the face of both changes to the checkpoint region size and reasonable reductions in checkpoint locality. Increasing the checkpoint area size to 200 cylinders raises the expected seek time to 3.05 ms. Reducing the checkpoint "hit rate" to 70% yields an expected seek time of 3.80 ms.

Conventional file system read performance is largely determined by the average seek delay (in this case, 6.94 ms). For comparing read delays, rotational latencies can be ignored: the expected rotation delay on both systems is one half of a rotation per read.

The difference in expected performance is largely immune to changes in extent size or prefetching, as both techniques can be used equally well on both systems. Both techniques reduce the total number of seeks performed; neither alters the underlying seek latency or distribution. Similarly, low utilization yields similar benefits in both systems by reducing the effect of long seeks. The read performance of both designs converges on the performance of the checkpointing design as utilization falls – on sufficiently small, packed data sets there is no meaningful difference in seek behavior.

3.3.2 Expected Write Behavior

As in log-based file systems, a checkpointing design potentially performs two writes for every dirty block: one to the checkpoint area and the second to the home locations. Migration is skipped for data that is remodified or deleted between the time of checkpoint and the time of migration. Given this, there are two thresholds of interest:

Current I/O	Preceding I/O	Distance	Time	Weighted
Checkpoint (85%)	Checkpoint (85%)	108 cyl	1.97 ms	1.42 ms
Checkpoint (85%)	Other (15%)	7387 cyl	5.27 ms	0.67 ms
Other (15%)	Checkpoint (85%)	7387 cyl	5.27 ms	0.67 ms
Other (15%)	Other (15%)	14775 cyl	**6.94 ms**	0.16 ms
		Weighted Seek Time		2.92 ms

Table 1: **Expected read latency**. Based on seek profile of the Seagate Cheetah ST373405Lc. The reported average seek time for this drive is 6.94 ms.

1. How many objects live long enough to get checkpointed.
2. Of those, how many live long enough to be migrated.

The best available data on file longevity is probably the data collected by Baker *et al.* [BHK+91]. Figure 4 of their paper indicates that 65% to 80% of all files live less than 30 seconds, that 80% of all files live less than 300 seconds, and that 90% of all files live less than 600 seconds (one checkpoint interval). We can estimate from this that less than half of all files that are checkpointed survive to be migrated. This is consistent with the measured behavior of KeyKOS: only 50% of the checkpoint data survives to be migrated.

The impact of this is surprising. Imagine that there are 400 kilobytes of file data to be written to a conventional file system. The key question proves to be: what are the run lengths? Figure 1 of the Baker measurements [BHK+91] shows that most file run lengths are small. While the figure does not differentiate read and write run lengths, Table 3 suggests that write run lengths are primarily driven by file size: 70% of all bytes written are “whole file” writes. Figure 2 shows that 80% of files are 10 kilobytes or less. Taken together, these numbers mean that the cost of bulk flushes of the data cache are dominated by rotational delay. The 400 kilobytes in question will be written at nearly 40 distinct locations, each of which will require 1/2 a rotation to bring the head to the correct position within the track. On the Cheetah, the rotational delay alone comes to 119 ms. Seek delays depend heavily on filesystem layout, but the same considerations apply in both checkpointed and conventional designs. If the runs are uniformly spread across the drive, the seeks (on the Cheetah) will come to an additional 112 ms, for a total of **231 ms**.

Now consider the same 400k under the checkpointing design. KeyKOS and EROS perform this write using bulk I/O and track at once operations. Depending on the drive, 400 to 600 kilobytes can be written to the checkpoint area in one seek (weighted cost 2.46ms on the Cheetah) plus 1.5 rotations (1/2 to start, 1 to complete) for a total of 11.42 ms. We must now consider the cost of migration.

The KeyKOS/EROS migration I/O behavior looks exactly like that of a file cache that uses deferred writes. Because the file semantics is unchanged this I/O has similar run lengths, and like the buffer cache flush it is done using bulk-sorted I/O. Seek times are amortized similarly because there are a large number of available blocks to write. The difference is that the migrated blocks have a longer time to die, and that the amount of data migrated is therefore half of the data that will be written by the deferred-write buffer cache. Because half of the data will die before migration, only 56 ms will be spent in rotational delay rewriting it and 70.47 ms of seek times (again under the uniform distribution assumption). The combined total cost of the checkpoint and migration writes is **137.89 ms**.

