Scalable Storage for Digital Libraries[*]

Paul Mather

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
USA

23 October 2001

Abstract

I propose a storage system optimised for digital libraries. Its key features are its heterogeneous scalability; its integration and exploitation of rich semantic metadata associated with digital objects; its use of a name space; and its aggressive performance optimisation in the digital library domain.

Contents

[*]This material was originally submitted as the Ph.D. preliminary exam document of the author. It is reproduced here without modification.

1 Introduction

More than ever before, there is a large amount of information available in digital form. Increasingly, this content is stored in large repositories accessed over wide-area networks. With the dramatic rise of digital libraries, electronic commerce and the everyday use of networked information sources in business has also come a pressing need for *high availability* of network-accessible data. In the information age, data are the lifeblood of many organisations, and the constant availability of those data is paramount.

To address the voracious appetite for storage, research in that arena recently has turned towards the issue of *scalability*—in terms of hardware, software, and administration. Moreover, a current focus has turned towards the use of commodity off-the-shelf (COTS) hardware as a means of achieving scalability. In particular, approaches such as *clustering*, *networked attached storage*, and *storage area networks* are fast becoming industry standards as means for achieving manageable, scalable storage on the order of terabytes (2^{40} bytes) and upwards.

The bulk of storage research at the operating system support level has looked at providing file system support and associated functions. Although there have been proposed some architectures for digital libraries storage repositories, with the Kahn-Wilensky architecture [88] finding favour, all digital library implementations thus far have been built atop traditional operating system file systems or databases (which are themselves usually built atop traditional operating system file systems). Such file systems are relatively weak in terms of providing support for the rich metadata and file types typically used in digital libraries.

Traditional file system metadata is geared towards providing the operating system logical and physical location of file data, and simple user protection mechanisms. It is typically canonical, and relatively crude compared with metadata for digital libraries, which provides multiple viewpoints of digital objects, and potentially sophisticated rights management. In addition, semantically different object (file) types might require radically different storage and access techniques. In traditional file systems, this often must be handled in an *ad hoc* fashion, often by means of arbitrary partitioning of storage and use of specialised device drivers.

Here, we aim to reconcile diverse strands of scalable storage design with digital library repository access semantics as the target. Particular emphasis will be placed on *scalability* in the storage realm, and on metadata support and system support for different object types in the digital libraries realm.

In Section 2 is background motivation. Section 3 contains an overview of related research in scalable storage. A brief guide to relevant work in digital library object repositories and access support is presented in Section 4. Section 5 contains an outline of the problem to be addressed in this research. A plan of work is given in Section 6.

2 Background Motivation

More data are available online than ever before. There is a dramatic increase in the amount available to the public served over wide-area networks, and the level of growth continues. In some areas, data gathering has followed approximately Moore's law: in astronomy, for example, the volume of data gathered is doubling roughly every 20 months [188].

Already, there are several terabyte-sized repositories of information of various kinds accessible to millions of users worldwide, and more are being proposed.

The *TerraServer* [13] provides aerial, satellite, and topographic images of Earth delivered via the World Wide Web (WWW). There are eight servers—six WWW servers and two database back-end servers—averaging 5–8 million hits and 50 GB of image downloads per day. As of February 2000, the database comprised over 1.5 terabytes of data. The two database back-end servers can contain up to 3.2 TB and 1.2 TB of data stored using RAID-5 on 324 9 GB Ultra Small Computer Systems Interface (SCSI) and 140 9 GB FiberChannel disks respectively. The system is configured to handle a maximum of 40 million hits per day, and 6,000 simultaneous users.

The Zoom Project [10, 190] provides access to over 70,000 images of the Fine Arts Museums of San Francisco over the WWW. The images comprise approximately 2.5 TB of storage served from a cluster of 20 storage nodes hosting a total of 396 8.4 GB Ultra-Wide SCSI disks. The storage cluster is called *Tertiary Disk* [189]. Four of the storage nodes are "disk heavy," with 70 disks per 2 nodes; the remaining 16 are "CPU-heavy," with 32 disks per 2 nodes. The storage nodes are networked via a 100 Mbps Ethernet local area network (LAN), and the cluster is connected to the Internet via an asynchronous transfer mode (ATM) network connection. Strings of disks are double-ended for added reliability, meaning that even if one storage node (or the SCSI host adapter) connected to the string fails, the other node can continue to access the disks.

The Sloan Digital Sky Survey (SDSS) intends to provide multi-terabyte astronomical observational data for interactive exploration and query via the Internet [188]. The SDSS expects to collect over 40 terabytes of raw data in the next five years. The proposed architecture will include a 20 node array aggregating 6 TB of database storage and amassing a cumulative 20 billions of instructions per second (BIPS) of processing power to scan continually the SDSS dataset to apply user-supplied query predicates. It is estimated that a scan rate of 2 GBps per node across the array can be achieved, allowing the entire dataset to be processed about every two minutes.

Even larger datasets are being planned. The NASA Earth Observing System Data Information System (EOSDIS) project [55, 93] seeks to provide access to a huge collection of remote sensing data collected over a 15 year time-span. By 2002, it is predicted that raw data will be collected at a rate of over 360 GB per day, and that the archive will contain over 3 petabytes (2^{50} bytes) of data across more than 260 data products. The system will utilise Distributed Active Archive Centers (DAACs) networked together via a high-speed backbone to collect and process the data, and the entire collection will be accessible to users via the WWW using a browser.

Even larger than EOSDIS is the data storage needs of the Large Hadron Collider (LHC) project at CERN [175]. This particle accelerator will enter operation in 2005, and experiments run on it will generate some 5 petabytes of data per year, with data rates in the region of 100 MB to 1.5 GB per second. Experiments will run over 15 years, generating in excess of 100 petabytes. The engineering challenge, however, tackles storage requirements in the exabyte (2^{60} bytes) range.

Along with such large scale projects, ordinary production and consumption of data continues at a pace that has even *outpaced* Moore's Law. Decision support systems (DSS) are rabid consumers of data, with demand doubling roughly every 9–12 months, in contrast to the approximately 18-month doubling period of Moore's Law [91]. Indeed, the largest DSS database reported in 1997 almost doubled in size from 2.4 TB to 4.4 TB in 1998 alone [205]. The biggest DSS database of 1998 more than tripled from 1.3 TB to 4.6 TB over its previous year's size [206].

The huge rate of increase in data suggests that massive-scale storage architectures need to be able to scale at a *faster* rate than the growth of processors and disks themselves: we cannot rely on hardware improvements alone to keep pace with the demand for data storage.

3 Scalable Storage

Computer storage has long been the subject of research. With the rise of parallel and distributed systems came a plethora of associated storage systems. For the purposes of this research, this activity can broadly be grouped in the areas of local file systems; parallel file systems; distributed file systems; and scalable networked storage. Within all these areas, attention has been paid to operating system and application programming support; layout, buffering, caching, and efficiency; reliability, fault-tolerance, and crash recovery; and security. Some of these goals are contradictory: high fault tolerance might require a lower efficiency, for example.

3.1 Local File Systems

A *file system* is a means by which data is managed on secondary storage. At the user level, these data are organised using an abstraction known as a *file*. Files are a logical structuring mechanism that allows data to be organised as a byte-addressable sequence that may be randomly accessed. File systems mask the details of how this logical structure is stored on actual physical devices attached to the system. File systems not only manage the actual long-term storage of data—layout and structuring—but also manage the access to that data—name resolution, buffering and caching. File systems employ metadata to assist in the organisation of files on storage. Examples of such metadata might be pointers locating blocks belonging to a given file on a disk and the length of that file, or a bitmap indicating the location of unused blocks on a disk.

We deem *local file systems* to be those whose storage is provided by directly-attached secondary storage. Most often, this storage is in the form of direct-access storage such as disk drives. The improvement in hard disk technology in terms of total storage (areal density) and transfer rate has improved roughly according to Moore's Law [73, 74]. However, rotational speeds and, especially, seek times have increased at a much slower rate. Head seek times, in particular, are limited by the physics of inertia, and may not see vast improvement. To address this, file system research has explored the use of alternative layout schemes [145], aggregation [31, 67], and aggressive caching and buffering [62, 148] amongst other techniques to ameliorate the effects of head seek times during reads to and writes from disk.

3.2 Workload Studies

There have been several studies of the content and usage of local file systems [49, 70, 144, 156, 200]. Such studies characterise the *workload* of the file system. Workloads provide important information to file system architects, and can guide the design of layout schemes; buffering and caching strategies; migration policies; and performance tuning.

One consistent feature to arise out of workload studies for typical engineering and office applications and environments is that most files are small: in Gibson and Miller's study [70] the majority of files are less than 4K in size, and in Douceur and Bolonsky [49] the median file size is 4K. The distribution of file sizes is heavy-tailed, with most bytes being concentrated in relatively few large files: most files are small, but large files use most disk capacity. Furthermore, a general trend is that most files are not accessed very often in the long term.

Traces of long-term file activity collected by Gibson and Miller [70] reveal that on a typical day, less than 3% of all files are used, and of that usage, file accesses account for over twice the number of either file creations or deletions, and the number of files modified is one third the number of creations or deletions. In addition, they found that files modified are about as likely to remain the same size as they are to grow, and that modified files rarely get smaller [70]. In fact, the amount of file growth during file modifications found in their study was usually less than 1K, regardless of the file's size [70]. Long-term activity of the file systems surveyed by Douceur and Bolonsky [49] determined the median file age to be 48 days, where file age is calculated as the time passed since the most recent of either the file's creation or last modification relative to when the file system contents snapshot was taken.

Traces capturing the short-term behaviour of file system workloads show that many files and accesses are ephemeral and bursty. The seminal Unix 4.2 BSD study of Ousterhout *et al.* [144] determined that files are almost always processed sequentially, and that more than 66% are whole-file transfers. Of the remaining non-whole file transfers, most read sequentially after initially performing an *lseek*, i.e., to append. When measured over a short time scale, files tend to be very short-lived, or only open for a very short time.

Subsequent studies have confirmed the results of Ousterhout *et al*, but tending towards extremes. Baker *et al.* [12] discovered the time between a file being created and subsequently either deleted or truncated to zero length to be less than 30 seconds for 65%–80% of all files, and those files tend to be small, accounting for only about 4%–27% of all new bytes deleted or overwritten within 30 seconds. Vogels [200] found that files are open for even shorter periods: 75% are open for less than 10 milliseconds, and 55% of new files are deleted within 5 seconds; 26% are overwritten within 4 milliseconds. Similarly, both Baker *et al.* [12] and Vogels [200] confirm that most files accessed are short, and sequentially accessed (but shifting more towards random access), but that most bytes transferred belong to large files, and that large files are getting larger (20% are 4 MB or larger).

Ramakrishnan *et al.* [156] collected relatively short-term traces of file system activity at eight different customer sites running VAX/VMS over several different time periods at various times of the day and night. The sites represented different workloads (interactive; timesharing and database activity; transaction processing; and batch data processing), but analysis showed that file system activity was consistent across different workload environments. The study revealed that for almost all workloads, most files—over 80%—are inactive (not opened, read, or written during the trace), and of those active files, a small percentage account for most of the file opens (in an airline transaction processing workload, less than 3% of active files account for over 80% of file opens). Over all workloads, over 50% of all active files were opened only once or twice, and 90% of active files were opened less than 10 times in a typical 9–12 hour "prime time" period.

Of those files active during the trace period, over 50% of them were read only once or twice, and 90% of active files in all but one workload had between 24 and 46 reads. In the case of writes, 31–46% of active files were not written at all, and 14–28% had only one write. Over 50% of the active files had fewer than two write operations. Once again, a small percentage of files often account for most writes, and skew the mean number of writes upwards. For the timesharing workloads studied, 1.3–2% of files accounted for 80–87% of writes. In the case of transaction processing workloads, 1.3–1.8% of files accounted for 95% of writes.

Studies that have tracked the semantic content of files have found there is a good correlation between file type and file size [18, 49, 165].

Analysis of the WWW by Adamic and Huberman [2] reveals that files and sites there follow a power law with respect to many attributes, including file size, site size, popularity, and site linkage, thus confirming in a wider context the "heavy-tailed distribution" phenomenon common to all file system studies.

Some consequences of workload studies are that the *name space* is dominated by small files, but that transfer bandwidth is dominated by large files. Furthermore, files exhibit a *generational* behaviour in that if they survive the short term then they will tend to persist—largely inactive—for the long term. The likely uncertain early lifetime makes write-back caching attractive in the anticipation that a file will subsequently be deleted or modified before actually being written to disk. The more of this volatility that can be smoothed out by caching or buffering the better.

3.2.1 Workload Characterisation and Modelling

Workload characterisation has long been used in assessing the performance of systems [27]. In storage research, analysis of workload traces, as described above, features prominently. These workload traces capture I/O activity as it happens over a period of time, and contain important information about access requests and access patterns.

Because actual traces are often large and difficult to capture (because of technical and political reasons), researchers often have to resort to using synthetic traces to model workload. Generating synthetic traces that are representative of real-world activity is a significant problem [60]. Ganger and Patt [61] advocate a system-level approach to representing I/O workloads, and, in particular, that different classes of I/O request be treated with different importance to better model their effect on overall system performance. SynRGen [54] is a tool for generating workloads based upon algorithmic specifications of tasks, offering more realistic artificial traces of system activity.

Traces are primarily used as input to simulations of storage systems. Simulation is popular because of the flexibility with which data can be collected and operational parameters controlled. This allows "what-if" scenarios to be investigated to examine the effect of specific parameters within the storage system on global system performance, e.g., cache size, seek time, block size, bus speed, etc.

Ruemmler and Wilkes [161] describe a detailed disk simulation, which they apply to the HP C2200A and HP 97560 disk drives. Worthington *et al.* [207] describe the extraction of disk drive parameters directly by interrogation of SCSI drives, combined with empirical measurements, thereby improving modeling accuracy. Ganger *et al.* [63] produced a highly-configurable disk simulator called DiskSim. DiskSim can model such storage aspects as device drivers, busses, controllers, adapters, and disk drives. DIXtrac [168] improves upon the extraction capabilities of [207] in that it can extract over 100 disk parameters in a fully automated fashion. DIXtrac can be used to provide input for disk drives simulated by DiskSim. Pantheon [202] is a toolkit that supports the development of storage simulations. It has been successfully used in such storage projects as TickerTAIP [29] and AutoRAID [203].

