Latency Management in Storage Systems

Rodney Van Meter*
Quantum Corp.
rdv@alumni.caltech.edu
Minxi Gao
U.C. Berkeley
minxi@eecs.berkeley.edu

Abstract

Storage Latency Estimation Descriptors, or SLEDs, are an API that allow applications to understand and take advantage of the dynamic state of a storage system. By accessing data in the file system cache or high-speed storage first, total I/O workloads can be reduced and performance improved. SLEDs report estimated data latency, allowing users, system utilities, and scripts to make file access decisions based on those retrieval time estimates. SLEDs thus can be used to improve individual application performance, reduce system workloads, and improve the user experience with more predictable behavior.

We have modified the Linux 2.2 kernel to support SLEDs, and several Unix utilities and astronomical applications have been modified to use them. As a result, execution times of the Unix utilities when data file sizes exceed the size of the file system buffer cache have been reduced from 50% up to more than an order of magnitude. The astronomical applications incurred 30-50% fewer page faults and reductions in execution time of 10-35%. Performance of applications which use SLEDs also degrade more gracefully as data file size grows.

1 Introduction

Storage Latency Estimation Descriptors, or SLEDs, abstract the basic characteristics of data retrieval in a device-independent fashion. The ultimate goal is to create a mechanism that reports detailed performance characteristics without being tied to a particular technology.

* Author's current address: Nokia, Santa Cruz, CA.

Storage systems consist of multiple devices with different performance characteristics, such as RAM (e.g., the operating system's file system buffer cache), hard disks, CD-ROMs, and magnetic tapes. These devices may be attached to the machine on which the application is running, or may be attached to a separate server machine. All of these elements communicate via a variety of interconnects, including SCSI buses and ethernets. As systems and applications create and access data, it moves among the various devices along these interconnects.

Hierarchical storage management (HSM) systems with capacities up to a petabyte currently exist, and systems up to 100PB are currently being designed [LLJR99, Shi98]. In such large systems, tape will continue to play an important role. Data is migrated to tape for long-term storage and fetched to disk as needed, analogous to movement between disk and RAM in conventional file systems. A CD jukebox or tape library automatically mounts media to retrieve requested data.

Storage systems have a significant amount of dynamic state, a result of the history of accesses to the system. Disks have head and rotational positions, tape drives have seek positions, autochangers have physical positions as well as a set of tapes mounted on various drives. File systems are often tuned to give cache priority to recently used data, as a heuristic for improving future accesses. As a result of this dynamic state, the latency and bandwidth of access to data can vary dramatically; in disk-based file systems, by four orders of magnitude (from microseconds for cached, unmapped data pages, to tens of milliseconds for data retrieved from disk), in HSM systems, by as much as eleven (microseconds up to hundreds of seconds for tape mount and seek).

File system interfaces are generally built to hide this variability in latency. A `read()` system call works the same for data to be read from the file system buffer cache

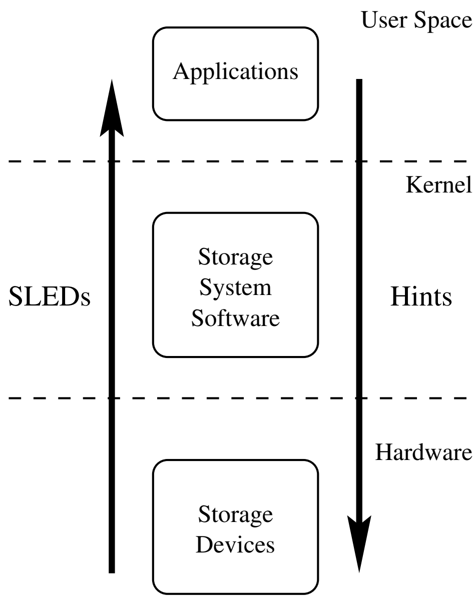

Figure 1: SLEDs and hints in the storage system stack

as for data to be read from disk. Only the behavior is different; the semantics are the same, but in the first case the data is obtained in microseconds, and in the second, in tens of milliseconds.

CPU performance is improving faster than storage device performance. It therefore becomes attractive to expend CPU instructions to make more intelligent decisions concerning I/O. However, with the strong abstraction of file system interfaces, applications are limited in their ability to contribute to I/O decisions; only the system has the information necessary to schedule I/Os.

SLEDs are an API that allows applications and libraries to understand both the dynamic state of the storage system and some elements of the physical characteristics of the devices involved, in a device-independent fashion. Using SLEDs, applications can manage their patterns of I/O calls appropriately. They may reorder or choose not to execute some I/O operations. They may also report predicted performance to users or other applications.

SLEDs can be constrasted to file system hints, as shown in Figure 1. Hints are the flow of information down the storage system stack, while SLEDs are the flow information up the stack. The figure is drawn with the storage devices as well as the storage system software participating. In current implementations of these concepts, the storage devices are purely passive, although their characteristics are measured and presented by proxy for SLEDs.

This paper presents the first implementation and measurement of the concept of SLEDs, which we proposed in an earlier paper [Van98]. We have implemented the SLEDs system in kernel and library code under Linux (Red Hat 6.0 and 6.1 with 2.2 kernels), and modified several applications to use SLEDs.

The applications we have modified demonstrate the different uses of SLEDs. `wc` and `grep` were adapted to reorder their I/O calls based on SLEDs information. The performance of `wc` and `grep` have been improved by 50% or more over a broad range of file sizes, and more than an order of magnitude under some conditions. `find` is capable of intelligently choosing not to perform certain I/Os. The GUI file manager `gmc` reports estimated retrieval times, improving the quality of information users have about the system.

We also modified LHEASOFT, a large, complex suite of applications used by professional astronomers for image processing [NAS00]. One member of the suite, `fimhisto`, which copies the data file and appends a histogram of the data to the file, showed a reduction in page faults of 30-50% and a 15-25% reduction in execution time for files larger than the file system buffer cache. `fimgbin`, which rebins an image, showed a reduction of 11-35% in execution time for various parameters. The smaller improvements are due in part to the complexity of the applications, relative to `wc` and `grep`.

The next section presents related work. This is followed by the SLEDs design, then details of the implementation, and results. The paper ends with future work and conclusions.

2 Related Work

The history of computer systems has generally pushed toward increasingly abstract interfaces hiding more of the state details from applications. In a few instances, HSM systems provide some ability for HSM-aware applications to determine if files are online (on disk) or offline (on tape or other low levels of the hierarchy). Microsoft's Windows 2000 (formerly NT5) [vI99], TOPS-20, and the RASH system [HP89] all provide or provided a single bit that indicates whether the file is online or offline. SLEDs extends this basic concept by providing more detailed information.