4 Locality and Object Allocation

There are two primary issues that impact the design of a single level store. The first is common to all disk-based storage designs: locality. It is necessary that the object allocation mechanism provide means to arrange the disk-level placement of objects for reasonable locality. In all generations of the EROS store this is accomplished by preserving a correlation between OID values and disk positions. The second is object allocation: because all objects must be recoverable after a crash, all allocations must (logically) be recorded using on-disk data structures.

4.1 Content Locality

The value of locality in general-purpose workloads is often misunderstood. While sequential data placement for file content is extremely important in the case of a single request stream, it is much less important when multiple accesses to disk occur concurrently. Disk-level traces collected by Ruemmler and Wilkes [RW93] show that it is very rare to see more than 8 kilobytes of sequential I/O at the level of the disk arm. While average file sizes have grown since that time, and (we presume) modern sequential accesses would be longer than 8 kilobytes, the underlying reasons for the limited dynamic sequentiality have not changed:

- Paging I/O is limited by the size of a page.
- Many file I/Os involve sequentially dependent accesses, as when traversing metadata.
- Directory I/Os are frequent. Directories are usually small, and the corresponding I/Os are therefore short.
- While read-ahead helps, excessive read ahead is counterproductive. Successful read-ahead works equally well in both designs, and can be thought of as achieving a larger extent size.
- The request streams compete for attention at the disk arm. Even if a second, potentially sequential I/O request is initiated quickly by the application or the operating system, the interrupt-level logic has already initiated an arm motion if multiple requests are present, preventing immediate service of the sequential request.

Two facts suggest that file system sequentiality may be important only up to a limited extent size. Log-structured designs organize data by temporal locality rather than spatial locality. In spite of this, read performance for general-purpose workloads is not degraded significantly in log-structured designs [SSB$^+$95]. It has also been established that file system aging ultimately has a more significant impact on overall I/O performance than the logging/clustering choice [SS95].

The EROS store design effectively optimizes for both cases. Newly modified objects are stored in the checkpoint area according to temporal locality, but the small size of the checkpoint area ensures physical locality as well. Data in home locations is placed according to locality determined at object allocation time, which is when maximal semantic knowledge (and therefore maximal knowledge of likely reference patterns) is available.

While single level stores do not always implement directories and indirect blocks in the style of file systems, the corresponding *concepts* are implemented elsewhere in the operating system, and the basic usage patterns involved are ultimately driven by application behavior. Similar extent size arguments should therefore apply in single level stores. This introduces a significant degree of freedom into file system or single level store design – one that we plan to leverage in the next generation of our store.

4.2 Metadata Locality

A more pressing issue in the EROS single-level store is *metadata* locality. In the Berkeley Fast File System design [MJLF84], for example, block location occurs through a two-stage hybrid translation scheme. The first stage translates the inode number to the inode data structure. This translation is performed at file open time, and the result is cached in an in-memory inode. The second stage traverses the file indirect blocks to locate individual blocks in the file. These indirect blocks are cached in memory according to the same rules as other data blocks, but due to higher frequency of access are likely to remain in memory for active files.

An EROS address space is a peristent mapping from offsets to bytes. Address spaces need not be associated with processes, and EROS therefore uses them to hold file data as well as application memory images. As a result, the EROS metadata that is most closely analogous to conventional file metadata is EROS address space metadata.

An EROS address space is organized as a tree of nodes whose leaves are pages (Figure 2), much as a UNIX file is organized as a tree of indirect blocks whose leaves are data blocks. Because EROS nodes are much “narrower” than typical indirect blocks, the height of the address space tree for any given address space is taller than the height of the indirect block tree for a UNIX file of corrresponding file ($log_{32}(size) > log_{256}(size)$). Any tree traversal of this type implies sequential disk accesses with associated seek delays. The Ruemmler data shows that these seeks are frequently interspersed with other accesses when multiple request streams are present.

Request interleave in turn interacts badly with typical, non-backtracking disk arm scheduling policies, and can lead to as much as a full disk seek before the next block down the tree will be fetched.[3] Because of their greater tree height, EROS address spaces potentially involve more levels of traversal, and it is correspondingly more important to managing the locality and prefetching of nodes within an address space. The discussion of the EROS space bank below (Section 5) describes how this locality is achieved.

4.3 Allocation Performance

The final performance issue in single level stores is the efficiency of object allocation – particularly with respect to ephemeral allocations such as heap pages or short-lived file content. In a conventional file system, these ephemeral blocks are allocated from swap space and do not survive system shutdown or failure. Because there is no expectation that these allocations are preserved across restarts, in-memory data structures and algorithms can be used to implement them. Linux, for example, keeps an in-memory allocation bitmap for each swap area [BC00].