Although simulation is prevalent in storage research, it has the disadvantage of being costly in terms of time and resources needed to run simulations. Analytical modeling has the advantage of being a quicker and more direct answer for exploring alternative design spaces. The disadvantage of analytical modeling is its inaccuracy compared to simulation.

Uysal *et al.* [196] present an analytical model to predict throughput of a modern disk array that captures many of the complex optimisations found in such hardware. They validate their model against a real commercial array and obtain prediction accuracy within 32% in most cases, and to within 15% on average. Shriver [177] presents a detailed analytical model of a single disk drive. Barve *et al.* [15] extend this by analysing the performance of several drives on the same SCSI bus, where bus contention becomes a factor when large request sizes are used. Shriver *et al.* [178] present an analytical model of disk drives that support read-ahead and request reordering. Their prediction accuracy is to within 17% across a variety of disk drives and workloads. Borowsky *et al.* [24] present an analytical model of response time of a device subject to phased and correlated workloads. Their model is designed to determine whether a given quality of service bound is satisfied for the workloads presented.

Because analytical models lend themselves to direct computation, unlike long-running simulations, this makes them ideal for use in storage optimisation problems. Of recent attention, in particular, is the area of attribute-managed storage [72]. Attribute-managed storage is a constrained optimisation problem in which a set of workload units and devices are presented as input to a solver. The output is a mapping of workload units to devices such that the needs of the workload units are met. Shriver [176] describes the workload and device attributes and their mapping in greater detail. Alvarez *et al.* [4] describe a suite of tools called Minerva that can be used to design storage systems automatically. In a test, Minerva is able successfully to design a storage system to handle a decision-support workload that performs as well as a human-designed configuration. The Minerva-generated system used fewer disks than the human-designed system (16 vs. 30), and took only 10–13 minutes to produce the design. Minerva is limited by its analytical models to the types of storage system it can design (currently, disk arrays).

3.3 Local File System Performance Issues

As mentioned previously, seek times dominate disk access times. A simple "break-even" point to make a seek worthwhile would be to spend as much time actually transferring data as seeking to find it. Obviously, though, the smaller the overall fraction of a total read or write is occupied by seeking the better. Even COTS hard drives have relatively high sustained transfer rates, making the time lost seeking even worse. For example, a "desktop use" 25 GB EIDE drive with an average seek time of 9 ms supports a sustained transfer rate of 15.5 MB/s [86]. A "server" 36 GB Ultra 160 SCSI drive with an average seek time of 4.9 ms supports a sustained transfer rate of 36 MB/s [87]. For the 25 GB drive, just over 142 KB of data could be transferred in the time spent in an average seek (0.009 s at 15.5 MB/s); for the 36 GB drive, just over 180 KB could be transferred (0.0049 s at 36 MB/s). If only, say, 1 KB were transferred before seeking elsewhere, then less than 1% of the total time would be spent actually transferring useful data using the above mentioned drives as examples. Even for a larger block size—4 KB—still less than 3% of the time is spent usefully transferring data. Successful use of the available disk bandwidth means making fewer seeks and larger transfers.

3.3.1 Caching

The impact of seeking on disk I/O is often addressed by caching. A large RAM-based cache can absorb the effects of multiple reads of the same disk block over time, and most modern operating systems employ such buffer caches. However, caches depend upon *locality of reference* to be successful: what happened before will happen again. As mentioned previously, workload studies indicate most reads are whole-file sequential reads. These have a negative impact on cache performance, as they can effectively stream through and flush the cache for large enough files, if action is not taken to mitigate such behaviour. Baker *et al.* report cache read miss rates of about 40% in their study, rising to 97% on machines processing large files [12].

Caches will cache the most commonly accessed disk blocks. In an engineering and office environment, the most commonly-used files will likely be application and system programs: word processors, compilers, editors, and similar. Across the entire user population, individual user data files will be accessed less often relative to the software that operates upon those data. Caching will likely be more successful for programs than data. The exception to this is perhaps the caching of writes.

A disk block that is read, modified, and written back will exhibit strong temporal locality of reference. Unfortunately, out of concern for safety, many applications will not update file data *in situ*, but, instead, often will make a temporary copy of the old file to which are applied the updates. After processing, the original file is either deleted or renamed as a backup copy, and the copy is renamed to the original. Baker *et al.* [12] report that only 10% of new data written is deleted or overwritten in the cache rather than actually being written to disk. However, they also report that the 30-second dirty block cleaning rule in their system accounted for over 70% of blocks written from the cache, indicating that cache content safety (flushing dirty blocks to disk to guard against loss due to hardware failure), not cache size, is a major factor on write-back cache performance.

3.3.2 Clustering and Fragmentation

Another method of ameliorating the effects of seeks is to co-locate related data physically close together on the disk. In this way, only small seeks or a rotational delay must be incurred to read successive blocks of a file. Physical clustering is especially useful in the case of file system metadata. The Fast File System (FFS) for Unix greatly improved the performance of the previous file system design by spreading file metadata throughout the disk [126]. The new design divided the disk into cylinder groups, with each cylinder group having its own local metadata. In the previous design, *all* file system metadata was clustered at the beginning of the disk. Data blocks located on tracks away from the start of the disk thus incurred long seeks every time the metadata for the file stored there was accessed. McKusick [126] estimates that such a segregation of metadata, coupled with the small 512-byte block size, meant that the old Unix file system design was only able to use 3-5% of the disk bandwidth. But, by interspersing (clustering) metadata and data blocks nearby on the disk, and using a larger block size (4 K and up), disk bandwidth utilisation increases to 47% in the new FFS design [126]. Clustering in FFS also improves crash recovery, as file system metadata is not physically concentrated in one localised area of the disk, but is spread over it.

3.3.3 Block Sizing and Allocation

A more general application of increasing the block size to lessen the effects of seeks and improve storage contiguity to benefit the large sequential transfers prevalent in file system traces is to adopt variable-sized blocks for files. In effect, the disk is treated as an area of storage from which variable-sized allocations are made. Such a problem is not new: efficient memory allocation and garbage collection has been long studied. Disks pose additional performance constraints because fragmentation seriously affects performance by translating into long seeks (excessive head movement), and garbage collection on disk is more costly than in main memory.

Wilson *et al.* [204] provide a comprehensive survey of the memory allocation literature. Memory allocation allowing variable-sized blocks has a disadvantage of extra metadata overheads for pointers to link allocated areas, and compaction suffers significant overheads. Knowlton [92] introduced the *buddy system* of allocation that is considerably more efficient when freeing blocks.

In buddy systems, only a fixed set of block sizes, $b_1 < b_2 < \cdots < b_k$, are available for allocation, and so there is a potential for storage to be wasted if an object is placed in a bigger block than necessary. However, such lossage is a problem for traditional fixed-size allocation schemes (though it is strictly bounded). A standard block sizing scheme is to use increasing powers of 2 (the *exponential buddy system*). Hirschberg [78] describes a *Fibonacci buddy system* in which the block sizes follow the Fibonacci series. The use of the Fibonacci series permits a larger number of different sizes, and hence increases the probability of a good fit, thereby decreasing internal fragmentation.

Koch [94] designed a disk allocation scheme based upon the exponential buddy system. Files are stored in a bounded number of contiguous *extents*, and a reallocation algorithm runs periodically to improve the allocation of poorly arranged files. Koch reports a mean number of extents per file of 1.5 and an average internal fragmentation of less than 4%.

Ghandeharizadeh *et al.* [65] present and analyse several algorithms for managing files in a hierarchical storage environment with a focus on the tradeoff between contiguity of files and the amount of wasted space. (Their work explicitly excludes support for striping, however.) Ghandeharizadeh *et al.* [64, 66] describe algorithms for the layout of files on disk such that each n block file is stored in no more than $\lceil \lg n \rceil$ separate extents in order to minimise seeks.

Seltzer and Stonebreaker [173] examine several variable block size file allocation strategies and conclude that such extent-based approaches offer excellent performance in read-mostly environments because of seek time minimisation and large sequential reads.

3.3.4 Log-Structured Approaches

A major trend in local file system design that took hold in the early 1990s was a move towards log-structured file systems to improve write performance [145, 159, 160, 170, 171]. Instead of the fixed layout schemes employed in traditional file systems such as FFS, log-structured file systems treat the blocks of the disk as a log to which data is appended. Unlike FFS-like file systems which incur seek overheads for each write, log-structured file systems instead batch all writes and then write all the changes to disk in a single large disk transfer operation. Typically, log-structured file systems divide the disk logically into *segments*, and write an entire segment at a time.

By performing large disk transfers, the impact of a seek is lessened. In addition, crash recovery is greatly assisted, because in the event of a crash, the file system software need only locate the last checkpoint and then perform a *roll forward* on the writes made after that point. This is in contrast to the time-consuming multi-pass consistency algorithm required in FFS-like file systems [127].

Because disks are finite in size, the log must *wrap* when it reaches the end of the disk. This necessitates special handling of free space in the file system. The most common approach is to employ a *cleaner* daemon that runs asynchronously alongside the mounted file system. The cleaner acts like a garbage collector, copying live blocks out of one or more segments containing deleted space and consolidating the live blocks together and writing them to clean segments in the log. The segments out of which the live blocks were copied are now free to be reused as clean segments for the log. Various policies can be enforced for segment cleaning, to balance the impact on the file system [124, 159, Chapter 5].

Another way to improve write performance, along the lines of log-structured file systems, is to log writes to a smaller, separate *logging medium* and then asynchronously commit the changes to the actual data disks of the file system. The logging medium could be non-volatile RAM [80, 112], or a dedicated, separate log disk [34]. Non-volatile RAM has also been used to implement reliable buffer and file cache mechanisms to mitigate the impact of periodic writing of dirty blocks to ensure file system consistency. Prestoserve [137] uses non-volatile RAM to cache NFS writes. Rio [32] and Phoenix [59] implement reliable file caches, making writes to the file cache as permanent and as safe as files on disk, but with much greater speed. Rio, for example, performs 4–22 times as fast as a standard Unix file system that employs write-through caching, or 2–14 times as fast as a standard Unix file system using write-back caching. Even when using delayed metadata writes in a standard Unix file system [62], Rio is 1–3 times as fast, but has the reliability advantage of synchronous write behaviour.

3.4 Disk Arrays

File systems that reside on a single physical disk have an inherent bottleneck by virtue of there being only a single head assembly through which to perform all reads and writes. As we have seen, the time needed to move this head assembly over the disk surface is probably the major limiting factor in terms of disk I/O and hence file system performance.

One way to mitigate this bottleneck is to increase, in effect, the number of head assemblies. The easiest way to achieve this is to create a *virtual disk* out of an *array* of individual physical disks. Not only does this increase the potential parallelism of reading and writing—because different areas of the virtual disk can be read and written simultaneously—but it also increases the potential reliability, as a hardware failure now no longer renders the entire virtual disk unusable. Such virtual disk arrays are commonly called *disk arrays*. An important, and the most prevalent, class of disk arrays, in which performance and reliability are emphasised, is called Redundant Array of Independent/Inexpensive Disks (RAID) [31, 67].

RAID employs *data striping* to improve performance, and *redundancy* to improve reliability. Striping is a way to distribute data transparently over disks in an array. The granularity of the stripe unit determines how many disks will be involved in a given I/O request. The larger the stripe unit, the fewer disks need be involved for small I/O requests, allowing multiple I/O requests to be serviced in parallel.

Redundancy in RAID involves using extra storage space to improve the reliability of the data stored in the RAID. The extra space holds *error correction* information for the actual file data stored on the RAID. The most prevalent error correction scheme used in RAID is *XOR parity*. In this scheme, a parity stripe unit is computed as the XOR of the corresponding stripe unit from each of the data disks. Such a scheme can tolerate the failure of a single disk, because the parity data can be used with the remaining good data to reconstitute the failed disk's data. There are other error correction schemes. For example, the *P+Q Redundancy* scheme uses *Reed-Solomon* codes to protect against the failure of up to two disks.

The major performance impact of using error correcting data in a RAID is that additional reads and writes are required when writing data, particularly for small writes that update only one data disk. This is because the parity data need also be updated on a write. In the case of a small write, a *read-modify-write* process must be used in which the old data must be read before the new data is written, to determine how the new differs from the old, and then applying the differences to the parity block. Thus, a small write actually requires two reads and two writes. Because parity is always written on a data write, if parity is kept on a single disk, that disk can easily become a bottleneck for the entire RAID.

The interleaving, or striping, of data and redundancy information over the disks of a RAID has a big impact on the performance and storage utilisation of the disks participating in the aggregate storage. There have emerged several characteristic organisations, or *levels* of RAID. These are briefly described below:

RAID 0: Data are striped across the disks comprising the RAID unit with no redundancy employed whatsoever. This gives maximal use of available disk space, and the best write performance, but with no fault tolerance. If a disk in the RAID fails, then the whole RAID fails (for all practical purposes).

RAID 1: Twice as many disks are employed in this RAID level as in level 0. Data are striped across the data disks, but, also, are *mirrored* on a corresponding redundant disk. So, for each data disk, there is another mirror disk. When data are written, both the data disk and its mirror are updated. When data are read, either of the data disk or the mirror can be read, as both contain identical information. Load balancing can be used to select the best disk to which to direct the read request, e.g., to the disk with the shortest request queue, or the one whose head is currently closest to the data. If a disk in a RAID 1 fails, all reads and writes will be to and from its mirror. RAID 1 has very high read performance.

RAID 2: This RAID level uses bit-level striping and memory-style error-correcting codes to detect and correct disk failures. So, a RAID with n data disks using Hamming codes for redundancy will use $\log n$ additional disks to store parity. If a disk fails, the parity disks can, in combination, detect which disk must have failed, and correct the failed data. As the RAID set grows larger, the number of parity disks grows more slowly. However, because disk failures are *self-identifying*, the extra parity encoding needed to identify which disk failed is usually redundant, making this scheme little-used.

RAID 3: Another bit-level striping scheme, RAID 3 uses the fact that disk failures are self-identifying to eliminate the $O(\log n)$ parity disks, replacing them with a single disk per RAID set. When a disk fails, the parity disk can be used in conjunction with the remaining good disks to reconstruct the failed disk's data. In this bit-interleaved scheme, every disk participates in each read or write, and, in effect, all disks operate identically (the heads are synchronised). This simplifies implementation, and enables a high bandwidth to be delivered, making RAID 3 attractive for multimedia applications where quality of service bounds must be observed.