Steere's file sets [Ste97] exploit the file system cache on a file granularity, ordering access to a group of files to present the cached files first. However, there is no notion of intra-file access ordering.

Some systems provide more direct control over what pages are selected for prefetching or for cache replacement. Examples include TOPS-10 [Dig76] and TOPS-20 and Mach's user-definable pagers, Cao's application-controlled file caching [CFKL96], and the High Performance Storage System (HPSS) [WC95].

Still other systems have improved system performance by a mechanism known as *hints*. Hints are flow of information from the application to the system about expected access orders and data reuse. They are, in effect, the inverse of SLEDs, in which information flows from the system to the application. Hints may allow the system to behave more efficiently, but do not allow the application to participate directly in I/O decisions, and cannot report expected I/O completion times to the application or user. Good improvements have been reported with hints over a broad range of applications [PGG$^+$95]. Reductions in elapsed time of 20 to 35 percent for a single-disk system were demonstrated, and as much as 75 percent for ten-disk systems. Hints cannot be used across program invocations, or take advantage of state left behind by previous applications. However, hints can help the system make more intelligent choices about what data should be kept in cache as an application runs.

Hillyer and Silberschatz developed a detailed device model for a DLT tape drive that allows applications to schedule I/Os effectively [HS96a, HS96b]. Sandstå and Midstraum extended their model, simplifying it and making it easier to use [SM99]. The goal is the same as SLEDs, effective application-level access ordering, but is achieved in a technology-aware manner. Such algorithms are good candidates to be incorporated into SLEDs libraries, hiding the details of the tape drive from application writers.

For disk drives, detailed models such as Ruemmler's [RW94] and scheduling work such as Worthington's [WGP94] may be used to enhance the accuracy of SLEDs.

Real-time file systems for multimedia, such as Anderson's continuous media file system [AOG92] and SGI's XFS [SDH$^+$96], take reservation requests and respond with an acceptance or rejection. SLEDs could be integrated with such systems to provide substrate (storage and transfer subsystems communicating their characteristics to the file system to improve its decision-making capabilities), or to increase the usefulness of the information provided to applications about their requests. Such systems calculate information similar to SLEDs internally, but currently do not expose it to applications, where it could be useful.

Distributed storage systems, such as Andrew and Coda [Sat90], Tiger [BFD97], Petal [LT96], and xFS [ADN$^+$95], present especially challenging problems in representing performance data, as many factors, including network characteristics and local caching, come into play. We propose that SLEDs be the vocabulary of communication between clients and servers as well as between applications and operating systems.

Mobile systems, including PDAs and cellular phones, are an especially important area where optimizing for latency and bandwidth are important [FGBA96]. Perhaps the work most like SLEDs is Odyssey [NSN$^+$97]. Applications make resource reservations with SLEDs-like requests, and register callbacks which can return a new value for a parameter such as network latency or bandwidth as system conditions change.

Attempts to improve file system performance through more intelligent caching and prefetching choices include Kroeger's [Kro00] and Griffioen's [GA95]. Both use file access histories to predict future access patterns so that the kernel can prefetch more effectively. Kroeger reports I/O wait time reductions of 33 to 90 percent under various conditions on his implementation, also done on a Linux 2.2.12 kernel.

Kotz has simulated and studied a mechanism called *disk-directed I/O* for parallel file systems [Kot94]. Compute processors (CPs) do not adjust their behavior depending on the state of the I/O system, but collectively aggregate requests to I/O processors (IOPs). This allows the I/O processors to work with deep request queues, sorting for efficient access to achieve a high percentage of the disk bandwidth. Unlike SLEDs, the total I/O load is not reduced by taking dynamic advantage of the state of client (CP) caches, though servers (IOPs) may gain a similar advantage by ordering already-cached data to be delivered first.

Asynchronous I/O, such as that provided by POSIX I/O or VMS, can also reduce application wait times by overlapping execution with I/O. In theory, posting asynchronous read requests for the entire file, and processing them as they arrive, would allow behavior similar to SLEDs. This would need to be coupled with a system-assigned buffer address scheme such as containers [PA94], since allocating enough buffers for files larger than memory would result in significant virtual memory thrashing.

```
struct sled {
  long offset; /* into the file */
  long length; /* of the segment */
  float latency; /* in seconds */
  float bandwidth; /* in bytes/sec */
};
```

Figure 2: SLED structure

3 SLEDs Design

The basic elements of the Storage Latency Estimation Descriptor structure are shown in Figure 2. SLEDs represent the estimated latency to retrieve specific data elements of a file. A SLED consists of the byte offset within the file, the length in bytes of this section, and the performance estimates. The estimates are the latency to the first byte of the section and the bandwidth at which data will arrive once it has begun. Floating point numbers are used to represent the latency because the necessary range exceeds that of a normal integer. We chose floating point numbers for bandwidth for consistency of representation and ease of arithmetic.

Different sections (usually blocks) of a file may have different retrieval characteristics, and so will be represented by separate SLEDs. For large files, as a file is used and reused, the state of a file may ultimately be represented by a hundred or more SLEDs. Moving from the beginning of the file to the end, each discontinuity in storage media, latency, or bandwidth results in another SLED in the representation.

Applications take advantage of the information in a SLED in one of three possible ways: reordering I/Os, pruning I/Os, or reporting latencies. Each is detailed in the following subsections.

3.1 Reordering I/Os

By accessing the data currently in primary memory first, and items that must be retrieved from secondary or tertiary storage later, the number of physical I/Os that must be performed may be reduced. This may require algorithmic changes in applications.

Figure 3 shows how two linear passes across a file behave with normal LRU caching when the file is larger than the cache size. A five-block file is accessed using a cache which is only three blocks. The contents of the

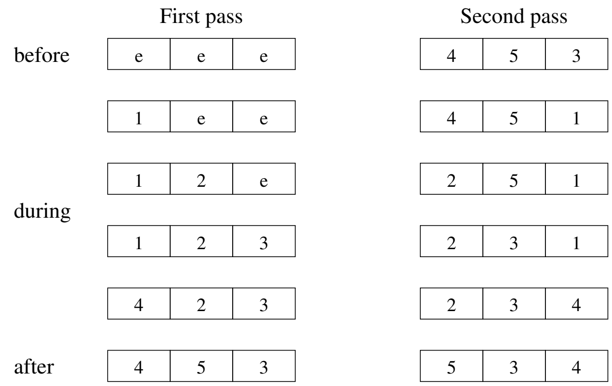

Figure 3: Movement of data among storage levels during two linear passes

cache are shown as file block numbers, with “e” being empty. The second pass gains no advantage from the data cached as a result of the first pass, as the tail of the file is progressively thrown out of cache during the current pass. The two passes may be within a single application, or two separate applications run consecutively. Behavior is similar whether the two levels are memory and disk, as in a normal file system, or disk and tape, as in any hierarchical storage management system which caches partial files from tape to disk.