There is no way to persistently store ephemeral allocation

[3] Many newer drives implement backtracking seeks, but doing so raises both convergence and variance issues that must be avoided by the operating system in real-time applications.

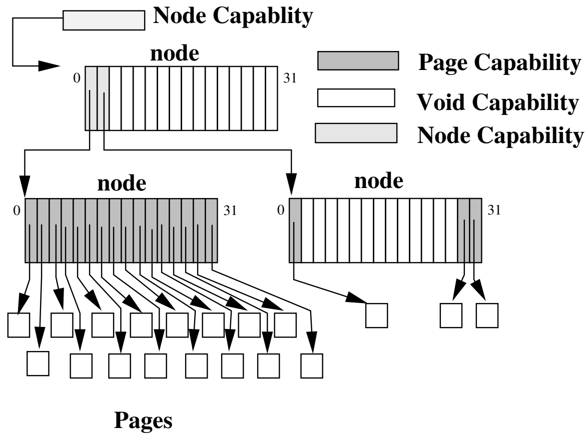

Figure 2: An EROS address space.

data without incurring *some* disk I/O overhead. The challenge in a single-level store is to keep this overhead to a minimum. EROS accomplishes this in two ways:

1. No recording of allocation is performed for objects that are allocated and deallocated within the same checkpoint interval. This largely recaptures the efficiency advantages of conventional swap area allocations.

2. In the current and previous versions of the EROS store, the store is divided into regions, each of which has an overhead page containing bits that indicate whether an object in the region is empty (zero). Sequential allocations first pull in the overhead page, but then avoid I/O's for successive objects within the same region. Deallocations update the bit rather than the on-disk object, with similar I/O reductions.

5 The Space Bank

The EROS storage manager, known as the "space bank," is a user-mode application that performs all storage allocation in the EROS system. There is a hierarchy of logical space banks, all of which are implemented by a single server process. Each logical bank:

- Allocates and deallocates individual pages and nodes on request.

- Remembers what objects have been allocated from that logical bank, so that they can be bulk-reclaimed when the bank is destroyed.

- Provides locality of allocation so long as this is feasible on the underlying disk, up to the limit of the system-designed extent size.

- Impose optional limits (quotas) on the total number of pages and nodes that can be allocated from that logical bank.

- Provides means to create "child" banks whose storage comes from the parent, creating a hierarchy of storage allocation.

5.1 Extent Caching

A naive implementation of the space bank would allocate one object at a time, recording each allocation in some suitable ordered collection. Typically, each dynamically allocated object in the system has associated with it at least one logical bank. For example, most address spaces are implemented as "copy on write" versions of some existing space, and the copied pages (and the nodes that reference them) are allocated from a per-space bank. One impact of this is that space bank invocations are frequent. As a result the single object approach would not provide good locality.

An obvious solution would be to allocate storage to each bank in extents, and allow the bank to suballocate objects from this extent. Unfortunately, this doesn't work well either. If we imagine that each extent contains 64 pages, and that there is some variation in address space sizes, we must conclude that when each address space, process, or other synthesized object has been completely allocated there would remain within its bank a partially allocated extent. In the absence of empirical data, we should expect that this residual extent would on average be half allocated. Unfortunately, there is no simple way to know which banks are done allocating. This means that there would be a very large number of outstanding banks (one per process, one per file, etc.) each of which has committed to it 32 page frames of disk storage that will never be allocated.

The solution to this is ***extent caching***. Instead of associating an individual extent with each bank, the space bank maintains a cache (~128 lines) of "active" extents. Each of these extents begins at an OID corresponding to a 64 page boundary on the disk and contains 64 page frames worth of OIDs (some of which may already be allocated). Extent caches are typed: nodes and pages are allocated from distinct extents, which helps to preserve metadata locality. The extent cache design relies on the fact that sequential OIDs correspond with high likelihood to sequential disk locations.

Every bank is associated with a line in the extent cache by a hash on the address of the logical bank data structure. When a bank needs to allocate an object, it first checks availability in its designated cache line. If that extent has no available space, then a "cache miss" occurs, and an attempt is made to allocate a fresh extent from the underlying disk space. If this proves impossible, as when disk space is near exhaustion, the needed object will be allocated from the first extent in the extent cache that has available space.

The effect of the extent cache is to ensure that banks receive sequential objects in probabilistic fashion up to the limits of the extent size. While it is possible for two banks that are simultaneously active to hash to the same extent, we have not observed it to be a problem. A secondary effect of the extent cache is that the disk page frame allocation map is consulted with reduced frequency. This is desirable because consulting the allocation map involves a linear search through a page and consequently flushes the CPU data cache, which has a significant impact on allocation speed. Our first space bank implementation did this, and we found that the cost of data cache reconstruction after allocation overwhelmed all other costs.