RAID 4: Similar to RAID 3, RAID 4 is a *block-interleaved* scheme that uses a single, dedicated parity disk. Read requests smaller than the striping unit need only access a single disk, increasing the I/O parallelism of this scheme. The drawback, however, is that all parity updates on writes go to a single disk, which can quickly become a bottleneck.

RAID 5: This organisation is similar to RAID 4, but improves upon that scheme by distributing the parity information across the data disks of the RAID, instead of having it reside on a separate disk. This *distributed parity* means that, now, all disks in the RAID are used to store data, but also, some blocks of each disk are used to store parity. The equivalent of a single disk's worth of storage is still consumed by parity storage, but it is spread across all the disks in the RAID. Although the layout is more complicated, it means that no single disk becomes a bottleneck for parity writes, and also has the advantage that all disks can participate in reads. The precise distribution of parity can affect performance [108]. RAID 5 generally has the best performance for small and large reads and large writes of any redundant disk array, but its biggest drawback is the read-modify-write penalty for small writes.

Those are the major RAID categories, but by no means all. For example, the aforementioned *P+Q redundancy* scheme, which can protect against up to two disk failures, is often classified as RAID level 6. In practice, the most commonly used RAID organisations are RAID levels 0, 1, 3, and 5.

Although excellent performance can be obtained from a RAID with all disks operating, performance can be severely degraded if a disk fails. When a disk fails, the RAID is said to be operating in *degraded mode*. In this mode, *all* disks must participate in each read and write (except for RAID level 1), removing any I/O parallelism.

To counter this, a RAID can have *online* or *hot spare* disks—disks that are part of the RAID, but that are not used to store either data or parity in the course of normal operation. When a disk fails, its data is *reconstructed* onto a spare disk. This is done transparently, to ensure the RAID is not offline. If hot spares are not used, data reconstruction takes place when the failed disk is eventually replaced with a working unit. If *hot swappable* drives are used in the RAID, this can be accomplished without shutting down the entire RAID. *Distributed sparing* [130] and *parity sparing* [30] are two techniques that take advantage of online spare disks to improve the normal performance of the RAID, whilst still allowing fast reconstruction to begin.

Even in non-degraded mode, care must be taken to safeguard the integrity of the RAID to protect against system crashes. In particular, the RAID must keep track of not only which disks have failed, but also which logical sectors of a failed disk have been reconstructed, or which logical sectors are currently being updated. This is necessary to avoid reading *stale data* from a RAID. This information kept track of is referred to as the *metastate* of the RAID.

In addition, it is important to keep track of which parity sectors are *consistent* and which are *inconsistent* in the event of a system crash. Customarily, this means that before each write, the parity sector must be marked as inconsistent until new, valid parity is written. Upon a system crash, recovery mandates that all inconsistent parity must be regenerated. Inconsistent parity in the presence of a disk failure means that the data will not be able to be reconstructed correctly. However, because reconstructing a consistent parity sector also results in a consistent parity sector, it is permissable to trigger the regeneration of all parity sectors upon return from a system crash, instead of maintaining stable parity metastate information.

As well as individual disk failures, it is also the case that multiple disks can fail systematically. Typically, this is because the controller or bus to which they are connected fails. Thus, instead of a single disk failing, a *string* of disks fails. One way to combat this problem is to arrange error correction groups orthogonal to the actual hardware arrangement. Such a strategy is broadly called *orthogonal RAID* [139, 169].

RAID organisations are the subject of intense research, particularly in improving write efficiency and reconstruction performance. *Floating parity* [131] and *parity logging* [183] can improve small write performance. *Declustered parity* [79, 132] can improve reconstruction performance by distributing the parity groups such that the load is balanced more evenly across the disk array in degraded mode.

Research has been undertaken into employing adaptive RAID organisations, to match workload with the best layout. The HP AutoRAID system [203] uses two levels of RAID between which files may migrate automatically depending upon how often they are updated. RAID 1 (mirroring) is used for frequently updated data, and RAID 5 for infrequently updated data. Initially, all but 10% of data is stored as RAID 5. In the course of normal operation, frequently-updated data in RAID 5 is promoted to mirroring.

One of the major problems with RAID schemes is that they are effectively *bus-based* solutions, and so have restricted scalability. Typically, the disks of a RAID array are connected directly to a *RAID controller* that mediates all disk I/O. The RAID controller may be a custom hardware controller, or, alternatively—and perhaps more increasingly—a "software RAID" in which a RAID disk driver is available as part of the operating system. The RAID driver can then act as a "higher level" layer upon the native disk device drivers to create arbitrary RAID configurations using those disks (or partitions thereof).

The *RAIDframe* [38] software toolkit is a research vehicle for experimenting with different software RAID organisations. It has been ported to several variants of Unix, offering various levels of functionality: as a RAID simulator and a RAID device driver. For example, NetBSD 1.4 onwards includes a port of RAIDframe that enables, through a kernel configuration option, a RAID pseudo device driver. This driver allows RAIDs of levels 0, 1, 4, and 5 to be created using block devices or even using other RAIDs. So, it is possible to create a RAID 0 array out of several RAID 5 arrays, or to construct other arbitrarily complex hierarchies. The driver also supports hot spares, various parity declustering options, and so on.

Widely-available software RAID makes it fairly straightforward to build large disk arrays using COTS hardware. For example, a single PC clone with four wide SCSI host adapter cards, each driving fifteen 40 GB SCSI hard drives, could provide over 2 terabytes of RAID storage. However, in such a setup, the limiting factor is likely to be the memory bus bandwidth.

3.5 Networked Storage

There is a vast literature on networked storage. Put simply, *networked storage* is storage available via a local-area or wide-area network. Typically, a *client/server* model is employed, in which the actual disk storage is provided by *servers*, and accessed over the network by *clients*. The division of labour between client and server can vary greatly. At one extreme, the server can act as a "black box," providing all functionality of authentication, name lookup, consistency, and data delivery (e.g., Network File System (NFS) [162]); at the other, it can provide simply a low-level storage abstraction that clients may read and write (e.g., the Swarm storage server [75]).

From a hardware perspective, networked storage can be divided into the following broad categories:

Server Attached Disks: Storage made available by the server consists of disks that are directly attached to the server host. An archetypal example is one or more hard disks connected to the server via a SCSI host adapter. Only the server host computer has direct access to the disks. All other access—by networked clients—must be via the server itself.

Network Attached Disks: This organisation has disks directly attached to the network and able to send data to clients via some high-level networking protocol such as TCP/IP. Usually, network attached disks require some form of lightweight aid in the form of file manager servers that mediate client requests to the disks themselves. The file manager interacts with the network attached disks via a private network to ensure integrity of data on the disks is maintained. Once authorised, network attached disks deliver data directly to the client making the request.

Storage Area Networks: Here, a high-speed, private network separates storage devices (disk, tape, etc.) and storage servers. The storage servers act as a front-end to the storage pool, and any storage server can communicate with any storage device on the storage area network (SAN). Clients make storage requests via any storage server.

Data is transferred between client and storage devices via servers.

From a server/file manager point of view, networked storage can be divided into the following broad categories:

Central: All client requests are served by a single server, which carries the full burden of file management functionality. NFS [162] is the archetypal centralised server. Central server designs are relatively straightforward in design and implementation, but lack scalability.

Distributed: Distributed servers cooperate with peer servers in order to satisfy client requests. Typically, distributed servers will partition the name space and each will handle some portion of it. Examples of this category are Andrew File System (AFS) [167] and the Sprite file system [143].

Merged client/server: In this category, clients can act as both clients and servers, acting as a server for locally-attached storage, and as a client when accessing non-local storage. Most of the file management burden is placed on the clients. Network of Workstations (NOW) [6] and xFS [7,201] are typical examples of this category.

There is a lack of standardised terminology in the area of large-scale, reliable, scalable storage. Devlin *et al.* [48] offer a taxonomy describing typical enterprise-scale storage organisations. Much terminology, however, is industry-driven [56,185,199].

Current emphasis is away from SCSI-based disk systems to fibre channel-capable drives, which supports the notion of network-attached storage. Fibre channel [17] is a high-speed serial interface offering speeds upwards of 100 MB/sec. It is more flexible than SCSI in that it supports three basic interconnection topologies that may be scaled up: point-to-point; fabric (switch); and arbitrated loop (ring). An arbitrated loop can support up to 126 devices without the need of a switch. This is more scalable than SCSI, which can support only up to 15 devices per bus. In addition, fibre channel can be driven over much larger distances—1 to 10 km, depending upon wiring—as opposed to a maximum of 25 metres with SCSI. Fibre channel integrates both storage and networking protocols, and is implemented using six protocol layers.

3.5.1 Intelligent Disks

Storage research continues the trend of migrating functionality into the disk itself. Instead of moving data to code, one approach is to move the code to the data and execute high-level functions on the drive itself [1,91,158]. In effect, make the disk an object-oriented device that responds to high-level method invocations from clients. In such a model, files become objects, the management of which becomes largely opaque to the outside world.

Keeton *et al.* [91] at Berkeley put forward a case for *Intelligent Disks*, or *IDISKs*, that provides higher-level application functionality on the disk itself. Gibson *et al.* [68,69] describe a disk-centric storage architecture called Network Attached Secure Disks (NASD) that also concentrates functionality on the disk itself. An extension of this work is that of *Active Disks* undertaken at Carnegie Mellon and at the University of California, Santa Barbara.

Riedel and Gibson [158] at Carnegie Mellon describe the issues and potential benefits of being able to execute code directly on NASD. Applications studied include database select and parallel sort. Their proposal of active disks localises activity on the disk, with relatively modest processing requirements and no communication between disks.

Acharya *et al.* [1] at UCSB also propose an active disk architecture that is a little more powerful in its scope. In it, *disklets* run on disks, dispatched and coordinated by a host. They outline several applications, including: SQL select and group-by; external sort; relational database datacube computation; image convolution; and generation of composite satellite images. Their simulation results indicate active disks outperform conventional disk architectures.

The IDISK architecture described in Keeton *et al.* [91] is probably the most powerful, in that it posits higher bandwidth disk-to-disk communication, allowing for more general-purpose parallelism.

All of the active disk architectures are predicated on increased technological sophistication of the disk drive itself. The processors used in current drives are typically one generation behind those used in the host computers driving them. Making a transition from 0.68 micron to a 0.35 micron process would, for example, enable the integration of a 200 MHz StrongARM RISC core into the current ASIC die space, plus an additional 100,000 gates for specialised processing in addition to incorporating all existing formatting, servo, ECC, and SCSI functions [158]. Experts at Seagate anticipated that by 2001, drives will provide upwards of 100 MIPS in processing power, plus up to 64 MB of RAM and 2–4 MB of flash memory on the drive [5].

3.5.2 Parallel File Systems

In terms of software organisation, many networked storage architectures have been proposed. One important class is that of parallel file systems. Initially, these were relatively simple transpositions of the traditional Unix file system semantics that treated files as linear sequences of bytes [152]. However, one distinguishing feature of parallel file systems is that disk I/O is tightly coupled with the parallel computation. Thus, parallel I/O began to assume more importance in parallel computing, especially in those applications that are data-driven. The Message Passing Interface (MPI) working group is evolving standards for parallel I/O [35], as is the Scalable I/O Initiative [37].

Some parallel file systems recognise that files can be structured to mirror the application and workload under which they will be used. The Vesta Parallel File System [36] treats files as multiple disjoint partitions that can be accessed in parallel, and allows layout to be tuned to anticipated access patterns. The Hurricane File System [97] provides hierarchical building blocks that can be composed to custom-build access methods for files in a shared-memory multiprocessor environment. Application data files can thus be associated with precisely-tuned semantics appropriate to their chosen application and access patterns.

Disk-Directed I/O [96] is a scalable I/O approach for MIMD multiprocessor systems. Instead of compute nodes sending requests to I/O nodes independently, they instead *collectively* send a single request to all I/O nodes. The I/O nodes then control the flow of data back to the compute nodes to best maximise disk bandwidth. The Galley Parallel File System [140] implements a version of disk-directed I/O, although its I/O requests are not explicitly collective. The PARADISE system [26] integrates many theoretical parallel I/O techniques into an experimental file system, including disk-directed I/O; cooperative caching; flexible file structuring via metadata; and selectable file access methods. The TickerTAIP [28] parallel RAID architecture bridges across the parallel I/O and RAID models. River [9] is another approach to parallel I/O management. It provides a data-flow programming model and I/O layer for cluster environments. It uses a *distributed queue* to load balance the workload amongst data consumers in the system, and *graduated declustering* to do workload balancing amongst the data producers of the system. The goal is to provide persistent peak performance across workload perturbations.

The IEEE Storage System Standards Working Group has been working on a reference model for a large-scale, wide-area, scalable storage system designed for petabyte-scale repositories of potentially billions of datasets, each of which could be terabytes in size and spread across heterogeneous secondary and tertiary storage [39, 82]. The model is designed to be widely dispersed, geographically, yet support an I/O throughput of gigabytes per second. The High Performance Storage System (HPSS) project is an implementation based upon the IEEE Mass Storage System Reference Model. The HPSS concentrates on highly-parallel, supercomputer, and cluster environments, and seeks to deliver data at upwards of 100 MB/sec [40]. One of the design criteria of the HPSS is to utilise existing and emerging standards for increased deployability. Another design goal is to provide API access at appropriate levels for use by separate digital library, object storage, and data management systems [40].

3.5.3 Distributed File Systems

Distributed file systems are networked file systems that spread file system contents over several machines. In the simplest case, servers handle all file system responsibilities, including serving data; maintaining consistency; and handling locking. In more advanced cases, this workload is shared between clients and servers, and in some systems, clients may act as both clients and servers.

Although the NFS may be used to mimic a distributed file system, it is not really considered one. Sprite [143], developed at Berkeley, is a distributed file system in which the global file system name space is partitioned across multiple domains. A file server may handle one or more domains. Sprite uses aggressive caching to improve performance. Clients cache files locally, to lessen server workloads. However, when a file becomes write-shared, the server becomes involved, sending call-back messages to all clients with the file open to disable caching for that file. The AFS [167] is similar to Sprite in that the file system is distributed across servers. Like Sprite, AFS maintains state, and performs callbacks to clients when cached shared file data is modified by another client. Consistency in AFS is at the whole file level, with the last file written being the version maintained by the server.