By using SLEDs, the second pass would be able to recognize that reading the tail of the file first is advantageous. In this case, blocks 3, 4, and 5 would be known to be cached, and read before blocks 1 and 2. The total number of blocks retrieved from the lower storage level in this second pass would only be two instead of five.

3.2 Pruning I/Os

By algorithmically choosing not to execute particular I/Os, an application may improve its performance by orders of magnitude, as well as be a better citizen by reducing system load.

A simple example that combines both pruning and reordering is an application which is looking for a specific record in a large file or set of files. If the desired record’s position is toward the end of the data set as normally ordered, but already resides in memory rather than on disk or tape (possibly as a result of recent creation or recent access), it may be among the first accessed. As a result, the application may terminate without requesting data which must be retrieved for disk or tape, and performance may improve by an order of magnitude or more.

It may also simply be desirable to run an application with the minimum number of I/O operations, even at the cost of reduced functionality. This may be applied in, for example, environments that charge per I/O operation, as used to be common for timesharing systems.

3.3 Reporting Latency

Applications in several categories depend on or can be improved by an ability to predict their performance. Quality of service with some real-time component is the most obvious but not the only such category; any I/O-intensive application on which a user depends for interactive response is a good candidate to use SLEDs.

Systems that provide quality of service guarantees generally do so with a reservation mechanism, in which applications request a specific performance level, and the system responds with a simple yes or no about its ability to provide the requested performance. Once the reservation has begun, QoS systems rarely provide any additional information about the arrival of impending data.

Most applications which users interact with directly are occasionally forced to retrieve significant amounts of data, resulting in the appearance of icons informing the user that she must wait, but with no indication of the expected duration. Better systems (including web browsers) provide visible progress indicators. Those indicators are generally estimated based on partial retrieval of the data, and so reflect current system conditions, but cannot be calculated until the data transfer has begun. Dynamically calculated estimates can be heavily skewed by high initial latency, such as in an HSM system. Using SLEDs instead provides a clearer picture of the relationship of the latency and bandwidth, providing complementary data to the dynamic estimate, and can be provided before the retrieval operation is initiated.

Both types of applications above have a common need for a mechanism to communicate predicted latency of I/O operations from storage devices to operating systems to libraries to applications. SLEDs is one proposal for the vocabulary of such communication.

3.4 Design Limitations

SLEDs, as currently implemented, describe the state of the storage system at a particular instant. This state, however, varies over time. Mechanical positioning of devices changes, and cached data can change as a result of I/Os performed by the application, other applications or system services, or even other clients in a distributed system. Adding a lock or reservation mechanism would improve the accuracy and lifetime of SLEDs by controlling access to the affected resources.

Another possibility is to include in the SLEDs themselves some description of how the system state will change over time, such as a program segment that applications could use to predict which pages of a file would be flushed from cache based on current page replacement algorithms.

4 Implementation

The implementation of SLEDs includes some kernel code to assess and report the state of data for an open file descriptor, an `ioctl` call for communicating that information to the application level, and a library that applications can use to simplify the job of ordering I/O requests based on that information.

This section describes the internal details of the implementation of the kernel code, library and application modifications.

4.1 Kernel

We modified the Linux kernel to determine which device the pages for a file reside on, and whether or not the pages are currently in memory. All of the changes were made in the virtual file system (VFS) layer, independent of the on-disk data structure of ext2 or ISO9660.

A `sleds_table`, kept in the kernel, is filled by calling a script from `/etc/rc.d/init.d` every time the machine is booted. The `sleds_table` has a latency and bandwidth entry for every storage device, as well as NFS-mounted partitions and primary memory. The latency and bandwidth for both local and network file systems are obtained by running the `lmbench` benchmark [MS96]. The current implementation keeps only a single entry per device; for better accuracy, entries which account for the different bandwidths of different disk zones will be added in a future version [Van97]. The script fills the kernel table via a new `ioctl` call, `FSLEDS_FILL`, added to the generic file system `ioctl`.

Applications can retrieve the SLEDs for a file using another new `ioctl` call, `FSLEDS_GET`, which returns a vector of SLEDs. To build the vector of SLEDs and their latency and bandwidth, each virtual memory page of the data file is checked. After the kernel finds out where a page of the data file resides, it assigns a latency and bandwidth from the `sleds_table` to this page. If consecutive pages have the same latency and bandwidth, i.e., they are in the same storage device, they are grouped into one SLED. During this process, the length and offset of the SLEDs are also assigned.

4.2 Library and API

The means of communication between application space and kernel space is via `ioctl` calls which return only SLEDs. This form is not directly very useful, so a library was also written that provides additional services. The library provides a routine to estimate delivery time for the entire file, and several routines to help applications order their I/O requests efficiently.

The three primary library routines for reordering I/O are `sleds_pick_init`, `sleds_pick_next_read`, and `sleds_pick_finish`. Applications first `open` the file, then call `sleds_pick_init`, which uses `ioctl` to retrieve the SLEDs from the kernel. `sleds_pick_next_read` is called repeatedly to advise the application where to read from the file next. The application then moves to the recommended position via `lseek`, and calls `read` to retrieve the next chunk of the file. The preferred size of the chunks the application wants to read is specified as an argument to `sleds_pick_init`, and `sleds_pick_next_read` will return chunks that size or smaller. The application is presumed to be following the library's advice, but it does not check. The library will return each chunk of the file exactly once.

The library checks for the lowest latency among unseen chunks, then chooses to return the chunk with the lowest file offset among those with equivalent latencies. In the simple case of a disk-based file system with a cold cache, this algorithm will degenerate to linear access of the file. As currently implemented, the SLEDs are retrieved from the kernel only when `sleds_pick_init` is called. Refreshing the state of those SLEDs occasionally would allow the library to take advantage of any changes in state caused by e.g. file prefetching.

A library routine, `sleds_total_delivery_time`, provides an estimate of the amount of time required

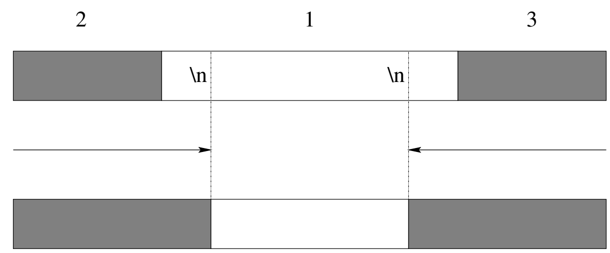

Figure 4: Adjusting SLEDs for record boundaries

to read the entire file, for applications only interested in reporting or using that value. It takes an argument, `attack_plan`, which currently can be either `SLEDS_LINEAR` or `SLEDS_BEST`, to describe the intended access pattern.