When objects are deallocated, they are restored to the extent cache only if the containing extent is still in the cache. Otherwise, they are returned directly to the free map. Newly freed objects are reused aggressively, because reusing objects that are still in memory eliminates extra disk I/Os that would record their deallocation. Reuse of old objects is deferred. The assumption behind this is that the objects allocated to a given bank share a common temporal extent and will tend to be deallocated as a group. Given this, it is better to wait as long as possible before reusing the available space in an older extent in order to maximize the likelihood that the entire extent has become free.

To support address space metadata locality, the space bank implements a two-level allocation scheme for nodes. The extent cache caches page frames. The space bank suballocates nodes sequentially from these frames. Because address spaces are constructed by copy on write methods, and because the copy on write process proceeds top down in the node tree (Figure 2), it is usual for the entire path of nodes from the root to the first page to be allocated from a single page frame on the disk. When the top node is fetched, its containing page frame is cached in the page cache. The effect of this is that the entire sequence of nodes from the address space root node to the referenced page is brought into memory with a single disk I/O.

5.2 Record-Keeping Locality

Conceptually, each space bank's record of allocated objects can be kept by any convenient balanced tree structure. The current EROS implementation uses a red-black tree. There are two potential complications that need to be considered in building this tree.

The first is sheer size. Collectively, the number of RB-tree nodes is on the same order as the number of disk objects. While these structures might fit within the space bank's virtual address space on current machines, they certainly will *not* fit within physical memory in large system configurations. However clever the data structure, care must be taken to ensure that traversals of these structures do not suffer from poor locality due to space bank heap fragmentation. This type of poor locality translates directly into paging. The current space bank implementation does not attempt to manage this issue, which is a potentially serious flaw. A simple solution would be to allocate tree nodes using an extent caching mechanism similar to the one already used for nodes and pages, or a slab-like allocation mechanism [Bon94, BA01].

The second is space overhead. Each OID occupies 64 bits, and it would be disproportionate to spend an additional two or three pointers per object to record allocations. As a result, the current bank tree nodes record extents rather than OIDs, and use a per-extent allocation bitmap to record which objects within an extent have been allocated. Even if two or three banks are simultaneously performing allocations from the same extent cache entry, the net space overhead of this is lower than per-OID recording. The current implementation relies on everything fitting within virtual memory. On the Intel x86 family implementation, this will be adequate until the attached disk space exceeds 2.6 terabytes.

6 Two Early Disk-Level Designs

The original EROS system, including its store design, followed the published design of KeyKOS [Har85]. Each disk is divided into ***ranges*** of sequentially numbered objects. Ranges are partitioned by object type; a given range contains nodes or pages, but not both. Every object's OID consists of a 64 bit "coded disk address" concatenated with a 32 bit "allocation count." The coded disk address describes the location of the object, and the allocation count indicates how many times a particular object has been allocated. In order for a capability to be valid, the allocation count in the capability must match the allocation count recorded on the disk for the corresponding object.

6.1 Original Storage Layout

At startup, the kernel probes all disks, identifies the ranges present, and builds an in-memory table with an entry for each range:

(node/page, startOID, endOID, disk, startSec)

It also scans the checkpoint area to rebuild the in-memory directory of object locations.

For nodes, the allocation count is recorded in the node itself. Nodes are numbered sequentially within a range, and are packed in page-sized units called ***node pots***; no node on the disk is split across two pages. Once the containing range has been identified, the disk location of the relative disk page frame containing a node can be computed by

$$(OID - startOID)/NodesPerPageFrame$$

For pages, the allocation count cannot be recorded within the page, because all of the available bytes are already in use. Instead, page ranges are further subdivided into subranges. Each subrange begins with an ***allocation pot***, which is a page that contains the allocation counts for the following pages in the subrange (Figure 3). The allocation pot also contains a byte containing various flags for the page, including one indicating whether that page is known to hold zeros.

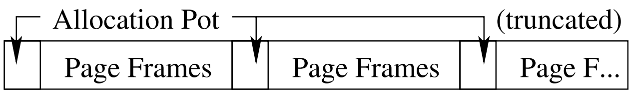

Figure 3: Page range layout.

Each allocation pot can hold information for up to 819 pages, so a page range is organized as a sequence of subranges, each 820 pages long and consisting of an allocation pot followed by its associated pages. Depending on the size of the underlying partition, the final subrange may be truncated. Once the containing range has been identified, the disk location of a relative disk page frame containing a page can be computed by

$$(OID - startOID) + (OID - startOID)/819 + 1$$

In either case, the relative frame can then be combined with the $startSec$ value to yield the starting sector for the I/O.