The biggest problem with AFS and Sprite is that they do not scale well to large numbers of clients. Availability is also a problem, should a given server become disconnected. The consistency granularity of AFS is also weaker than traditional Unix semantics. AFS was commercialised and developed as the Distributed File System (DFS). The DFS is the basis of the Open Software Foundation (OSF) Distributed Computing Environment (DCE) [141], and provides stronger consistency semantics over AFS. Coda ("COnstant Data Availability") [166] builds further upon AFS, and is intended to provide improved availability and even support disconnected operation for mobile users. In Coda, the global name space is partitioned into volumes consisting of files and directories. A volume is the unit of replication. It may be explicitly replicated when it is created, and a *volume replication database* is used to keep track of all volumes. The volume replication database is itself replicated at every server.

Coda clients use more aggressive caching than in AFS in that they can proactively cache entire files for future disconnected operation. Disconnected operation may be the result of network inaccessibility to any Coda server replicating a volume used by the client, or a voluntary disconnection, such as a laptop computer being removed physically from the network. The local disk of a Coda client is treated as a file cache, and, during disconnected operation the client becomes a server, handling all requests out of its local cache. Upon reconnection, modifications are consolidated with the servers of the volumes used by the disconnected client. A repair tool is provided to allow the user to resolve inconsistencies that cannot be resolved by Coda automatically.

Berkeley's NOW project produced xFS, a *Serverless Network File System* [7]. In xFS, clients are both clients and servers: there is no centralised server (or small subset of machines designated as servers) in xFS. This greatly improves scalability. Metadata management is distributed across all nodes, instead of being partitioned onto canonical servers.

The Global File System (GFS) is a serverless file system in which a cluster of clients connects to network attached storage devices arranged in a network storage pool connected via a storage area network [182]. GFS uses a distributed lock manager controlled by the storage devices. Devices in the network storage pool are arranged into *subpools*, grouping devices with similar characteristics. These subpools are then exploited for efficiency. For example, file metadata is placed on low-latency subpools, whereas file data is placed on high-bandwidth subpools. Although GFS is designed for clusters, it can be extended to LAN and WAN environment by GFS clients exporting file systems via other protocols such as NFS and Hypertext Transfer Protocol (HTTP) [21].

Clusters are receiving a lot of attention as a way of producing low-cost, powerful, scalable computing facilities. Correspondingly, there has been significant cluster development resulting in mature products. Clusters are attractive because of their smaller administrative domain than wide-area distributed file systems. This allows for lower latency, higher stable bandwidth, and simpler security considerations. A distributed lock manager, for example, is easier to deploy with high performance in a SAN or cluster environment, than over a wide-area network.

VAXClusters and Tru64 Unix cluster provide high-performance, fault-tolerant cluster-level file systems [99, 125]. Cluster members can share locally attached storage, or use NASD over a variety of interconnection fabrics. A hierarchical lock structure and distributed lock manager control file sharing. The lock manager can detect deadlocks, and the system features failure recovery. The Lustre cluster file system is similar in scope [122].

The Berkeley NOW work has sparked much cluster-level file system work. The latter development of Sprite [143] gave rise to Zebra [76], which provided striping across disks. Instead of striping files, Zebra uses a per-client log which is striped across the disks rather like a log-structured file system. The Swarm storage system, developed at the University of Arizona, is similar in that it provides a striped log abstraction on cluster storage nodes for clients [75]. DEC's Frangipani [192] is a cluster-level storage system based upon their Petal [109] virtual disks. Petal provides virtual disks, each of which has a 64-bit address space. A virtual disk is made up of a pool of physical disks that may be spread across multiple servers. Petal is a software RAID in many respects. It features component failure tolerance and chained declustered data access. Copy-on-write techniques enable Petal virtual disks to support efficient snapshots, for online backup purposes. Frangipani builds on Petal to provide a consistent view of the same set of files to all clients, and uses a distributed lock manager to ensure coherence.

4 Digital Libraries Object Repositories

Although there has been extensive and intense research in the area of file and disk storage, there has been comparatively little undertaken in the area of storage for digital libraries. Most of what has been done is at the application level: data structures and file layouts for indices; protocols for search and data interoperation; file formats for archival storage; etc. Digital library metrics studies have focused on user interface and behavioural concerns, along with search session and query characterisation [110].

Although digital libraries are voracious consumers of storage, a tacit assumption in most cases is that the underlying operating system I/O substrate will be used for storage. An application-layer shim is applied to provide the more sophisticated I/O that is characteristic of digital libraries [135,136]. This may lead to a severe "impedance mismatch" between the lower and upper layers.

One notable exception, in which storage and access is given a great deal of attention, is in the area of digital multimedia. There is a large literature on the design and implementation of architectures for digital multimedia, with a particular focus on video servers. Although this is but a subset of the digital library universe, it indicates that an attention to low-level detail is warranted where performance is necessary.

Traditional file systems treat files abstractly as simple linear sequences of bytes, with little or no additional semantics beyond that. Traditional file system files have little in the way of *metadata* attached to files. What metadata exists is limited to information about the low-level storage of the file, and the ownership and access dispositions of the file. Almost no semantic information is associated with the file, reinforcing the "neutral linear byte stream" semantics prevalent in traditional file systems. There, the mantra is that applications provide semantics. Whereas that is true to an extent, as we have seen, great performance improvements can result from matching a file's storage and access at the low level to its intended application (e.g., Hurricane, PARADISE, *et al.*). File system studies that have tracked file types consistently report a good correlation between file size and file type (semantics) [49].

In addition, a file's type may limit its access semantics. For example, compressed files are often not amenable to random access, because the decompression state of the current block depends upon that of all the previous blocks in the file. Such files, therefore, will almost certainly be accessed linear sequentially. (There are compressed file formats that support nonlinear access, most notably many multimedia file formats.)

All this semantic information may be used to improve storage layout and efficiency. A file system that naively assumes all files will be accessed in the same way misses an opportunity to tune performance to file semantics.

4.1 Digital Objects vs. Files

A major differentiation between file systems and digital libraries is that file systems operate on files and digital libraries operate on *digital objects*. Digital objects are more sophisticated than files, and more abstract. Files, by comparison, are concrete and simple.

Digital objects have an underlying data representation of the digital object, and they also have associated metadata that contains information *about* the digital object. At a minimum, the metadata includes a unique *handle* [186] by which the digital object is known.

Files reside in file systems, whereas digital objects reside in *repositories* [88]. Repositories are responsible for storing digital objects and securing them according to *rights* associated with the digital object. These rights may include *terms and conditions* under which the digital object may be *disseminated* by the repository. A *repository access protocol* is used to effect appropriate access to digital objects stored in a repository, and to undertake the *deposit* of digital objects into a repository.

A digital object repository is *not* a digital library. A digital library is much more than a digital object repository, and includes functionality to facilitate user interaction; searching; browsing; work-flow and content management; and more. A digital object repository is one *element* of a digital library, much in the same way a file system is part of a larger *operating system* infrastructure. A digital object repository can provide a functional substrate upon which digital libraries may be built.

Another characteristic difference between digital objects and files is that digital objects are typically longer-lived. An important facet of digital libraries—like that of conventional libraries—is *content management*. Digital libraries usually host *collections*, the contents of which are chosen carefully. There is cataloguing effort associated with introducing an item into a collection, and so to maximise the return on such investment, items are usually introduced with a long-term archival storage goal in mind.

Although collection items do degrade, go missing, or are otherwise deleted from conventional library collections, the frequency of deletion is small compared with the frequency of access in the collection as a whole. Unlike conventional file systems, which may contain very short-lived "temporary files," or files whose contents are updated frequently, digital objects are relatively *static*, or *read-mostly* in file system terms. Even then, updating a digital object is usually considered a significant event, and may result in different explicit *versions* of the digital object.

Digital objects also may be complex compound objects. For example, a digital object could be the result of running a given program on a given input file with given parameters.

Finally, a digital object may actually contain no digital data at all: it may consist entirely of metadata that refers to some non-digital archival object such as an *objet d'art* owned by a museum. Unlike file system metadata, which is relatively simple and focused on low-level storage concerns, digital object metadata is very rich in that it may not be uniform from digital object to digital object. In fact, a given digital object may have several encodings of the same underlying metadata, to make it interoperable between several different cataloguing conventions.

4.2 Naming and Location

Naming is very important for digital libraries. Unlike local file systems, names for digital objects are usually intended to be *globally unique*. Part of this stems from conventional libraries, which deal with authoritative versions of objects, and part stems from the desire for federation.

Names in file systems denote specific bitstreams as they exist at the current time. Names for digital objects denote the underlying *intellectual property*, or digital object, and not specific bitstreams or particular renditions of that digital object. A digital object may contain or support several possible disseminations of that digital object, but these are not generally considered to be separate digital objects in themselves. In file system terms, a digital object is ultimately more like a *directory* in that it is a *container* for all the digital artifacts (metadata and data) directly pertaining to that digital object. One could represent a digital object using a directory in which reside individual files containing the digital content (metadata and data), or even symbolic links to other directories representing digital objects. The directory would represent the digital object, and the directory pathname would be its name.

The important point to note about digital objects is that they are *containers* rather than simply individual flat entities. So, whereas a name refers to the digital object container, accessing the innards of that container is a more complicated operation, and one akin to accessing an *object* through its *methods* in the object oriented paradigm. In this respect, digital objects are closer to the object-based storage paradigm of IDISK, Active Disks, and NASD (see the discussion starting on page 13). So, to access the content of a digital object might require qualifying the name with additional data (e.g., the "method"). The result of an access might be actual data, or other names to dereference.

4.2.1 Uniqueness and Location Dependence

There are two major classes of names: *locally unique* and *globally unique*. Locally unique names are unique to a particular server, that is, the same name on two different servers may refer to different objects. (For example, the file `/etc/motd` may contain two entirely different message of the day contents on two different servers at the same instance in time, even though they have the same name: `/etc/motd`.)

File system file names are typically locally unique, unless the file system is a wide-area distributed file system, in which case they are usually globally unique (often made so by qualifying them with the global server name).

In addition, a name can be either *location dependent* or *location independent*. Names that are location dependent are tied to a particular server or server cluster, and incorporate the server name as some facet of the name itself. Location independent names incorporate no reference to any particular server holding the named object, and part of the name resolution process involves identifying a server storing the object to which the name refers.

Location dependent names are undesirable for at least three reasons: 1) they are tied to a "physical location," in that if an object is moved (not copied) to another server, the original name becomes invalid; 2) it is not immediately discernible whether two differently named objects are in fact the same thing; and 3) a set of location dependent names is harder to scale because the set is tied to a particular cluster.

A ubiquitous example of a location dependent globally unique name is the Uniform Resource Locator (URL) [22] used in the WWW. URLs explicitly encode the host and location within that host of a given resource. When accessing an object named by a URL, the host named in the URL is contacted. For popular objects, this can lead to severe load problems on the server host so named. There are techniques for alleviating such congestion. HTTP redirects [21], Active Names [197], Persistent Uniform Resource Locator (PURL)s [174], the Internet2 Distributed Storage Initiative [16], etc., seek to make location dependent names less so by interjecting a layer of indirection in the resolution process.

4.2.2 Uniform Resource Names

A URL is a form of Uniform Resource Identifier (URI) [20]. A Uniform Resource Name (URN) [181] is also a form of URI, but one that is location independent. Unlike URLs, which have precise syntax and semantics, URNs have a precise syntax [134] but currently rather loose semantics [180].

An overview of work on URNs is given in [195]. Paskin [150] delineates many of the functional requirements and progress towards implementing URNs. The fundamental requirements of URNs include: global scope; global uniqueness; persistence; scalability; legacy support; extensibility; and independence [181].

The basic syntactic form of a URN is *urn:nid:nss*, where *nid* is the *namespace identifier*, that identifies the particular type of URN, and *nss* is the *namespace-specific string* under the auspices of the *nid*. The semantic interpretation and syntactic format of the *nss* is local to, and defined by, the scheme implemented by the naming authority controlling the *nid* namespace.

There are several efforts currently underway to introduce a working URN scheme. The World Wide Web Consortium (W3C) has a working group looking at URNs [83]. Their effort concentrates on resolver discovery, and they propose adaptations to the existing Domain Name System (DNS) to enable a resolver to be located for a given URN [128]. This approach adds a new type of record to DNS databases: the Naming Authority Pointer (NAPTR) [129]. The NAPTR includes rewriting rules to enable a URI to locate an appropriate resolver for resolution [129]. The W3C has proposed a simple mechanism for resolving URNs via HTTP [44]. Powell [154] describes a simple technique that enables the resolution of URNs using the Squid WWW proxy cache.

Two major URN initiatives are the CNRI Handle system [186] and the Document Object Identifier (DOI) [149]. The DOI is a publisher-led effort to develop a URN scheme for persistent identification of intellectual property that includes a mechanism for resolving the DOI to some useful resource or service associated with that intellectual property. As well as identifying an item of intellectual property, a DOI includes basic metadata describing the target of the DOI [149].

The DOI actually uses the Handle system as its resolution mechanism. The Handle system is a global, decentralised, replicated URN system. It is comprised of a *Global Handle Registry* and many *Local Handle Services*. The general syntax of a handle is `urn:hdl:`*naming-authority*`/`*unique-identifier*. The *naming-authority* is the organisation that has authority over that particular portion of the global handle namespace. In DNS terms, it is akin to being authoritative for a domain. The *naming-authority* has administrative control over the *unique-identifier*s.

The Global Handle Registry is akin to the root nameservers in the DNS. Local Handle Services manage actual handles for one or more naming authorities. A naming authority may be *homed* at either the Global Handle Registry, or at a Local Handle Service. A naming authority can be homed at only one Local Handle Service (or, alternatively, at the Global Handle Registry). Both the Global Handle Registry and a Local Handle Service may actually be comprised of several servers, for the purposes of fault tolerance and load balancing. The service at which a naming authority is homed has administrative control over the handles within that naming authority.