Generally, the library and application are most efficient if these accesses can be done on page boundaries. However, many applications are interested in variable-sized records, such as lines of text. An argument to `sleds_pick_init` allows the application to ask for record-oriented SLEDs, and to specify the character used to identify record boundaries.

The library prevents applications from running over the edge of a low-latency SLED and causing data to be fetched from higher-latency storage when in record-oriented mode. It does this by pulling in the edges of the SLEDs from page boundaries to record boundaries, as shown in Figure 4. The leading and trailing record fragments are pushed out to the neighboring SLEDs, which are higher latency. This requires the SLEDs library to perform some I/O itself to find the record boundaries. In the figure, the gray areas are high-latency SLEDs in a file, and the white area is a low-latency SLED. The numbers above represent the access ordering assigned by the library. The upper line is before adjustment for record boundaries, and the lower line is after adjustment for variable-sized records with a linefeed record separator.

4.3 Applications

Figure 5 shows pseudocode for an application using the SLEDs pick library. After initializing the SLEDs library, the application loops, first asking the library to select a file offset, then seeking to that location and reading the amount recommended.

Applications that have been modified to use SLEDs include the GNU versions of the Unix system utilities `wc`,

function	arguments	return value
sleds_pick_init	file descriptor, preferred buffer size	buffer size
sleds_pick_next_read	file descriptor, buffer size, record flag	read location, size
sleds_pick_finish	file descriptor	(none)
sleds_total_delivery_time	file descriptor, attack plan	estimated delivery time

Table 1: SLEDs library routines

```
int offset, nbytes, Remain;
int FileSize, fd;
char buffer[BUFSIZE];

fd = open(FileName, flags);
sleds_pick_init(fd,BUFSIZE);

for( Remain = FileSize ; Remain ;
     Remain -= nbytes ){
  nelem = MIN(Remain,BUFSIZE);
  sleds_pick_next_read(fd, &offset,
                          &nbytes);

  lseek(fd, offset, SEEK_SET);
  read(fd, buffer, nbytes);
  process_data(buffer, nbytes);
}
sleds_pick_finish(fd);
close(fd);
```

Figure 5: Application pseudocode

`grep`, and `find`, and the GNOME file manager `gmc`. These examples demonstrate the three ways in which SLEDs can be useful. The first two use SLEDs to reorder I/O operations, gaining performance and reducing total I/O operations by taking advantage of cached data first, using algorithms similar to figure 5. `find` is modified to include a predicate which allows the user to select files based on total estimated latency (either greater than or less than a specified value). This can be used to prune expensive I/O operations. `gmc` reports expected file retrieval times to the user, allowing him or her to choose whether or not to access the file.

We have also adapted two members of the LHEASOFT suite of applications, `fimhisto` and `fimgbin`, to use SLEDs. NASA's Goddard Space Flight Center supports LHEASOFT, which provides numerous utilities for the processing of images in the Flexible Image Transport System, or FITS, format used by professional astronomers. The FITS format includes image metadata, as well as the data itself.

4.4 Implementation Limitations

The current implementation provides only a basic estimate of latency based on device characteristics, with no indication of current head or rotational position. The primary information provided is a distinction between levels of the storage system, with estimates of the bandwidth and latency to retrieve data at each level. This information is effective for disk drives, but will need to be updated for tape drives. Future extensions are expected to provide more detailed mechanical estimates.

5 Results and Analysis

The benefits of SLEDs include both useful predictability in I/O execution times, and improvements in execution times for those applications which can reorder their I/O requests. In this section we discuss primarily the latter, objectively measurable aspects.

We hypothesize that reordering I/O requests according to SLEDs information will reduce the number of hard page faults, that this will translate directly to decreased execution times, and that the effort required to adopt SLEDs is reasonable. To validate these hypotheses, we measured both page faults and elapsed time for the modified applications described above, and report the number of lines of code changed.

SLEDs are expected to benefit hierarchical storage management systems, with their very high latencies, more than other types of file systems. The experiments shown here are for normal on-disk file system structures cached in primary memory. Thus, the results here can be viewed as indicative of the positive benefits of SLEDs, but gains may be much greater with HSM systems.

5.1 Experimental Setup

To measure the effect of reordering data requests, the average time and page faults taken to execute the applications with SLEDs were plotted against the values without SLEDs. We used the system `time` command to do the measurements. Tests were run on `wc` and `grep` for data files that reside on hard disk, CD-ROM, and NFS file systems. `grep` was tested in two different modes, once doing a full pass across the data file, and once terminating on the first match (using the `-q` switch). The modified LHEASOFT applications were run only against hard disk file systems.

During the test runs, no other user activity was present on the systems being measured. However, some variability in execution times is unavoidable, due to the physical nature of I/O and the somewhat random nature of page replacement algorithms and background system activity. All runs were done twelve times (representing a couple of days' execution time in total) and 90% confidence intervals calculated. The graphs show the mean and confidence intervals, though in some cases the confidence intervals are too small to see.

Data was taken for test file sizes of 8 to 128 megabytes, in multiples of eight, for most of the experiments. With a primary memory size of 64 MB, this upper bound is twice the size of primary memory and roughly three times the size of the portion of memory available to cache file pages. We expect no surprises in the range above this value, but a gradual decrease in the relative improvement.

Because SLEDs are intended to expose and take advantage of dynamic system state, all experiments were done with a warm file cache. A warm cache is the natural state of the system during use, since files that have been recently read or written are commonly accessed again within a short period of time. SLEDs provide no benefit in the case of a completely cold RAM cache for a disk-based file system. The first run to warm the cache was discarded from the result. The runs were done repeatedly in the same mode, so that, for example, the second run of `grep` without SLEDs found the file system buffer cache in the state that the first run had left it.

Tables 2 and 3 contain the device characteristics used during these experiments.