6.2 Unified Object Spaces

The design of Section 6.1 suffers from an irritating flaw: it partitions the disk into typed ranges. There is no easy way to know in advance the correct proportion of nodes to pages, and the design does not provide any simple means to reorganize the disk (there are no forwarding pointers) or to interconvert ranges from one type to another. We found that we were continuously adjusting our directions to the disk formatting program to add or remove objects of some type.

The solution was to adopt the page range layout for all ranges, and use an available bit in the allocation pot to indicate the "type" (node or page) of the corresponding disk page frame (Figure 4). If the frame type is "page," the allocation count in the allocation pot is the allocation count of the page, otherwise it is the *max* of the allocation counts of all nodes contained in the frame. The OID encoding was also reorganized, using the least 8 bits as the index of the object within the frame and the upper 56 bits as the "frame OID." The frame offset computation proceeds as previously described for page frames, with the caveat that the OID value must be shifted before performing the computation.

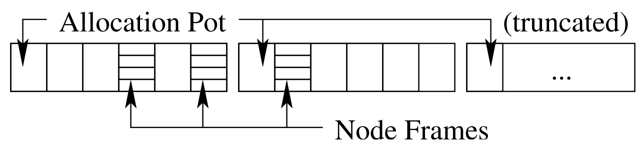

Figure 4: Unified range layout.

In the revised design, the kernel converts a frame from one type to another whenever an object of the "wrong" type is allocated by the space bank. The kernel assumes that the space bank has kept track of available storage, and that it will not unintentionally reallocate storage that is already in use. There is a minor complication, which is that the kernel must ensure that allocation count is never decreased by conversion. This is assured by setting the allocation count to $max(node\ alloc\ counts) + 1$ when converting a frame from nodes to pages and setting all node allocation counts to the page allocation count when converting a frame from pages to nodes.

The switch to unified ranges simplifies the kernel object management code, but more importantly it simplifies the use and allocation of disk storage. Disk frames can now be traded back and forth between types as needed. In addition to allowing address space metadata and data to be placed in a localized fashion (thereby facilitating read-ahead across object types), the new design can potentially be extended to perform defragmentation in order to improve extent effectiveness. Individual banks effectively record the relationships between pages, nodes, and their containing objects, and the ability to retype frames supports storage compaction. To perform compaction, two additional bits can be taken from the "flags" field to "lock" an object, copy it's content to a new destination frame, use the old frame to record the new location, and then use a second flags bit to mark the object forwarded. Either the space bank or a helper application can now iterate through all nodes, rewriting their capabilities to reflect the new object location.

7 Embedded EROS

In early 2000, as part of an exploratory research collaboration with Panasonic, we started to investigate the possibility of an embedded version of EROS for selected real-time applications. As part of this, a decision was made to remove all remaining drivers from the kernel. Together, these decisions introduced three new requirements into the overall system design:

1. In order to support DMA, we needed a way to support pages (but not nodes) whose physical memory address could be known to a driver.
2. The kernel now needed some mechanism for allocation of non-pageable and non-checkpointed objects.
3. Ordinary disk ranges now needed to be served by user-mode drivers.

To address these requirements, the "range" notion was generalized to the notion of ***object sources***. An object source implements some sequential range of OIDs. This range may be only partially populated.

By well-known convention, two ranges of OIDs are reserved. One corresponds to physical memory pages, while the other allocates non-pageable objects. EROS already uses main memory as an object cache. The physical page object source will allocate any OID whose range-relative frame index corresponds to a physical memory page frame that is part of the page cache. The effect of allocating a capability for such an OID is to evict the current resident of that page cache entry and relabel the entry as a physical page object. The non-pageable object range is similar, though there is no guarantee that the object will occupy any particular physical address. Both physical memory pages and non-pageable objects are exempted from checkpointing and eviction. When these objects are freed, the corresponding cache locations are returned to the object cache free pool for later reuse.

In the embedded design, the system is partitioned into a non-persistent space that contains drivers and the object store manager, and a persistent space that operates exactly as before. The driver portion of the system is loaded from ROM, and uses an ***object source registry*** capability to register support for a persistent range if one is to be implemented. The persistent OID range (if present) is "backed" by a user-mode object source driver, and the kernel defines a protocol by which this driver can insert or remove objects whose OIDs fall within the range it controls. When completed, there will also be a protocol by which the persistent source driver indicates how many dirty objects can be permitted for its range at any given time. The implementation of the checkpoint mechanism in this design is relocated to the persistent source driver; the kernel remains responsible for snapshot and for "writing back" the checkpointed objects to the persistent source driver.