A client library is required for applications wishing to resolve handles. Handle servers feature caching, and handle resolution is somewhat akin to that of the DNS in that first the authoritative naming authority is sought, and then the handle is resolved via a handle server of that naming authority. Handles may actually be comprised of a set of values. Each value has its own information, including a *type* entry. Handle resolution queries can thus specify that only values of a certain type, e.g., URL, be returned by a server. This allows a measure of *content negotiation* by the client.

The Handle system is described in more detail in [187]. The chief disadvantage of using handles as a ubiquitous naming scheme is the overhead of registering and resolving them in the global system. For short-lived, temporary objects, this is a significant additional overhead. However, handles are designed primarily to designate significant archival content: digital objects.

4.2.3 Scalable Object Location

Name space operations are characteristically lookup operations. An abstract way to think of name space operations is as a hash table in which the key is the name, and the value is the set of operations supported by the underlying object.

One way to scalably distribute such a hash table across the nodes of a cluster is to use a scalable distributed data structure (SDDS) [47, 98, 118–120, 198]. Born out of dynamic hashing [105, 106], *linear hashing* [107, 113] is one of the most common family of SDDS. Litwin, in particularly, has proposed many extensions to the basic linear hashing approach, including variants featuring grouping [115], mirroring [116], security [117], scalable availability [114], Reed-Solomon codes [121], and a variant tuned for switched multicomputers [90].

SDDS have three basic design properties: 1) there is no central directory through which clients must access, avoiding "hot spots;" 2) expansion to new servers is gradual, and according to load; and 3) the access and maintenance primitives do not require atomic updates to multiple clients. These properties provide excellent incremental scalability.

SDDS hash tables are accessed via a primary key. For a digital library object repository, we will use the object's URN as a primary key. Because a URN is variable in length, we can canonicalise its format by hashing the URN string to a fixed signature, for example to the MD5 hash of the string. The MD5 hash will canonicalise any URN string to a 128-bit "object ID" with low probability of collision (2^{-128}). It will also pseudo-randomise any URN, due to the one-way nature of the MD5 hash function. Other suitable candidate one-way hash functions include the Secure Hash Algorithm (SHA) [57], which hashes an arbitrary length bit stream to a 160-bit string. SHA-1 is used in cryptographic applications for digital signatures.

Randomised placement approaches have been gaining favour in storage area networks because they offer statistical guarantees for access and provide good incremental scalability. The problem with approaches such as RAID and similar striping approaches is that they do not scale well with the addition of new disks. With high-efficiency parity layouts, often the entire layout must be recomputed with the addition of new disks, necessitating a large movement of data. Randomised placement also has the advantage that it performs well for non-predictive access patterns.

The SICMA [25] and RIO [23] projects use randomised placement for multimedia servers. Both are resource efficient, and ensure an even distribution of requests amongst the disks in the system, even in the presence of non-predictive access patterns. RIO also employs partial replication of disk blocks to further evenly distribute the request load across the disks in the system [163]. It has been shown that even for predictive access patterns, RIO performs equivalently to that of normal striping approaches [164].

The PRESTO [19] project improves upon SICMA and RIO. PRESTO uses a parity approach rather than straightforward replication to improve space efficiency. PRESTO also has an efficient re-mapping technique that requires only $1/(n+1)$ virtual data blocks to be moved when adding an additional disk [19].

With the increasing popularity of peer-to-peer networks for cooperative wide-area storage and serving has come a focus on algorithms for scalable object location. The explosive growth of the WWW has necessitated a need for scalability to alleviate "hot spots" and ensure consistent and timely access to highly popular content. In particular, Karger *et al.* [89] introduced the concept of *consistent hashing* for locating trees of caches within a dynamically changing set of hosts. Consistent hashing has the desirable property that each node in the distributed hash table stores approximately the same number of keys, but there is relatively little movement of keys as nodes enter or leave the system. Consistent hashing also has the desirable property that not all clients need know about all the nodes participating in the distributed hash table, removing the need for global consistency and synchronisation.

Chord [184] uses a variant of consistent hashing to effect distributed lookup in a peer-to-peer environment. It is used as the distributed hash layer of the Cooperative File System (CFS) [43], a peer-to-peer wide-area read-only storage system. Chord uses $O(\log N)$ messages to look up a key in a network of N nodes. The caching scheme of Karger *et al.* [89] has $O(1)$ lookup, but with more routing information stored at each node than Chord.

Plaxton *et al.* [153] describe a distributed data location technique designed to locate the closest copy of a data item to a client. Like Chord, it also uses $O(\log N)$ messages in the lookup, but has stronger guarantees about how far those messages travel within the network. The OceanStore [100] distributed storage system uses a variant of the algorithm of Plaxton *et al.*

Ratnasamy *et al.* use a d-dimensional Cartesian coordinate space for their Content-Addressable Network (CAN) [157]. The CAN provides $O(dN^{\frac{1}{d}})$ message key lookup, and requires $O(d)$ state maintained at each node. This keeps the per-node state fixed, but at an asymptotically worse lookup cost. Past [50] uses a protocol similar to Chord, but is based on prefix routing.

The proliferation of recent schemes illustrates the necessity of a scalable lookup system for quickly and efficiently locating resources in a networked environment. The peer-to-peer schemes described above also tackle wider problems such as highly dynamic node participation in the network; inconsistent views; system security; anonymity; administration; and fault tolerance. In particular, the volatility of both the network membership and its topology is expected to be significantly lower in a storage cluster: nodes will join the network when new disks are added, and leave when they are removed for replacement.

4.3 Redundant Encoding for Reliability

Many schemes have been proposed for redundantly encoding data to improve reliability and tolerate faults. Several schemes have been proposed for the LH* family of SDDS, including mirroring [116], bit-level striping with parity [117], data with scalable parity [114], and Reed-Solomon encoding [121].

Other approaches are available for increasing reliability. Upfal and Wigderson [194] introduce a technique for replicating k copies of a datum across nodes in a distributed system to improve reliability in the presence of failures. Their technique requires that only a majority of the k copies be read or written in any given read or write of the datum.

Rabin [155] proposes a technique for "efficient dispersal of information" in which data can be encoded into n pieces such that only m out of n need be read to recreate the data. Alon and Luby [3] present a linear-time algorithm for an (n, c, l, r)-erasure-resilient code that can be used for encoding and decoding a n-bit string into a set of l-bit pieces such that when decoding, any subset of the l-bit pieces whose combined length is r is sufficient to recreate the original string. The encoded string is of size cn, and r need only be slightly larger than n.

The PRESTO project [19] employs a simple parity-based redundancy scheme. Each logical block is decomposed into k equal sized sub-blocks ($k > 1$), plus an extra parity block the same size as a sub-block. When reading a logical block, any k of the $k + 1$ sub-blocks can be read to reconstruct the original logical block. When writing, all $k+1$ blocks (sub-blocks plus parity sub-block) are updated. The sub-blocks are pseudo-randomly distributed throughout the system. The system can tolerate with high probability the failure of any single storage server. In PRESTO, $k = 4$ for read-mostly objects, giving a 25% storage redundancy overhead [19].

4.4 Metadata

Metadata is, broadly speaking, "data about data." However, that simplification hides the rich complexity inherent in metadata. Metadata has widely varying uses, and that use may be subjective, meaning that metadata is interpreted in a specific *context*. A digital object may be put to different uses by different entities, such as publishers, archivists, and end-users. Such entities will often create and interpret metadata specific to their own intended purposes. Rather than requiring a global union of all possible metadata, it is more flexible to be able to *aggregate* diverse sets of metadata for a given object to enable interpretation of that object by a diverse set of end users.

One approach that has curried widespread support is to consider a digital object as a *container* into which is placed digital content and associated metadata. The Warwick Framework [102], which arose out of a metadata conference, describes this approach. It seeks to address some of the limitations of the Dublin Core [51, 193] metadata set, the chief one being that the Dublin Core seeks to be a global metadata set, and such generality is undesirable in some instances.

In the Warwick Framework, the metadata for an object is actually a container holding *packages* of metadata. Each package can either be, itself, a metadata container; a *metadata set* containing actual metadata; or an *indirect* reference to an external metadata object, referenced via a URN. A client may access the contents of a metadata container via an operation that returns a sequence of packages inside the container. The client may then skip over unknown packages—metadata that the client does not know how to deal with, or cannot access—and use only metadata that it "understands" for the application at hand [102].

4.4.1 Buckets

The theoretical framework for the Warwick Framework was laid down by Kahn and Wilensky [88]. The Warwick Framework was subsequently generalised by recognising that there is no real distinction between metadata and data, and so containers should store both. Daniel and Lagoze [45] describe such a generalisation, known as *Distributed Active Relationships*, later more commonly known as *buckets* [138]. An architecture based around this approach is FEDORA [151].

The bucket paradigm treats digital objects as abstract data types whose contents are accessible through defined methods that allow a user to interrogate the contents of the digital object. In this way, more "intelligence" is invested in the digital object itself, and this makes the digital object less reliant on functionality built into the repository in which it is stored. This philosophy is termed Smart Objects, Dumb Archives (SODA) by Maly *et al.* [123]. There are interesting parallels between the philosophy of SODA and the disk-centric approach of IDISK and NASD, most notably that all emphasise self-management of objects.

Each bucket in the SODA model has a handle, and contains one or more typed *elements* and zero or more *packages*. Packages are used to aggregate elements. Typically, several different renditions of the same document, e.g., PostScript, PDF, TeX will form a single package, with each different format of the same document being a distinct element of that package. A package can also contain metadata, and terms and conditions for accessing the bucket, and can even be a pointer to a remote bucket, element, or package. Packages or elements in a bucket can, themselves, be buckets. The entire bucket has several primitive methods to allow controlled access to the contents of the bucket. The NCSTRL+ project [138] seeks to expand upon the NCSTRL distributed digital library [111] by using the bucket approach, amongst other things.

The FEDORA model is very similar, except that packages are, instead, typed byte streams called *DataStreams* that are accessed via *disseminators*. A *Primitive Disseminator* allows bucket-level access, whilst individual *Content-Type Disseminators* allow structured access to the individual DataStreams.

4.4.2 Terms and Conditions

An advantage of making digital objects self-contained, as in the bucket approach, is that it ties them less to a particular repository, and so makes replication and distribution easier in a heterogenous architectural environment. In particular, terms and conditions can be implanted within the digital object, instead of being wholly the responsibility of the repository. So, the move is to secure digital objects, rather than communication channels, which simplifies the superdistribution of digital objects.

Cryptolopes [71] adhere to this philosophy by encrypting the contents of a digital object, and including encrypted versions of the keys used to do so, along with digital certificates and checksums that may be used to verify the integrity and authenticity of the content. A third-party *clearinghouse* is used to enforce the terms and conditions of the bucket, facilitating the buying or obtaining of the appropriate keys to decrypt the content [95]. A similar technique is used in the DigiBox architecture [179].

Finally, the Resource Description Framework (RDF) is gaining popularity as a model for encoding metadata for digital objects (resources) [133]. The RDF uses XML as its encoding systems, and provides a means for associating properties with resources. The properties may be assigned their own XML namespace, to disambiguate the notion of, say, "creator" in one scheme from "creator" in another scheme. In this way, the RDF supports the ability of being able to describe the same resource (digital object) with heterogenous metadata sets, as in the Warwick Framework.

The major disadvantage of many metadata encoding formats, from a low-level storage viewpoint, is that they are often free-format, and so relatively uneconomical and unwieldy in terms of storage management. Importantly, too, they are relatively expensive to parse when compared to typical file system metadata, which, for efficiency reasons, is necessarily compact and fixed in format. It is a significant challenge to extend the richness of digital object metadata to low-level repository storage.

4.5 Digital Object Repositories

Digital object repositories are akin to the file system layer of traditional operating systems. They are responsible for the storage, management, and access of digital objects. They provide low-level functionality upon which digital library functionality may be built.

Traditionally, a digital object repository is seen as a form of "middleware" that provides a shim to the operating system file system and network layers to allow low-level access to structured digital object content. This is a utilitarian stance, to leverage existing storage system functionality in a platform-independent fashion. It does, however, have the potential to divorce digital object storage from the actual low-level storage subsystem, enough to cause a significant potential loss of efficiency.

There are two competing goals when designing a digital object repository: preservation versus efficiency. A repository designed with preservation as the primary goal seeks to make the storage as simple and transparent as possible, so that recovery of data can be done as easily as possible. This involves the use of *self describing* files and data structures; no deletions; and use of only disposable auxiliary structures (ones that can be faithfully recreated from scratch, if necessary). The major impact of such an approach is a significant detriment on performance, and so little work has been done in this area. Crespo and García-Molina [42] propose a layered digital archive architecture with long-term data preservation as its primary goal.

Instead of designing preservation into a digital object repository from the ground-up, it is usually assumed that the low-level storage will be supported by a disaster-recovery plan that allows it to be recovered after significant organisational loss. Allied to that is a *copy-forward* preservation approach, that migrates the low-level storage and organisation of digital objects into new, appropriate formats as system software undergoes generational change.

4.5.1 Kahn-Wilensky and its Extensions

A significant abstract semi-formal description of a digital object repository was introduced by Kahn and Wilensky [88]. They introduce many key concepts that have been built upon subsequently by researchers. Important amongst these concepts are the notion of a digital object and its handle, as discussed previously. They also describe the notion of *repository*, and a *repository access protocol (RAP)* which may be used to control the deposit of and access to digital objects in the repository.

A repository is a logical entity in that the name by which a repository is identified need not correspond to a unique server. Rather, it may correspond to a list of servers, each of which are responsible for the stewardship of digital objects stored in that repository. This necessarily makes the concept of repository a distributed one.

In the Kahn-Wilensky approach, digital objects are accessed via their associated handle. Handle servers are used to locate a repository at which a digital object is stored. This repository can then be used to access the digital object by means of the RAP. Access to a digital object at a repository is by means of the digital object's handle, a service type, and possible additional parameters. Such an access is termed a *dissemination*.

Kahn and Wilensky [88] define only three service types for their RAP: ACCESS_DO, DEPOSIT_DO, and ACCESS_REF. The first two are for accessing and depositing a digital object, respectively. The third is to access the reference services of the repository. This allows a client to determine the contents of a repository.

Lagoze [101] describes an implementation of the Kahn-Wilensky repository approach called Inter-operable Secure Object Stores (ISOS). ISOS uses the distributed object store Common Object Request Broker Architecture (CORBA) for its underlying implementation mechanism.