Table 4 lists the number of lines of source code modified in each application. The "src" columns are lines of code in the main application source files. The "lib" are lines of code in additional, shared, linked-in libraries. The "modified" columns are lines of code added or modified, and the "total" columns are the totals. The LHEASOFT `cfitsio` library modifications are shared, used in both `fimhisto` and `fimgbin`. The `grep` modifications are most extensive because of the need to buffer and sort output in a different fashion.

level	latency	throughput
memory	175 nsec	48 MB/s
hard disk	18 msec	9.0 MB/s
CD-ROM	130 msec	2.8 MB/s
NFS	270 msec	1.0 MB/s

Table 2: Storage levels used for measuring Unix utilities

level	latency	throughput
memory	210 nsec	87 MB/s
hard disk	16.5 msec	7.0 MB/s

Table 3: Storage levels used for measuring LHEASOFT utilities

5.2 Unix Utilities

In `gmc`, a new simple panel is added to the file properties dialog box, as shown in figure 6. This follows closely the implementation of other windows such as the "general" and "URL" properties panels. The SLEDs panel reports the length, offset, latency, and bandwidth of each SLED, as well as the estimated total delivery time for the file. Users can interactively use this panel to decide whether or not to access files; this is expected to be especially useful in HSM systems and low-bandwidth distributed systems. The same approach could be used with a web browser, if HTTP were extended to support SLEDs across the network.

application	src lines of code		lib lines of code	
	modified	total	modified	total
`grep`	560	1930	-	20K
`wc`	140	530	-	48K
`find`	70	1,600	-	23K
`gmc`	93	1,500	-	180K
`cfitsio`	-	-	190	101K
`fimhisto`	49	645	190	260K
`fimgbin`	45	870	190	260K

Table 4: Lines of code modified

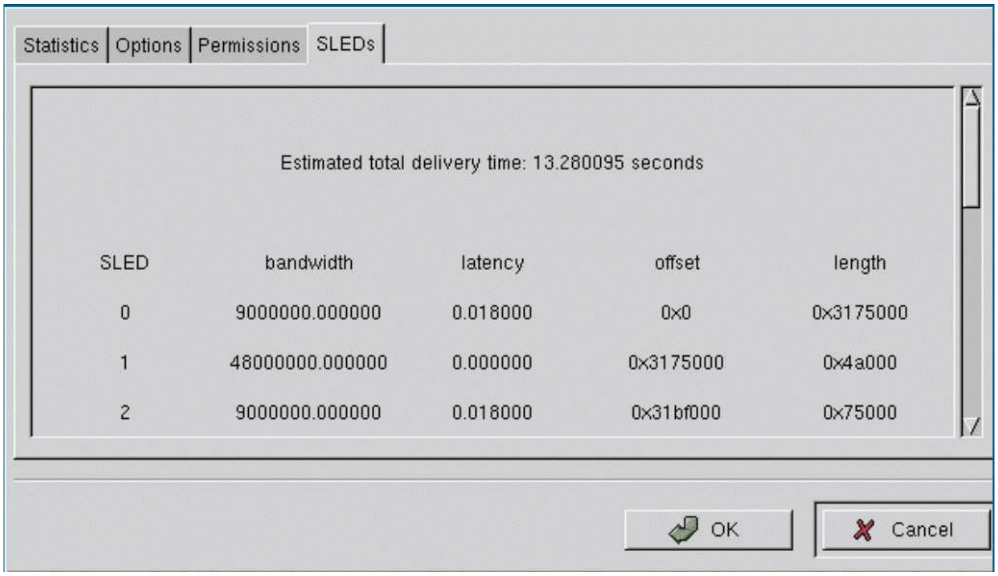

Figure 6: `gmc` file properties panel with SLEDs

The applications `wc` and `grep` implemented with SLEDs have a switch on the command that allows the user to choose whether or not to use SLEDs. If the SLEDs switch is specified, instead of accessing the data file from the beginning to the end, the application will call `sleds_pick_init`, `sleds_pick_next_read` and `sleds_pick_finish`, and use them as described in section 4.2.

For `wc`, since the order of data access is not significant, little overhead is generated in modifying the code. For applications where the order of data access is influential in code design, such as `grep`, more code changes are needed and as a result may have heavier execution overhead. In our implementation, most of the design with SLEDs is adopted from the one without SLEDs. However, unless the user chooses not to output the matches, the result will have to be output to `stdout` in the order that they appear in the file. To deal with this, we have to store a match in a linked list when traversing the data file in the order recommended by SLEDs. We sort the matches in the end by their offset in the file and then dump them to `stdout`. As a result, switches such as `-A`, `-B`, `-b`, and `-n` had to be reimplemented.

Consider searching a large source tree, such as the Linux kernel. Programmers may do `find -exec grep` (which runs the `grep` text search utility for every file in a directory tree that matches certain criteria, such as ending in `.c`) while looking for a particular routine. If the routine is near the end of the set of files as normally scanned by `find`, or if the user types control-C after seeing what he wants to see, the entry may be cached but earlier files may already have been flushed. Repeating the operation, then, causes a complete rescan and fetch from high-latency storage. The first author often does exactly this, and the SLEDs-aware `find` allows him to search cache first, then higher latency data only as needed.

Standard `find` provides a switch that stops it from crossing mount points as it searches directory trees. This is useful to, for example, prevent `find` from running on NFS-mounted partitions, where it can overload a server and impair response time for all clients. On HSM systems, users may wish to ignore all tape-resident data, or to read data from a tape currently mounted on a drive, but ignore those that would require mounting a new tape. In wide-area systems, users may wish to ignore large files that must come across the network.

In our modified `find`, the user can choose to find files that have a total delivery time of less than, equal to, or greater than a certain time. `find -latency +n` looks for files with more than n seconds total retrieval time, `n` means exactly n seconds and `-n` means less than n seconds. `mn` or `Mn` instead of `n` can be used for units of milliseconds, and `un` or `Un` used for microseconds. The SLEDs library routine `sleds_total_delivery_time` was used for this comparison. Only 2 extra routines (less than 100 lines of code) were needed to add SLEDs to `find` and all functionality has been kept the same. These two extra routines work and were implemented similarly to other

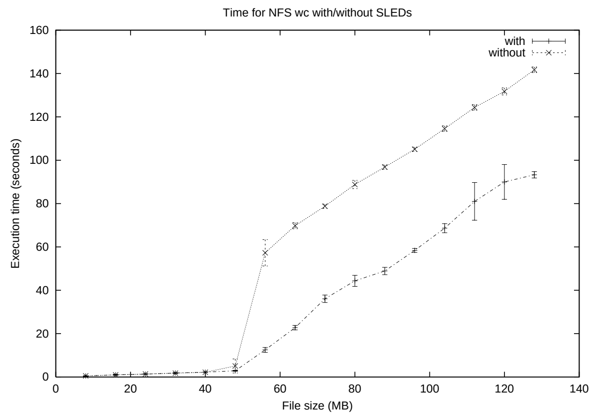

Figure 7: `wc` times over NFS, with and without SLEDs, warm cache. The legends on the graphs are correct, but somewhat difficult to follow; look first for the plus signs and Xes. The dashed and solid lines in the legend refer to the error bars, not the data lines.