8 The Vertical View

As with conventional file systems, the effectiveness of a single-level store design relies on the interaction of temporal, spatial, and referential efficiencies implemented cooperatively by several vertical layers in the system. This section briefly recaps the critical points in a single place so that their combined effect can be more readily seen. Each item is annotated by the section that discusses it.

Temporal Efficiency:

- The kernel ensures that objects that are allocated and deallocated within a single checkpoint interval generate no I/O to home locations provided that capabilities to them are never written to the disk. [4.3]
- The space bank eagerly reuses young, dead objects to reduce unnecessary recording of deallocations, and to aggressively reuse allocation pots that it knows must already be in memory. [5.1]

Spatial Efficiency:

- The space bank allocates objects using bank-wise extents, which helps to preserve disk-level locality. Separate extents are used for pages and nodes. [5.1]
- There is a direct correspondence between OIDs and page frame placement in the store, eliminating the need for directory or indirection blocks in the object store. [6.1,6.2]

Referential Efficiency:

- The address space copy on write implementation combines a dedicated bank (and therefore a dedicated extent) with top-down metadata traversal, ensuring that all "indirect blocks" in a given traversal will tend to be fetched in a single I/O operation. [4.2]
- Both the checkpoint and the migration systems use bulk, sorted I/O, reducing total seek latency in spite of performing a larger number of object writes. [3.3]
- Empty (zero) objects are neither written nor read to the home locations – only their allocation pots are revised. [6.1]
- Both the checkpoint directory [3.2] and the range table [6.1] are kept in memory. No additional disk I/Os are required to determine the location of a target page or node.

The combined effect of this may be illustrated by describing in more detail what happens when an object is to be loaded.

When a page or node is to be fetched, the EROS kernel first consults an in-memory object hash table to determine of the object is already in memory. This includes checking for the containing node pot or allocation pot as appropriate. Next, the checkpoint area directory is consulted to see if the current version of this object is located in the checkpoint area. If the object is not found in the checkpoint area, the range table is consulted and an I/O is initiated for the objects containing page frame and (if needed) its allocation pot. In the typical case, ignoring read-ahead, only one I/O is performed. The effectiveness of the node allocation strategy tends to yield one I/O for every 7 nodes (because 7 nodes fit in a node pot). Similarly, the allocation pot I/O is performed only once for a given 819 frame region in the home locations; the overhead of these I/Os is negligable.

The total effectiveness of the single-level store relies on collaboration between the storage allocator, its clients, the kernel, and the underlying store design. Each of these pieces, taken individually, is relatively straightforward.

Conceptually, this layering is not so different from what happens in a file system. An important difference is that in the file system design this layering is opaque. In EROS, it is straightforward to implement customized memory managers or use multiple space banks for more explicit extent management.

9 Related Work

While the idea of a single-level store is widely known among operating system implementors, relatively little has been published about their design. As mentioned in the introduction, the AS/400 is perhaps the best known implementation, but the only widely available reference on this design [Sol96] provides inadequate details. Even within IBM, information on the AS/400 implementation is closely held.

Both Grasshopper [DdBF$^+$94] and Mungi [HEV$^+$98] use single-level stores. Neither has published details on the storage system itself. Like the AS/400, persistence in the Mungi system is explicitly managed. Grasshopper's is transparent, but its strategy for computing transitive dependencies is both complex and expensive.

Consistent checkpointing has been the subject of several previous papers, most notably work by Elnozahy *et al.* [EJZ92] and Chandy and Lamport [CL85].

KeyKOS, from which the EROS design is derived, uses a single level store and consistent checkpoint mechanism described in [Lan92]. The design of the store itself has never previously been published.

The L3 system [Lie93] implemented a transparent checkpointing mechanism in its user-level address space manager. Like the KeyKOS and EROS checkpointing designs, this implementation uses asynchronous copy on write for interactive responsiveness. Fluke similarly implemented an experimental system-wide checkpoint mechanism at user level [TLFH96], but this implementation is a "stop and copy" implementation. Disk writes are performed before execution can proceed making the Fluke implementation unsuitable for interactive or real-time applications. Neither the L3 nor the Fluke checkpointers perform any sort of systemwide consistency check prior to writing a checkpoint, introducing the likelihood that system state errors resulting from either imperfect implementation or ambient background radiation will be rendered permanent.