Dienst [46, 104] is another implementation of the Kahn-Wilensky approach. Dienst uses HTTP as its basic transport protocol, and partitions digital library functionality into several basic services. These services may be further refined by arguments to the service type when accessing digital objects within Dienst, analogous to the service type and parameter mode of repository access in the Kahn-Wilensky model. Dienst uses handles to name its objects.

The Networked Computer Science Technical Report Library (NCSTRL) [111] previously used Dienst for its digital library services. NCSTRL has been subject to active development, primarily to extend to it the notion of multiple collections or genres [138].

The Cornell Digital Library Research Group designed the Cornell Reference Architecture for Distributed Digital Libraries (CRADDL) infrastructure to support the collection concept [103]. CRADDL is a layered model, and uses CORBA for its implementation. The repository service layer of CRADDL is called Flexible and Extensible Digital Object Repository Architecture (FEDORA) [151]. FEDORA is designed to support the bucket model of digital objects and metadata, as described in Section 4.4. The FEDORA approach invests much of the intelligence within the objects themselves. Each object is opaquely packaged and contains a *primitive disseminator* through which the actual contents of the object may be inspected and retrieved. The primitive disseminator can be thought of as basic *bootstrapping* primitive methods that allow a remote entity to access the particular content of a given object. The primitive disseminator guarantees a minimal set of services for handling an object. Handling, in this context, also means incorporating new content into a digital object [151].

Because much of the functionality resides within the objects themselves, the RAP in FEDORA is relatively simple, and involves mainly the creation and deletion of digital objects, and associating URNs with them. The repository also provides an environment in which content access and authorisation methods may be executed [151].

4.5.2 Other Repository Approaches

An alternative approach to digital object repositories is to view digital object stores as semi-structured databases. The metadata of a digital object is stored in fields in a database table, whilst the data of a digital object are treated as binary large objects [77] stored in the database. A database provides a flexible method of searching and storing metadata, and the advances in distributed databases provides a ready platform upon which distributed digital object repositories may be deployed.

IBM Digital Library [85] employs the database approach. Its architecture consists of a *library server* and one or more *object servers*. The library server handles the metadata and searching, whilst the object servers handle storage and delivery of the actual digital content.

A client accessing the digital library gains access to a digital object via the library server. Once the library server has authenticated and located the object server storing the digital object the client wishes to access, the digital object is transferred directly between the object server in question and the client, requiring no further participation from the library server.

Although there may be multiple object servers, and those object servers may be attuned to the content they deliver, there is only a single library server per digital library. This creates a bottleneck in the architecture, especially in metadata-intensive applications. Because the library server is a database, it should be possible, though, to parallelise it to allow it to scale up.

The Oracle Internet File System (*i*FS) is another database-driven file system [142]. The underlying database engine allows more flexible and extensible metadata to be attached to files than is supported by traditional file systems. It also supports flexible "views" of files within the file system. The *i*FS also supports versioning of files, and check-in and check-out access for file update. The protection model is access control list-based, rather than permission-based as with many file systems.

Files in *i*FS are *parsed* and stored in a canonical internal format. *Renderers* are used to reconstruct parsed files in an application-specific way. One of the features of *i*FS is it aims to be "protocol independent" when accessing files. The *i*FS can be accessed via SMB, HTTP, FTP, IMAP4, and SMTP. XML and Java are the technologies through which the *i*FS API is used.

Like the IBM Digital Library, *i*FS is a step towards *content management* rather than *file management*. Content management is similar in most respects to the area of digital libraries in that they share common goals and services. There is perhaps more emphasis on *work-flow*, though, in the area of content management.

The SDSC Storage Resource Broker (SRB) architecture [14] is an API for managing access to widely distributed heterogenous data objects. One notable feature of SRB is that it allows access to objects stored in a variety of storage systems, such as databases, file systems, and tertiary storage, providing a simple, clean interface to the client. The Storage Resource Broker uses a metadata catalogue (MCAT) to keep track of resources belonging to collections managed by SRB.

The Stanford Infobus [146] is a CORBA-based digital library object repository mechanism for mediating access to distributed digital objects. The Infobus uses *library services* to perform various functions on objects stored within the Infobus. *Library service proxies* mediate client access to the Infobus and its library services (and hence to the objects therein). The University of Michigan Digital Library [53] uses an agent-based approach to broker access to data stored within it.

Many of these systems are designed to handle widely distributed collections, and do not focus on local, cluster-based repositories, though. This complicates the design requirements, placing stricter emphasis on authentication, heterogeneity, and network infrastructure. This, in turn, makes the opportunities for optimisation scarcer.

5 Proposed Research

Several important themes have emerged in the preceeding discussion. Chief amongst these are the following: there is a great demand for scalable storage; a storage system's workload has significant impact on its performance and on what design helps optimise that performance; different types of files have different access semantics and requirements (e.g., high-bandwidth vs. low latency); a storage system's performance will significantly improve if it is tuned to its workload; there is a trend towards serverless, intelligent disks and self-management of data; and, digital library object repositories are characteristically different from ordinary general-purpose file systems, primarily in that they are object-based, have richer metadata, and are read-mostly.

Previous work on storage systems has been almost exclusively in the context of operating systems services in which the policies and mechanisms have been limited by the demands of the overall system performance. For example, the need to provide CPU time to running jobs, and maintain a good response time to users means that time spent, say, in the block allocation algorithm for newly-written blocks must be relatively limited, placing greater limits on the amount of layout optimisation that can be achieved. It is only with the recent advent of intelligent disks and network-attached storage that dedicated processing power and autonomy for file system related activities has the potential to improve storage efficiency and I/O performance significantly. Advances in hardware, and the increased emphasis on storage, means that it is now cost-effective to devote relatively substantial computing resources to support storage than was possible before.

File type (and by definition, its semantics) has enormous impact on file storage and access performance. Yet, aside from application-driven file I/O prominent only in parallel computing, file type has not played a role in file system design. Part of this stems from the need to provide a system-wide compromise solution within the available system resources. However, a larger problem lies with the difficulty of identifying file types, especially as storage systems often deal with a lower-level abstraction than files, namely that of blocks. At a block level, it is harder to optimise macro-scale performance. Yet it is that macro-scale behaviour that can seriously affect performance. Looking only on the smaller scale can miss potential optimisations.

With digital libraries, digital objects carry much richer explicit metadata than do conventional files in file systems, allowing for closer "typing" of files via explicit cataloguing rather than relying on guessing via signature "magic numbers."

In this research we propose to investigate the use of metadata and workload information in cluster-based scalable storage system performance. In doing so, we will address several of the themes mentioned at the beginning of this section.

5.1 Measuring Storage Systems

For the purposes of this research, we must identify aspects of storage performance in which we are interested. We must then originate some way to measure those aspects, to impart some objective means to observe performance of different schemes employed.

There are several aspects of storage system performance worthy of investigation. These aspects include the following: 1) access speed; 2) storage utilisation; 3) cohesion; and, 4) fault tolerance. In cluster or networked storage, we are also interested in network overheads.

For access speed, two time measures are of interest: access latency and sustained transfer speed. The former is the delay between requesting an object stored in the system and receiving the first data. The latter is the average rate of data transferred during the actual data transfer period. Sustained transfer speed may be calculated both with and without the initial latency included.

Storage utilisation usually measures the ratio of the amount of data stored in the system against the amount of storage allocated for storage. This may include all ancillary metadata structures. Almost always, it measures the percentage of space "lost" due to blocking, i.e., the amount of bytes unused in an allocated block.

Cohesion is sometimes referred to as *fragmentation*, and usually is a measure of the spatial locality of blocks comprising a file. Cohesion could also measure temporal locality of reference. For the purposes of our research, the amount of gross head movement is a good measure of overall file system cohesion.

Fault tolerance is the ability to operate in the presence of and recover from faults. Faults may include the loss of hardware components such as disk blocks, entire disks, disk controllers, network adapters, or cluster hosts. Faults may also include software failures due to application or operating system bugs. Fault tolerance may complicate storage system design, and lower performance. However, some fault tolerant techniques, e.g., replication, may actually improve performance by increasing the opportunities for parallelism in data access. Measures of fault tolerance are usually in terms of system performance in the presence of defined faults. For example, the access and sustained transfer speed could be reported for increasing numbers of disk failures.

Finally, network overheads measure the amount of network traffic needed in addition to that needed actually to transfer data. Such overheads may include control and coordination messages needed to effect a transfer, especially if data are striped across more than one storage unit.

5.2 Online workload characterisation

If the type and past access history of a file provides useful data as to its efficient storage and access, it would be useful if an accurate snapshot of the digital object repository contents were available. Most file system work makes either static assumptions about its workload, or seeks to exploit temporal locality of reference. But, the handling of a large multimedia file might benefit from a radically different approach to that of a small text file. Even within the same type of file, some files might be more active than others. There may be characteristic access patterns common to certain file types, and files of these types may even exhibit particular discernable characteristic access patterns.

Although we believe the digital object repository workload has significant impact on its performance, there is currently little research into the actual workload characteristics of digital libraries. Much of the digital library metrics work has focused on user interface concerns, and low-level information retrieval concerns such as query lengths and term frequencies.

We believe an "instrumented" storage system would not only provide useful research data on digital library content usage, but would also be a useful input into storage system placement and access algorithms during its actual operation.

There are two levels of access information in which we are interested. The first is object-level access data. This charts access on an individual digital object basis. We can think of this as a *metadata-level* trace. The other level is that of individual disk storage units comprising the storage cluster. This is more of a *block-storage-level* trace, and measures in part the overall utilisation of an individual disk node in the system.

For the metadata-level trace, we are particularly interested in tracking file attributes and access statistics with the type of digital object. Particularly, we are interested in whether access for a given file type is typically read or write; random or sequential. For file attributes, we are significantly interested in the file size. Combined with characteristic access mode, file size is an important component to determining an effective block size for a given object, and whether or not caching will be effective.

For the block-level trace, we are interested in disk node utilisation both in terms of idleness and also node storage utilisation. This will give data on load balancing—in terms of workload and storage allocation—across the disk nodes.

5.3 Specialisation

The central idea in this research is whether the special characteristics of digital libraries object repositories in a cluster-based storage environment can be used to improve performance. Specifically, can type-specific information and a read-mostly collection be exploited?

One important way in which type specific information for an object can be exploited is to provide different storage and handling methods according to object type. So, for example, a different block size, file layout, caching policy, and striping approach may be adopted for MPEG multimedia files than for text documents.

One way to cater to object specialisation is at the software level. A repertoire of policies and mechanisms may be known to the storage system, and these employed according to an object's determined characteristics (e.g., object type).

Another form of specialisation is at the hardware level. Some storage devices have different characteristics. So, a storage device with low latency and low bandwidth may benefit certain types of files more than a storage device with high latency but high bandwidth. In addition, because certain storage organisations benefit certain workloads to the detriment of others, storage devices also should be allowed to specialise for certain tasks. Rather than being a jack of all trades and master of none, a storage device should be allowed to excel at certain tasks best attuned to its hardware.

One possible disadvantage of such specialisation is that it may be done badly. For example, too many devices may specialise in catering to write-intensive objects when the storage system global workload has few writes. It is hoped that the ongoing online workload characterisation mentioned previously would assist in providing feedback in that regard, allowing for a dynamic reconfiguration of device roles or collection apportionment. If we think of a storage system as a kind of market economy, the device specialisation would be seen as the supply, and the workload statistics as an indication of demand. Attuning the two would be a long-term goal of the storage system.

5.4 Exploiting Low Volatility

It is contended here that digital libraries object repositories exhibit less volatility than general purpose file systems. In particularly, they tend to be "read mostly," in that the ratio of reads to writes is very high. Part of this stems from the generally long-term archival nature of digital libraries object repositories. Another rationale stems from the overt cataloguing effort that takes place in digital libraries compared to ordinary file systems. A result of this is that such objects tend to be longer-lived. Longer-lived objects make long-term optimisation more worthwhile.

It is anticipated that most explicit write activity in a digital libraries object repository will take place when new objects are deposited in the system. To ameliorate the effects of this activity from normal read activity, one approach would be to adopt a "staging area" technique in which designated write-optimised storage units accept new content. After the new object is written and analysed, it can be relocated to a more permanent home on another storage device (or across multiple devices).

A big advantage of using a staging area in a read-mostly environment is that the staging area can be optimised for writing and recovery. In particular, log-based storage techniques and expensive persistent caches can be employed in staging area devices, thereby improving and speeding up crash recovery. In addition, file system studies have found the general trend that files tend to be either very ephemeral or else relatively long lived. Like *generational garbage collection*, a staging area allows longer-lived objects to be promoted to longer-term or *deeper* storage within the system.

If we accept the thesis that in a read-mostly environment, most files are typically written and then read thereafter, the overheads of moving from the staging area is a small cost easily amortised over the lifetime of the file. Yet the staging area provides an opportunity of excellent write and recovery performance, and is an effective way to "weed out" those objects which are indeed ephemeral.

Objects in the staging area could be "flushed" to more permanent storage according to various policies. For example, objects could be flushed out after attaining a certain age (similar to the asynchronous buffer cache flushing daemon implemented under many Unix systems); when the staging area attains a designated "fill level" (i.e., like a cache replacement policy); or when an optimal location in long-term storage has been determined for the object.

Note that the staging area is simply an extreme form of the specialisation approach described in Section 5.3, except that instead of specialising according to certain file types, the specialisation is more towards a particular function (i.e., writing). Exploiting such specialisation is a topic of interesting future research.

5.5 Decentralising the Name Space

A common theme emerging from workload studies is the following phenomenon: pick a byte at random, and it is likely to belong to a large file; pick a file name at random and it is likely to name a small file. So, although the name space is dominated by small files, the transfer bandwidth is dominated by large files. Furthermore, name space operations are characteristically small transfers favouring low-latency, and often are bursty in nature. This favours a decoupling of control and data pathways of the storage system, allowing optimisation of network traffic on each.

As we have seen previously, digital libraries object repositories are object-based rather than block-based. Content is accessed by name, or more abstractly, by invoking a method at a name. This resolution mechanism is a significant workload for the system to bear. Clearly, to handle such a task at a single "gateway" node to the storage cluster is not scalable. Serverless file systems indicate clearly that scalable performance is attainable by distributing the server burden across all nodes in the system: in effect, every node becomes both client and server.