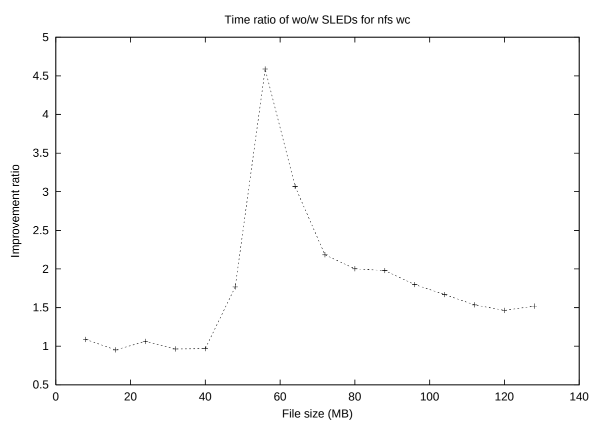

Figure 8: `wc` time ratios (speedup) over NFS, with and without SLEDs, warm cache

predicates such as `-atime`.

Figure 7 presents the execution times for `wc` against file size on an NFS file system with and without SLEDs. As to be expected, SLEDs starts to show an advantage at file sizes over about 50MB as the cache becomes unable to hold the entire file. The difference in execution time remains about constant afterwards since the average usage of cached data by SLEDs is expected to be determined by the cache size, which is constant. As a result, we have the best percentage gain at around 60MB. We also noticed a very consistent performance gain, as shown by the small error bars in the plot. Figure 8 is the ratio of the two curves in Figure 7. The execution time without SLEDs is divided by the execution time with SLEDs, providing a speedup number. As we can see, this ratio

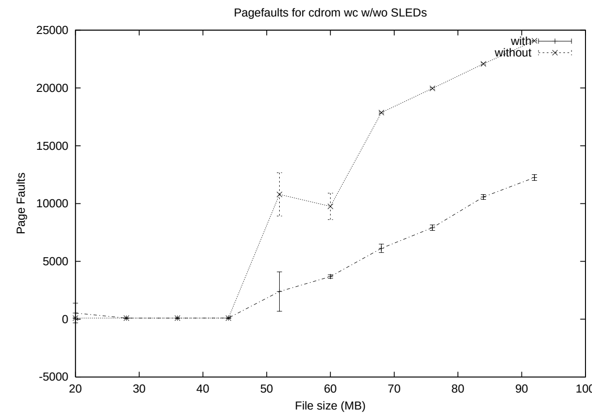

Figure 9: `wc` page faults on CD-ROM, with and without SLEDs, warm cache

peaks at around 60MB at a value of as large as 4.5, and can be comfortably considered to be a 50% or better improvement over a broad range.

Figure 9 plots the pagefaults for `wc` against file size on cdrom with and without SLEDs. As to be expected, this result shows a close correlation with the execution time. Without SLEDs, both the execution time and pagefaults increase sharply. With SLEDs, the increase in both is gradual.

Figure 10 plots the execution time for `grep` for all matches against file size on cdrom with and without SLEDs. Although there is a small amount of overhead for small files, `grep` also demonstrated a very favorable gain of about 15 seconds for CD-ROM for large files. This can be interpreted as the time taken to fill the file cache from CD when SLEDs are not used, as the application derives essentially no benefit from the cached data in this case.

Because this approach requires buffering matches before output, if the percentage of matches is large, performance can be hurt by causing additional paging. All experiments presented here are for small match percentages (kilobytes out of megabytes) output in the correct order.

The increase in execution time for small files is all CPU time. This is due to the additional complexity of record management with SLEDs, and to more data copying. We used `read()`, rather than `mmap()`, which does not copy the data to meet application alignment criteria. An `mmap`-friendly SLEDs library is feasible, which should reduce the CPU penalty. The increase appears large in percentage terms, but is a small absolute value. Regard-

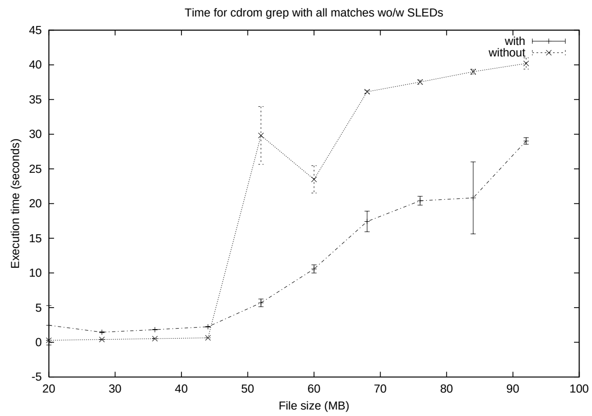

Figure 10: Execution time of `grep` for all matches, CD-ROM, warm cache

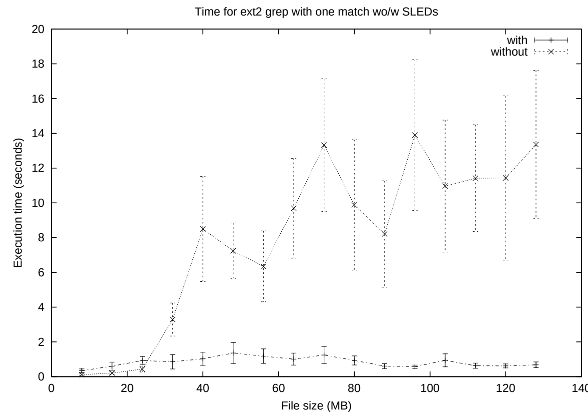

Figure 11: Execution time of `grep` for one match, ext2 FS, warm cache

less, one of the premises of SLEDs is that modest CPU increases are an acceptable price to pay for reduced I/O loads.

Figure 11 plots the execution time for `grep` for the first match against file size on an ext2 file system (local hard disk) with and without SLEDs, for a single match that was placed randomly in the test file. The first match termination, if it finds a hit on cached data, can run without executing any physical I/O at all. Because the application reads all cached data first when using SLEDs, it has a higher probability of terminating before doing any physical I/O. The non-SLEDs run is often forced to do lots of I/O because it reads from the beginning of the file rather than reading cached data first, regardless of location. Quite dramatic speedups can therefore occur when using SLEDs, relative to a non-SLEDs run. This is the ideal benchmark for SLEDs in terms of individual application performance.

The execution time ratio for `grep` with the first match against file size on the ext2 file system, with and without SLEDs, is shown in Figure 12. In addition, we have computed the cumulative distribution function (CDF) for `grep` for the first match on an NFS file system with and without SLEDs, as shown in Figure 13.

The large error bars in Figure 11 for the case without SLEDs are indicative of high variability in the execution time caused by poor cache performance. The cumulative distribution function for execution times shown in Figure 13 suggests that `grep` without SLEDs gained essentially no benefit from the fact that a majority of the test file is cached.