The checkpoint design presented here is similar in many respects to the behavior of log-structured file systems such as LFS [SBMS93, MR92]. As with log-structured file systems, the the checkpoint mechanism converts random writes into sequential writes. Unlike the log-structured design, the EROS checkpointing design quickly converts this temporally localized data into physically localized data by migrating it into locations that were allocated based on desired long-term locality. The resulting performance remains faster than conventional file systems, but does not decay as file system utilization increases.

10 Future Work

The store designs described in sections 6 and 7 reflect a mature placement strategy that has been tested over a long period of time. While effective, this placement strategy suffers from a significant limitation: it is difficult to administer changes to the underlying disk configuration. While the EROS system implements software duplexing to allow storage to be rearranged, the rearrangement process is neither efficient nor "hands free." It would be better to have an automated means to take advantage of new and larger stores.

The current EROS space bank implementation can theoretically handle stores slightly larger than 2^{29} pages (2.6 terabytes). This corresponds approximately to one fully-populated RAID subsystem containing 10 drives, each providing 60 gigabytes of storage. While not common on the desktop, these configurations appear more and more frequently on servers. The paging behavior of the current space bank implementation would be quite bad for this size store. While the space bank could be reimplemented to eliminate thrashing during allocation, a better approach overall would be to loosen the association between OIDs and disk page frames. Extent-level locality is essential, but a sparsely allocatable OID space would eliminate the need for the red-black trees that are currently used to record object allocations.

From an addressing standpoint, larger stores do not present an immediate problem for EROS. The underlying OID space can handle stores of up to 2^{56} pages.[4] Given that there is no basic addressability problem, the key question is "how will we manage the growth?" The sequential allocation strategy used by the space bank does not do an effective job of balancing load over disk arms in the absence of a RAID controller. Further, the current mapping strategy from OIDs to disk page frames does not lend itself to physical rearrangement of objects as the available storage grows. All of these issues point to a need for a logical volume mechanism.

The next and (we hope) final version of the EROS single level store will likely be based on randomization-based extent placement. This is a nearly complete departure from the designs described here:

- Node and page OID spaces are once again partitioned.
- Allocation counts are abandoned. OIDs can be reallocated only if the space bank knows that no capability using the OID exists on the disk.
- The direct map from OIDs to disk ranges has been abandoned entirely. Objects are now placed using a randomization-based strategy.
- While extent-based object placement continues to be honored on a "best effort" basis using low-order bits of the OID, there is no longer any direct association between an OID and the location of an object on disk.
- Ranges can be dynamically grown or shrunk as disks are added and removed.
- Where the previous operations on ranges were "allocate" and "deallocate," the new design separates object allocation into two parts: storage reservation and object name binding.

The inspiration for this departure is a new, randomization-based disk placement strategy being explored within the Systems Research Laboratory based on prior work by Brinkmann and Scheideler [BSS00]. We plan to adopt a single-disk variant of this strategy in the EROS single-level store.

The key motivation for this change is the ability to divorce OIDs from physical placement without losing the ability to directly compute object addresses. Under the new strategy, disks or partitions can be added or removed from the system at will without needing to garbage collect or renumber OIDs, and data can be transparently shifted to balance load across available disk arms. This renders the system more easily scalable, and provides the type of load balancing and latency properties needed for multimedia applications.

11 Acknowledgements

The KeyKOS single level store was designed by Norm Hardy and Charles Landau while at Tymshare, Inc. Both have been helpful and patient in describing the workings of KeyKOS and encouraging the development and evolution of the EROS system. Bryan Ford was kind enough to explain the Fluke checkpointing implementations in some detail.

Jochen Liedtke similarly took time to explain the L3 checkpointing implementations. Jochen's continuous advances in the performance and design of microkernel operating systems led to improvements in the EROS implementation and drove us to a deeper and more careful understanding of operating system design.

12 Conclusions

This paper describes three working implementations of a single-level store. To our knowledge, this is the first time that any single-level store design has been comprehensively described in the public literature. The code is available online, and can be found at the EROS web site [Sha]. In describing our design, we have attempted to identify both the critical performance issues that arise in single level store designs and the solutions we have found to those issues.

One key to an effective single-level store is the interaction between temporal, spatial, and referential efficiency. This is made possible in EROS by the fact that disk-level locality information is rendered directly available for application use. Where file systems make locality decisions at the time the file is closed and the file cache is *flushed*, the EROS space bank makes these decisions when the corresponding storage is *allocated*, which is the point where maximal semantic knowledge of intended usage is at hand. An interesting challenge in the randomization-based design is to preserve an effective balance between spatial locality and adaptive scalability.