In a cluster-based/SAN environment there is the complicating notion of the storage nodes inhabiting a "private" network, whilst requests enter the system via some portal connecting to the outside "public" network. Necessarily, there is some finite entry window. However, this entry load can be mitigated by minimising the task performed at such a gateway. Typically, the job of gateway machines is to perform some kind of efficient packet routing or request handoff, using techniques such as distributed packet rewriting [11, 81], or locality-aware request distribution [147], for example.

The approach to be employed in this research is to pseudo-randomise the name space across the nodes in the storage cluster by means of a cryptographic one-way hash function such as MD5 or SHA on the URN. The MD5 hash function will hash an arbitrary string to a 128-bit quantity with low probability of collision. This 128-bit quantity will represent the *object ID* of the digital object named by the URN, and can be considered a primary key for all access to the object. We can see the MD5 as a way of randomising a potentially highly-correlated initial key space (URNs with likely highly clustered prefixes) to a uniformly distributed one.

SDDS research has usually focused on minimising the number of servers required for a given file, and only utilising new servers when load dictates. In our case, our servers are dedicated to storage, and so not to utilise all of those available would be inefficient. In practical terms, this means setting the initial number of buckets to the number of servers in the storage cluster (or to a subset earmarked for metadata storage, should we wish not to use all of the disks in the cluster for metadata storage). The problem, then, involves what happens when new disks are added to the system, or existing disks are removed. This question is addressed by the splitting and merging policy employed.

SDDSs are designed to grow and shrink the file according to load. Growing involves splitting an existing bucket, transferring a portion of the existing data to the newly created bucket. The split policy has significant impact on the performance of the SDDS. Existing split criteria revolve around either a bucket overflowing (this presumes a fixed size bucket), or the bucket's utilisation exceeding some split threshold (e.g., between 70% and 100%). Splitting can be undertaken concurrently, and with or without the need for a designated split coordinator [120].

Because our storage servers are almost certainly heterogeneous, a fixed split criterion is undesirable for all. The load split threshold should be computed as a function of total space utilisation in the server. (This implies that each server manages logically variable sized buckets.) Vingralek *et al.* [198] examine a SDDS with emphasis on server load management. They also use an interesting indirection between bucket numbers and server numbers. Not only does this allow multiple logical buckets to be handled by the same server, but also it allows load to be managed by bucket *migration* as well as bucket *splitting*.

When a new disk is added to the cluster, we can choose to split one or more existing buckets to distribute load onto the new disk, or simply migrate existing buckets from multiple-bucket servers based upon current individual server load. When a disk is explicitly removed from the cluster, we can either migrate its buckets onto remaining servers, or else trigger a merge (the opposite of splitting) of its buckets.

It is important to note that this distribution of the name space across the servers in a cluster implies the hashing of an object (based upon its URN) to a "home server" for that object. The home server of an object is responsible for maintaining the metadata associated with that object. Depending upon the object, its home server also may manage the storage associated with the object. We *decouple* metadata storage from object storage. The primary reason behind this is that the metadata can be used to give valuable hints as to the most efficient location and layout for an object, and also the fact that some objects may be unable to reside within a single storage server.

As well as managing the metadata of an object, we also assign the important task of undertaking the *scheduling* of the I/O transfer of an object to its home server. In this way, as well as distributing the name space over the storage nodes, we also distribute the request and buffering workload. It is our thesis that the uniform randomisation of the URN name space will also uniformly randomise the workload, especially as load increases.

For the purposes of this research, we will not address encoding for fault tolerance, and assume a reliable communications and server infrastructure.

5.6 Exploiting Object Metadata

We described in the previous section an approach to distribute the name space over the storage cluster in a scalable fashion. In fact, this actually distributes the *object metadata* and object retrieval workload over the storage cluster. It is another of our central contentions that object metadata can be exploited to improve the layout and performance of digital objects.

In our scheme, we decouple metadata handling and data storage. That is, the layout and I/O of an object can be wholly independent of how its metadata is handled, and the metadata contains information as to the disposition and retrieval method of the object to which it applies. We can see this as an explicit separation of policy and mechanism for object storage. In effect, we view storage I/O methods as ones that can be overloaded on a per-object basis.

A storage system performs best when it is tuned to its workload. However, file systems usually are built under static assumptions and do not adapt well to the objects they hold. Furthermore, much file system design has been undertaken under the assumptions of small disks, small memories, and limited processing power. This has lead to an emphasis on maximising space utilisation and has limited file allocation strategies (which must execute under limited CPU resources). The much larger disks of today, and high collective processing power of a cluster of "active disks" allows us to be much more adventurous in storage management.

File system traces over the years point to the fact that most files are read, not written, and that most I/O tends to be whole file I/O. Furthermore, many new files are deleted shortly after creation, and most long-lived files tend to lie dormant. The "read-mostly" aspect is particularly true of digital libraries, in which new object creation is a considered act due to the explicit cataloguing that accompanies it. There are few available usage traces of extant digital libraries, though indicators of digital library workloads may be inferred by the behaviour of large scale distributed data repositories such as the World Wide Web, and various audio and generic peer-to-peer file sharing protocols. These systems are overwhelmingly read more than they are written, in that the ratio of object reads to writes is very high. This asymmetry is recognised to the point of the introduction of communications systems such as Asymmetric Digital Subscriber Line (ADSL) which model the observed phenomenon that most clients download far more data than they upload.

Another significant observed impact on performance is to tune the block size to the object being stored. Larger block sizes give better performance for large files because they mitigate the tremendous impact that head seek movement has upon overall transfer time. Traditionally, smaller block sizes have been used out of a desire to minimise wasted disk space due to partial block usage. With larger disks and higher-bandwidth controller interfaces, head movement rather than disk space is likely to become a more precious commodity. Clearly, the trend is towards larger block sizes.

Although it is not often tracked, file system traces that have attempted to track the type (semantics) of a file and the way it is accessed have found a strong correlation between the two. Except for specialised types of file servers, such as multimedia servers, file systems often cater to a "one size fits all" model. We recognise that individual file semantics can have enormous performance implications, and that knowing the semantics and past history of a file can offer valuable hints as to its performance optimisation.

We will use the information provided by digital object metadata to track not only its high-level semantic information, but also to store and help infer its low-level storage disposition. We also will consider access history and performance statistics to be part of a digital object's metadata, even though normally only a fraction of this information is explicitly catalogued (typically the object's creation date). Hence, in the "bucket" view of metadata we adopt here, as discussed in Section 4.4, the object layout and performance information become two elements of the bucket maintained by the system.

Buckets are accessed abstractly in an object-oriented fashion, in that all access to the bucket contents is via interface methods provided by the bucket. The same given method may behave differently according to the contents of the bucket, i.e., the methods are inherently overloaded by the elements of the bucket, and a general mechanism is provided to enable the exported methods to be enumerated and hence used.

The storage system can take advantage of this abstract model by implicitly overloading the storage and retrieval of each element of the bucket. If not explicitly overloaded, a *de facto* global method can be used. Alternatively, the element's type can be used to obtain a storage and retrieval method for that type of object. Finally, for elements that do not conform to type, or that require explicit handling, the element's storage and retrieval method can be overloaded individually.

We can thus view the access methods as belonging to an abstract class hierarchy according to the elements, and the storage metadata as providing input data to those access methods.

In this research, we will focus on the following tunable storage parameters: block size and striping threshold. We will be particularly interested in block size and when it is worthwhile striping a file across more than one storage server.

Historical studies have shown that most requests are whole-file sequential reads of small objects. Ideally, then, the goal should be to service an entire request within a single node. This maximises the parallelism of the entire storage cluster, because a request for a given object does not tie up (involve the use of) more than one disk head assembly, and disk head movement is the major limiting factor on disk performance. Historical studies also show, however, that most bytes stored are concentrated in relatively few objects. A way of achieving a more equitable node utilisation, therefore, would be to stripe large objects over larger (less utilised) storage nodes in preference to smaller ones. This would not allow larger objects to monopolise smaller storage nodes, but implies a more careful placement strategy when an object breaches the "striping threshold," or the point at which it should be stored across more than one node.

Fortunately, the aggregate computing power available in a storage cluster, combined with the self-management of the active disk approach, assist greatly in placement of large objects. Several strategies are possible. One, pursued here, is for each storage node to maintain statistics on its own utilisation, as well as a rough image of the utilisation of the rest of the storage cluster. By "rough image" we mean that the utilisation need not be accurate, only approximately accurate. Nor do we imply that it necessarily be complete: it may consist only of knowledge of a node's "neighbourhood" (as determined by network topology, say), or of a cache of recent responses to utilisation queries. Utilisation need not be solely in terms of storage (space) utilisation. It may also factor in the hardware characteristics of the storage node, e.g., the disk and CPU speed, amount of RAM, and so on. Abstractly, utilisation is a measure of load, weighted towards storage space, and each node is free to advertise its own utilisation according to its own local policy.

Based upon utilisation a storage node needing to stripe an object across several nodes will solicit the assistance of sufficient extra nodes to store the striped portions depending upon the chosen stripe size (decided based upon the object type), and the utilisation knowledge this node has of the other nodes in the cluster. Such dispersal of an object is partly a contract between the "home node" of an object and the other nodes across which it is striped. Incomplete knowledge of true utilisation of other nodes will mean each node will operate an *admission policy* for striped foreign objects, and, necessarily, there will be a *negotiation phase* as each node proferred a portion of an object being striped will decide whether or not to accept it. (Such a solicitation/negotiation phase is ripe for investigation of possible strategies, but we will leave this largely for future research.)

Utilisation measures advertised by a node will be normalised by the node to a number between 0 and 1. The makes it easier for a node to compare a set of returned utilisation figures. However, it also is useful for probablistic load balancing. By that, we mean a node can use the utilisation as a probability of rejecting a request to store foreign data on that node. It can also use the utilisation as the probability of ignoring a request to broadcast its current load (and hence lower the probability of it being asked to store further foreign objects). Unfortunately, using the utilisation load as a probability of responding to broadcasts directly has the disadvantage that most nodes will respond when the storage cluster is largely empty. We can combat this somewhat, for example, by thresholding the probability of ignoring a request to some minimum value, so as not to burden the network. So, the ignore probability could be set at $\max(t, u)$, $0 \leqslant t, u \leqslant 1$, where t is the floor threshold just mentioned, and u is the node utilisation. A refinement would be to have t decay from an initial elevated threshold value based upon the elapsed time since the last reply to a broadcast load query.

We will develop a metadata set for storage layout and retrieval purposes that encompasses both disposition tracking and performance history.

5.7 Improving Performance in a Read-Mostly Repository

Objects in a read-mostly environment are read much more than they are written. Typical engineering/office environments have measured an 80:20 ratio of reads to writes for normal file systems [144]. (Even then, a good percentage of writes are accounted for by short-lived objects.) It is highly likely that digital library object repositories will exhibit an even higher ratio of reads to writes, and that objects deposited will tend to be longer-lived.

When an object is written in a file system, a decision about its placement must be made under relatively tight time and resource constraints. Rarely is the object's disposition subsequently reassessed, except at the explicit direction of performance improvement tools run by system administrators. Most file system reorganisation is directed towards reducing internal fragmentation: some file systems provide user-level defragmentation tools; for others, the recourse to fragmentation is to restore the file system's contents anew from backup. Other file systems—notably the family of log-structured file systems—feature an asynchronous "cleaner daemon" that garbage collects the file system to ensure free segments are available for writes.

With long-lived, read-mostly objects in a storage environment with relatively large amounts of computing resources available, it seems worthwhile to take greater effort in determining a good storage disposition of an object that is newly deposited. It is also worthwhile to use the available aggregate computing resources to monitor and optimise the storage of the repository contents.

The major question regarding asynchronous performance optimisation involves its impact on the I/O performance of the storage system. Seltzer *et al.* [172] report the deleterious performance effect of the LFS cleaner daemon under high utilisation. It is important, therefore, that any background optimisation process operate only under periods of low load, and be able to checkpoint and restart its work.

Because our namespace approach fosters the notion of a "home" server for an object, and also because our storage approach is object-based rather than block-based, this encourages local, decentralised, object-based optimisation rather than a central global optimisation strategy. Active disks and NASD promote local management of storage objects, and are naturally decentralised at the object level.

Some objects, however, are spread over more than one storage server, so there is a need for some cross-server optimisation. Such optimisation requires some knowledge of state (load) at other servers. The namespace SDDS estimates global state in terms of server load, so that offers some input. In addition, there are efficient distributed global state techniques for optimising within a distributed environment. Also, Arpaci-Dusseau [8] discusses implicitly controlled systems in which the cooperating entities do not explicitly exchange control or state information but infer it from various sources. Her particular focus is on implicit job scheduling within a distributed system.

Our read-mostly assumption offers several possibilities for performance optimisation. Firstly, we can treat writes (object deposit) as a special case and forward such objects to a staging area optimised for writing (as discussed previously). The servers comprising the staging area would migrate out those objects that age sufficiently, or which consume sufficiently large space resources to impact the performance of the staging area.

Secondly, we can use ongoing access statistics for objects to determine whether a given object is a good candidate for optimisation. A successful measure of predictive usage of a file object is its "temperature," which is a measure of how often it has been accessed over a given time frame. The more accesses, the "hotter" an object will become. So, object temperature can be used as a threshold trigger to attempt to optimise a given object (e.g., by defragmenting it), or a median temperature of all the objects on a storage server may be used to trigger optimisation of the storage server itself.

As discussed previously, the namespace is dominated by small files. There is an incentive, therefore, to keep the initial portion of a given object local to its home storage server (especially as most reads are sequential whole-file reads). This approach is often termed "inode stuffing," because the inode size is increased to be larger than needed (e.g., to the size of a standard data block), and the tail portion is used to hold the initial data of the file. This is a big performance win for small files because the file metadata and data can be transferred in a single transfer.

Inode stuffing can be used analogously in a cluster-based storage system in that it can provide a threshold at which data remain local to a node as opposed to being striped over other servers. The higher this threshold, the more local data will be. This will incur less network usage (fetching striped portions from remote servers), but will involve more local disk activity.