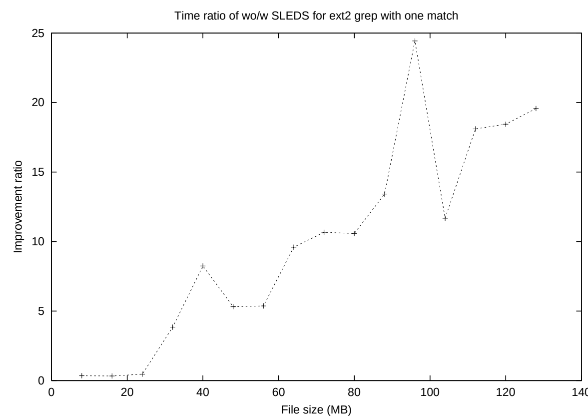

Figure 12: Ratio of mean execution time (speedup) of `grep` for one match, ext2 FS, warm cache, with and without SLEDs

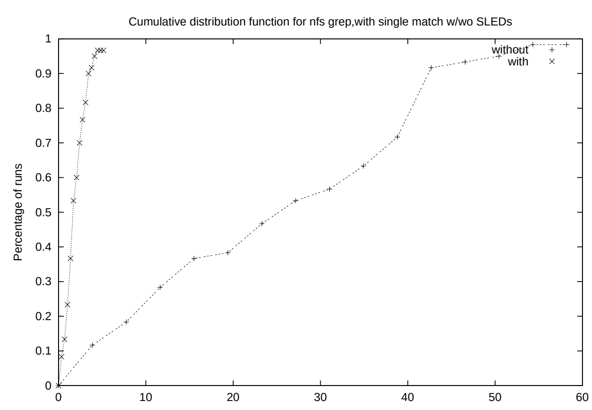

Figure 13: Cumulative distribution of execution time of `grep` for one match, NFS, warm cache, for a 64MB file

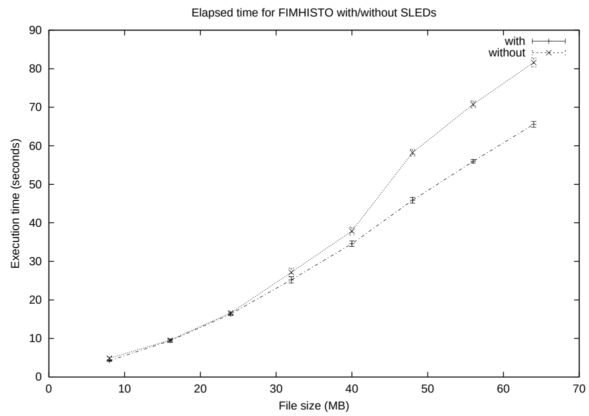

Figure 14: Elapsed time for `fimhisto`, ext2 FS, warm cache

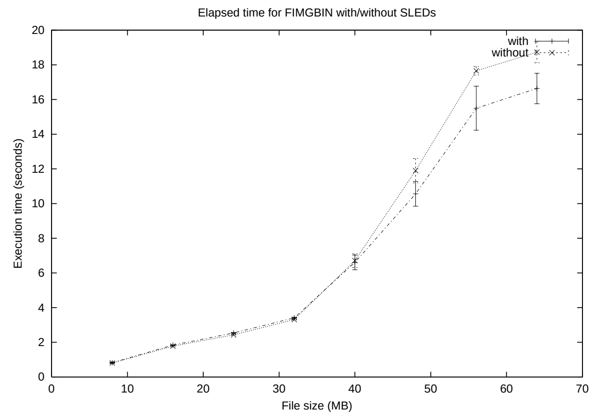

Figure 15: Elapsed time for `fimgbin`, ext2 FS, warm cache, 4x data reduction

5.3 LHEASOFT

`fimhisto` copies an input data image file to an output file and appends an additional data column containing a histogram of the pixel values. It is implemented in three passes. The first pass copies the main data unit without any processing. The second pass reads the data again (including performing a data format conversion, if necessary) to prepare for binning the data into the histogram. The third pass performs the actual binning operation, then appends the histogram to the output file. This three-pass algorithm resulted in observed cache behavior like that shown in Figure 3.

We adapted `fimhisto` to use SLEDs in the second and third passes over the data, reordering the pixels read and processed to take advantage of cached data. We implemented an additional library for LHEASOFT that allows applications to access SLEDs in units of data elements (usually floating point numbers), rather than bytes; the calls are the same, with `ff_` prepended. Tests were performed only on an ext2 file system, and only for file sizes up to 64 MB.

`fimhisto` showed somewhat lower gains than `wc` and `grep`, due to the complexity of the application, but still provided a 15-25% reduction in elapsed time and 30-50% reduction in page faults on files of 48 to 64 MB. Figure 14 shows the familiar pattern of SLEDs offering a benefit above roughly the file system buffer cache size. `fimhisto`'s I/O workload is one fourth writes, which SLEDs does not benefit, and includes data format conversion as well. These factors contribute to the difference in performance gains compared to the above applications.

We modified `fimgbin` to reorder the reads on its input file according to SLEDs. `fimgbin` rebins an image with a rectangular boxcar filter. The amount of data written is smaller than the input by a fixed factor, typically four or 16. It can also be considered representative of utilities that combine multiple input images to create a single output image. The main `fimgbin` code is in Fortran, so we added Fortran bindings for the principal SLEDs library functions.

Figure 15 shows elapsed time for `fimgbin`. It shows an eleven percent reduction in elapsed time with SLEDs for a data reduction factor of four on file sizes of 48MB or more. This is smaller than the benefit to `fimhisto`, despite similar reductions of 30-50% in pagefaults. We believe this is due to differences in the write path of the array-based code, which is substantially more complex and does more internal buffering, partially defeating our attempts to fully order I/Os. For a data reduction factor of 16, the elapsed time gains were 25-35% over the same range, indicating that the write traffic is an important factor. [1]

6 Future Work

The biggest areas of future work are increasing the range of applications that use SLEDs, improving the accuracy of SLEDs both in mechanical details and dynamic system load, and the communication of SLEDs among components of the system (including between file servers and clients). The limitations discussed in section 3.4

[1] A bug in the SLEDs implementation currently limits the rebinning parameters which operate correctly.

need to be addressed.

Devices can be characterized either externally or internally. Hillyer and Sandstå did external device characterization on tape drives, and we have done so on zoned disk drives [Van97]. The devices or subsystems could be engineered to report their own performance characteristics. Cooperation of subsystem vendors will be required to report SLEDs to the file system. Without this data, building true QoS-capable storage systems out of complex components such as HP's AutoRAID [WGSS95] will be difficult, whether done with or without SLEDs.

Work has begun on a migrating hierarchical storage management system for Linux [Sch00]. This will provide an excellent platform for continued development of SLEDs.