A surprising attribute of the EROS single level store is that in spite of its vertical integration it has undergone several major changes with minimal application-level impact. We have changed the capability size, the OID encoding, the checkpoint design, and removed the object driver from

[4] The disk drive industry has yet to produce 2^{56} pages of total disk storage over the lifespan of the industry, but will soon cross this mark.

the kernel. The only application code that changed was the space bank. Within the kernel itself, even the cache management code has gone largely unchanged as these modifications occurred.

From a design perspective, this paper has illustrated that single level stores simplify operating system design by removing an unnecessary layer of abstraction from the system. Instead of implementing a new and different semantics at the file system layer, a single level store extends the memory mapping model downwards to include the disk, allowing applications to control placement directly. In EROS these placement controls are generally provided by standard fault handling programs; most applications simply use these handlers, and require no code at all for storage management – separation of concerns is effectively maintained. On the other hand, applications with unusual requirements can replace these fault handlers if needed. The total EROS system size is roughly 25% that of Linux.

Microbenchmarks [SSF99] show that performance-critical object allocations in EROS are fast. Hand examination shows that the mechanisms described here are actually generating good disk-level locality. EROS-specific benchmarks show that EROS makes effective use of the available sustained disk bandwidth. In practice the main problem with checkpointing seems to finding a heuristic that does the associated I/Os *slowly* enough to avoid interfering with interactive processing. We also know that the KeyKOS database system, whose disk performance is critical, delivered exceptionally strong performance. All this being said, a key missing piece in this paper is application-level benchmarks. We are in the process of porting several server and client applications to EROS, and plan to measure application-level performance when this has been done.

This paper is dedicated to the memory of Prof. Dr. Jochen Liedtke (1953–2001).

13 About the Authors

Jonathan S. Shapiro is an Assistant Professor in the Computer Science department of Johns Hopkins University. His current research interests include secure operating systems, high-assurance software development, and adaptive storage management. He is also a key participant in the Hopkins Information Security Institute.

Jonathan Adams is a recent graduate of the California Institute of Technology, and is now a Member of Technical Staff in the Solaris Kernel Development group at Sun Microsystems.

References

[BA01] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many cpu's and arbitrary resources. In *Proc. 2001 USENIX Technical Conference*. USENIX Association, 2001.

[BC00] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Press, October 2000.

[BHK+91] 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 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, 1991.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[BSS00] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proc. of the 12th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 119–128, 2000.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[DdBF+94] Alan Dearl, Red di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, and Francis Vaughn. Grasshopper: An orthogonally persistent operating system. *Computer Systems*, 7(3):289–312, 1994.

[Dv66] J. B. Dennis and E. C. van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–154, March 1966.

[EJZ92] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[Har85] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software – Practice and Experience*, 28(9):901–928, 1998.

[IBM98] *AS/400 Machine Internal Functional Reference*. Number SC41-5810-01. IBM Corporation, 1998.

[Lan92] Charles R. Landau. The checkpoint mechanism in KeyKOS. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, September 1992.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 175–188. ACM, 1993.

[MJLF84] 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, 1984.

[MR92] J. Ousterhout M. Rosenblum. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, (1):26–52, February 1992.

[RW93] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *Proc. USENIX Winter 1993 Technical Conference*, pages 405–420, San Diego, California, January 1993.

[SBMS93] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 307–326, San Diego, CA, USA, 25–29 1993.

[SFS96] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. State caching in the EROS kernel – implementing efficient orthogonal persistence in a pure capability system. In *Proc. 7th International Workshop on Persistent Object Systems*, pages 88–100, Cape May, NJ, USA, 1996.

[SG99] Jiri Schindler and Gregory R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.

[Sha] Jonathan S. Shapiro. *The EROS Web Site*. http://www.eros-os.org.

[Sol96] Frank G. Soltis. *Inside the AS/400*. Duke Press, Loveland, Colorado, 1996.

[SS95] Keith A. Smith and Margo I. Seltzer. File system aging – increasing the relevance of file system benchmarks. In *Proc. 1995 USENIX Technical Conference*, pages 249–264, New Orleans, LA, USA, January 1995.

[SSB+95] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McManis, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proc. 1995 USENIX Technical Conference*, pages 249–264, New Orleans, LA, USA, January 1995.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.

[TLFH96] Patrick Tullmann, Jay Lepreau, Bryan Ford, and Mike Hibler. User-level checkpointing through exportable kernel state. In *Proc. 5th IEEE International Workshop on Object-Orientation in Operating Systems*, pages 85–88, October 1996.