7 Conclusion

This paper has shown that Storage Latency Estimation Descriptors, or SLEDs, provide significant improvements by allowing applications to take advantage of the dynamic state of the multiple levels of file system caching. Applications may report expected file retrieval time, prune I/Os to avoid expensive operations, or reorder I/Os to utilize cached data effectively.

Reporting latency, as we have done with `gmc`, is useful for interactive applications to provide the user with more insight into the behavior of the system. This can be expected to improve user satisfaction with system performance, as well as reduce the amount of time users actually spend idle. When users are told how long it will take to retrieve needed data, they can decide whether or not to wait, or productively multitask while waiting.

Pruning I/Os is especially important in heavily loaded systems, and for applications such as `find` that can impose heavy loads. This is useful for network file systems and hierarchical storage management systems, where retrieval times may be high and impact on other users is a significant concern. Because the SLEDs interface is independent of the file system and physical device structure, users do not need to be aware of mount points, volume managers, or HSM organization. Scripts and other utilities built around this concept will remain useful even as storage systems continue to evolve.

Reordering I/Os has been shown, through a series of experiments on `wc` and `grep`, to provide improvement in execution time of from 50 percent to more than an order of magnitude for file sizes of one to three times the size of the file system buffer cache. Experiments showed an 11-25 percent reduction in elapsed time for `fimhisto` and `fimgbin`, members of a large suite of professional astronomical image processing software. SLEDs-enabled applications have more stable performance in this area as well, showing a gradual decline in performance compared to the sudden step phenomenon at just above the file system buffer cache size exhibited without SLEDs. These experiments were run on ext2, NFS and CD-ROM file systems; the effects are expected to be much more pronounced on hierarchical storage management systems.

Availability and Acknowledgements

The authors thank USC/ISI's Joe Bannister, Ted Faber, John Heidemann, Steve Hotz and Joe Touch for their constructive criticism and support. Quantum's Daniel Stodolsky, as a SLEDs skeptic, and Rick Carlson, as a supporter, both deserve thanks for sharpening the arguments for and against over the last two years. Nokia's Tom Kroeger provided useful feedback on the paper, as well as help with the figures. Finally, our shepherd, Frans Kaashoek, contributed significantly to the structure and clarity of the paper.

Much of the implementation and testing presented here were done by the second author as an intern at Quantum in the Storage Systems Software Unit. Quantum's Hugo Patterson and Tex Schenkkan supported this work.

The source code for SLEDs and the modified applications are available from the first author.

References

[ADN+95] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 109–126. ACM, December 1995.

[AOG92] David P. Anderson, Yoshitomo Osawa, and Ramesh Govindan. A file system for continuous media. *Trans. on Computing Systems*, 10(4):311–337, November 1992.

[BFD97] William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed schedule management in the Tiger video fileserver. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 212–223. ACM, October 1997.

[CFKL96] Pei Cao, Edward E. Felten, Anna R. Karlin, and Kai Li. Implemention and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[Dig76] Digital Equipment Corporation. *decsystem10 Monitor Calls*, June 1976.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proc. ACM Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–170. ACM, October 1996.

[GA95] James Griffioen and Randy Appleton. Performance measurements of automatic prefetching. In *ProceedingsInternational Conference on Parallel and Distributed Computing Systems*, September 1995.

[HP89] Robert L. Henderson and Alan Poston. MSS-II and RASH: A mainframe UNIX based mass storage system with a rapid access storage hierarchy file management system. In *Proc. 1989 USENIX Winter Technical Conference*, pages 65–84. USENIX, 1989.

[HS96a] Bruce Hillyer and Avi Silberschatz. On the modeling and performance characteristics of a serpentine tape drive. In *Proc. ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 170–179. ACM, May 1996.

[HS96b] Bruce K. Hillyer and Avi Silberschatz. Random I/O scheduling in online tertiary storage systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 195–204. ACM, June 1996.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[Kro00] Thomas M. Kroeger. *Modeling File Access Patterns to Improve Caching Performance*. PhD thesis, University of California Santa Cruz, March 2000.

[LLJR99] Per Lysne, Gary Lee, Lynn Jones, and Mark Roschke. HPSS at Los Alamos: Experiences and analysis. In *Proc. Sixteenth IEEE Symposium on Mass Storage Systems in cooperation with the Seventh NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 150–157. IEEE, March 1999.

[LT96] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proc. ACM Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92. ACM, October 1996.

[MS96] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 279–294. USENIX, January 1996.

[NAS00] NASA. Lheasoft. Goddard Space Flight Center Laboratory for High Energy Astrophysics, http://rxte.gsfc.nasa.gov/lheasoft/, April 2000.

[NSN+97] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 276–287. ACM, October 1997.

[PA94] Joseph Pasquale and Eric Anderson. Container shipping: Operating system support for I/O-intensive applications. *IEEE Computer*, 27(3):84–93, March 1994.

[PGG+95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching.

In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 79–95. ACM, December 1995.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[Sat90] Mahadev Satyanarayanan. Scalable, secure, and high available distributed file access. *IEEE Computer*, pages 9–21, May 1990.

[Sch00] Marc Schaefer. The migration filesystem. web page, April 2000. http://www-internal.alphanet.ch/˜schaefer/mfs.html.

[SDH+96] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proc. 1996 USENIX Technical Conference*, pages 1–14. USENIX, January 1996.

[Shi98] Jamie Shiers. Building a database for the large hadron collider (LHC): the exabyte challenge. In Ben Kobler, editor, *Proc. Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in Cooperation with Fifteenth IEEE Symposium on Mass Storage Systems*, pages 385–396, March 1998.

[SM99] Olav Sandstå and Roger Midstraum. Low-cost access time model for serpentine tape drive. In *Proc. Sixteenth IEEE Symposium on Mass Storage Systems in cooperation with the Seventh NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 116–127. IEEE, March 1999.

[Ste97] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 252–263. ACM, October 1997.

[Van97] Rodney Van Meter. Observing the effects of multi-zone disks. In *Proc. USENIX '97 Technical Conference*, pages 19–30. USENIX, January 1997.

[Van98] Rodney Van Meter. SLEDs: Storage latency estimation descriptors. In Ben Kobler, editor, *Proc. Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in Cooperation with Fifteenth IEEE Symposium on Mass Storage Systems*, pages 249–260, March 1998.

[vI99] Catherine van Ingen. Storage in Windows 2000. presentation, February 1999.

[WC95] R. W. Watson and R. A. Coyne. The parallel I/O architecture of the high-performance storage system (HPSS). In *Proc. Fourteenth IEEE Symposium on Mass Storage Systems*, pages 27–44. IEEE, September 1995.

[WGP94] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling for modern disk drives and non-random workloads. Technical Report CSE-TR-194-94, University of Michigan, March 1994.

[WGSS95] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 96–108. ACM, December 1995.