Given that an object's metadata is maintained by its home node, there is good reason to store the initial data at that same node, too. In the case of small objects, this will improve performance, as both can be blocked together, as in inode stuffing. However, we will investigate how much initial data stored at a node (the off-node threshold) affects performance. Modern high-speed networking versus slow disks has made networking overheads perhaps less of an issue. Indeed, network RAM [58] is predicated on the notion that it is now quicker to page data across a network to another system than to page it to local disk due to the orders of magnitude difference in operational speed of the two media.

Another issue greatly affecting performance according to file system trace studies is the effect of block size on I/O performance. As mentioned previously, larger block sizes favour better performance, the main disadvantage being storage space lost to fragmentation. Historic studies show that most files are small, but most bytes belong to large files. This suggests several strategies for choosing and adapting block sizes.

Firstly, we can favour large files by choosing a large block size. This has the disadvantage of wasting space for small files, and also of being a static scheme. The impact on small files could be mitigated by batching smaller files into the larger blocks, e.g., similar to the approach of file systems such as ReiserFS [191] that are optimised for small files yet still perform well for large files.

Secondly, we could choose a simple strategy of having two block sizes: one for objects wholly local to the storage node, and the other—much larger—for objects that have been striped across several servers. Because big objects will be striped, the larger block size will benefit their performance. Similarly, the smaller block size for the smaller, local object will benefit performance by decreasing space lost to internal fragmentation.

The disadvantage of this approach is chiefly that of the previous technique in that it is essentially static. However, it is also more difficult to implement in storage servers consisting of a single disk (e.g., an "intelligent disk"). If the larger block size is not some exact multiple of the smaller one, it becomes difficult to manage allocation within the same disk real estate. Generally, the disk must be partitioned into multiple areas employing a different block size in each area (e.g., two partitions: one using the smaller and one using the larger block sizes). Choosing how much to allocate to each partition is difficult, as partitions are not easily resized once created.

Storage servers that consist of multiple disks per server node fare better, however. Each disk can be used with a different block size. Once again, the ratio of local (small) to remote (large) objects is not easy to control, but in terms of the global cluster usage the effect is not so serious as nodes can "lease" the extra remote space for use by other servers. If most bytes reside in large files, space in the large block sized area will be used effectively by other nodes in the cluster. The question becomes one of sizing the small block sized storage, which is used by small objects and metadata. However, the SDDS used to allocate the namespace (metadata) will tend to compensate for undersizing or oversizing any particular storage node by factoring it into the splitting or merging of the items stored at that node.

The previous two approaches have the advantage of being relatively simple to implement and require relatively few resources. The greater available dedicated processing power available in the storage node suggests something more ambitious should be attempted, and points to a third, more general approach: use variable sized blocks.

In an object-based, read-mostly system where I/O tends to be whole-file I/O, using large, contiguously allocated disk blocks is clearly advantageous (see Section 3.3.3). We shall thus adopt an extent-based layout approach. We shall use contiguity and semantic file type as the criteria for choosing the block size. As stated before, a threshold must be chosen at which to begin striping an object so as not to monopolise local storage space. Once again, this threshold can be decided according to object file types, or may be determined according to the local server needs.

5.8 Summary

Analysis of digital libraries object repository requirements and storage system research has led us to develop an architecture which meets the challenges and requirements of the former by applying the best appropriate practices of the latter.

Digital objects are naturally object-based, not block-based, and have rich hints available via explicitly catalogued metadata. By augmenting this explicit metadata with workload statistics, we can add a new, semantic-driven input to storage access and layout policy and mechanisms. This is not present in conventional storage systems, which, aside from some application-specific domains, treat files as neutral linear byte streams and thus miss a potential rich source of performance optimisations.

Our target architecture is a cluster of intelligent disk nodes that supports a repository access protocol. Higher-level digital libraries functions are assumed to be provided elsewhere.

We propose to use scalable hashing with randomisation as a means of handling both the name space and workload distribution across the nodes of the cluster. Scalable hashing addresses the problem of quickly and scalably locating an object within the cluster, and allowing for the growth or shrinking of the cluster with minimal re-mapping of existing objects to other disks.

Disk seeks are an expensive commodity in any disk access, because they are limited by physical laws that severely limit their ongoing performance improvement. However, most objects are small in relation to the collection, but most bytes are concentrated in relatively few objects. We thus propose to manage the tradeoff between minimising seeks as a threshold decision of when to stripe an object across more than one disk node. This serves two purposes: it allows even use of space across all disk nodes (and prevents large objects monopolising space on individual disk nodes), and it permits high-bandwidth transfers of large objects by involving more disk head assemblies. The size threshold at which to go "off node" is a matter for investigation. In a simple fashion, it could be a fixed size for all objects. In a more complex implementation, it could be based upon the semantic type of the object, as identified by the object's metadata or access history.

Archival digital library collections are predominantly read-mostly. Read mostly objects benefit from large transfer sizes, as they are overwhelmingly accessed in a whole-object sequential fashion. We propose to use a variable block size extent-based allocation scheme to improve I/O performance by making objects more sequential on disk. This will minimise seeking, and improve performance.

Our proposed architecture targets directly the needs of digital libraries object repositories. Beyond all else, if offers a platform for future experimentation tied directly to that important domain.

6 Research Plan and Expected Contributions

The proposed research centres around the theme of designing a scalable cluster-based digital library object repository based upon digital library collection characteristics. The emphasis is on long-term scalability in an implicitly heterogeneous storage environment. Unlike conventional approaches that are block-based, our emphasis is on object-based storage.

No attempt will be made to implement a working production prototype. Instead, the emphasis will be on the simulation of the proposed concepts and techniques. Simulation enables the investigation of a much wider size and array of hardware and networking topologies than is possible using actual hardware. This is especially true when investigating scalability, which involves large amounts of hardware resources. Tools such as SynRGen [54] and DiskSim [63] can greatly assist in the simulation process.

First, several real digital library collections will be analysed to obtain collection statistics to use when generating synthetic workloads. One collection will be the repository of electronic theses and dissertations at Virginia Tech. Although this collection does not have a large amount of multimedia associated with it, it does have good metadata, which is important to our studies. It is also essentially a bucket-based repository in that each thesis or dissertation equates to a bucket.

Another good digital library to be examined is the Perseus digital library at Tufts [41]. This is a digital library centred around the humanities that has been in development for over ten years now. It contains eight main collections containing a wide range of material both geographically and in time, including collections on the Classics; papyri; English Renaissance; London; California; the Upper Midwest; Chesapeake; and Tufts history. The collections include both texts and images, as well as geographical information in some cases. Some documents have both text transcriptions and page images. The content of the digital library is both large and diverse, though, by nature, somewhat lacking in digital video and audio.

The VARIATIONS digital library at Indiana University [52] contains over 7000 audio titles, and is a good example of a large collection of digital audio. Its successor project, the Indiana University Digital Music Library, recently was launched. It seeks to synthesise many different types of media related to music including digital audio, music notation files, score images, and MIDI files. This presents the possibility of a more bucket-based approach than the current VARIATIONS system, but lacks an extant mature collection due to the early stages of the project.

The available digital library collections will be analysed with respect to the criteria expounded in Section 5.1. It is important to ascertain the distribution of object sizes and accesses, both globally and according to semantic file type. Summary statistics will provide input as to choice of striping threshold for various semantic file types.

Using the collection analysis, we will then generate synthetic collections in order to "scale up" the problem. The synthetic collections will mimic the characteristics of the exsiting ones, but be orders of magnitude larger. We intend to simulate collections in these sizes: 100 GB, 1 TB, 100 TB, and 1 PB. Each collection will be simulated on clusters of varying sizes: 100 nodes, 1000 nodes, and 10000 nodes. In addition, each cluster configuration will employ disk sizes in these random intervals at each node: 5–15 GB, 20–50 GB, and 50–150 GB. (Random disk sizes per node are intended to simulate the heterogeneity that comes over time. Note that some of the synthetic collection sizes will not fit in some of the cluster configurations, e.g., 10 TB will not fit in a 100 node cluster whose maximum node capacity is 15 GB.) We will also simulate two speeds of interconnection fabric: "slow" 10 Mbps and "fast" 1 Gbps. Experimental results will be analysed using R [84].

When generating synthetic collections we will use the characterisation of existing file types and sizes as a basis, and then scale up the numbers of those files and objects (with random perturbation) to achieve larger collection sizes within the desired simulated ranges. It is the collection size that will be scaled up, not the sizes of individual object types, or the percentage with which that object makes up in the collection as a whole. We will randomly sample objects from existing collections until the desired collection size is attained in each case, e.g., randomly sample until 1 TB is amassed for simulating a 1 TB collection.

Using the synthetic collections as our repository contents, we will simulate a digital library object repository storage cluster using the ideas described previously. Figure 1 shows the general architecture. In the figure, "D" represents disk nodes within the storage cluster, and "G" represents gateway nodes that handle client requests from external networks. The disk storage

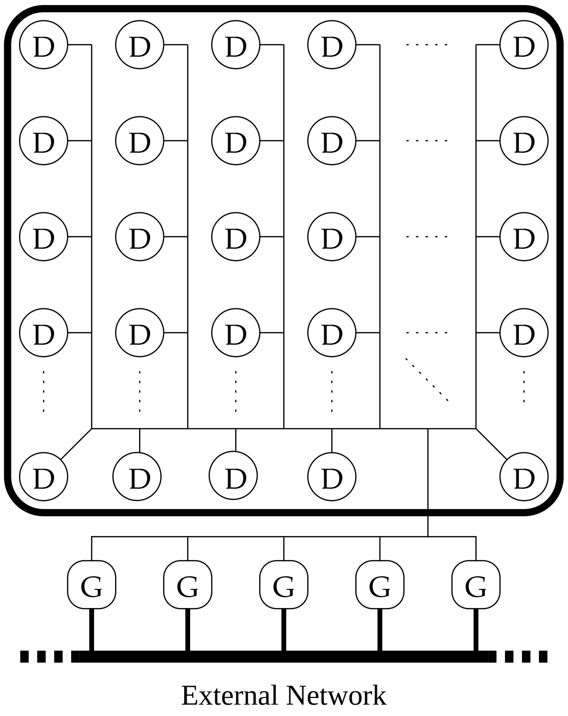

Figure 1: Architecture of digital library object repository storage cluster

nodes exist on a private network, not addressable by external networks. The gateway nodes are thus needed to facilitate a "hand-off" of requests to the disk nodes within the cluster. Their functionality is lightweight, and analogous to that of front-end request processors in scalable WWW server clusters.

External clients accessing the digital library object repository are assumed to support a repository access protocol of the type proposed by Kahn and Wilensky [88]. Global URN resolution is performed at the client by some external, to locate the actual repository at which the object to which the URN refers is stored. All objects in the repository are accessed via their URN handle. A typical access of a digital object from the repository by an oblivious client proceeds as follows:

1. The client issues an ACCESS_DO request to the repository, with the digital object's URN handle as argument;
2. If the handle is resolvable to the repository, the repository replies with a reference to the primitive disseminator/bucket associated with the object;
3. The client requests a list of packages/DataStreams contained in the digital object via the reference just returned;
4. The repository replies with a list of packages;
5. The client requests a list of elements contained in a given package from the repository;
6. The repository responds with a list of elements contained in the package;
7. The client requests a list of supported methods of a given element desired to be accessed;
8. The repository responds with the supported methods of the element;
9. The client invokes the desired access method of the element at the repository;
10. The repository returns a byte stream to the client produced by invoking the access method

Note that the access sequence just described can be collapsed tremendously depending upon the knowledge the client possesses about the content of the digital object or the collection. For example, if a client knows that a certain access method is supported on an element of a package, known *a priori*, then the client can invoke that method on that element of that package of the object directly. In this way, much of the worst-case "discovery phase" of accessing a digital object can be eliminated. We will assume our clients are nonoblivious, and are driven by a higher-level digital library application layer that is cognisant, somewhat, about the collections it serves. Also, although the bucket-based approach allows for programmatic disseminators, we shall restrict our simulation to the access of static content (i.e., disseminators that return byte streams of stored data, not of data generated algorithmically on-the-fly).

The first, and most important aspect to simulate is the effectiveness of the namespace and workload distribution hashing scheme. It is important to verify that this balances load adequately in the limit. We will measure the effectiveness according to three criteria: uniformity of name distribution; equitability of byte distribution; and equitability of workload distribution.

The uniformity of name distribution is assessed by measuring the distribution of the names of all stored objects across all nodes in the storage cluster. Ideally, for a storage cluster of k nodes storing a total of n objects, each node should handle the stewardship of $\frac{n}{k}$ objects. The pseudo-random hashing of object names should ensure good uniform distribution across the total cluster. This is reinforced by the load-balancing nature of the LH* hashing scheme.

The equitability of byte distribution is analogous to that of name distribution. Again, balancing is determined based upon node utilisation relative to the entire cluster. In terms of byte storage, two possible measures manifest immediately: for system of k nodes, each node could store $\frac{1}{k}$th of the total bytes, or operate at p% of its total storage capacity. In a storage cluster in which each storage node has the same capacity, both measures are in fact the same. In a heterogeneous environment, though, they are not. The former measure will have smaller disks operating closer to being full, and larger ones closer to being empty. This is not necessarily a bad thing in terms of access performance, because, naively speaking, each disk has only a single head assembly, and head movement is the most precious commodity for disk performance. In a uniformly distributed workload, $\frac{1}{k}$th byte utilisation at each node equates to $\frac{1}{k}$th of the object space at that node. A p% utilisation strategy is good only if *proportionally bigger objects* are being stored at storage nodes with higher capacity.

Equitability of workload distribution is measured by the proportion of time a disk node spends servicing requests, i.e., the amount of time it is "busy." A well-balanced cluster should have each storage node active the same percentage of time.

To test load handling under various cluster configurations as described above, we also will need several synthetic workloads. As mentioned, some of the collections have available access logs that can be used as a basis for generating synthetic workloads. The important characteristics we want to scale up from these are the proportion of objects within the collection that are accessed frequently, the distribution of accesses by object types, and the distribution of times between sucessive accesses of the same object.

As well as using synthetic trace workloads to simulate real-world conditions, we will also use SynRGen [54] to generate user-driven workloads. SynRGen can be used to model classes of users, and can algorithmically generate workload traces. A particular user or access behaviour can be modelled in SynRGen, and the output from the SynRGen run used as an input workload to the simulation. In this way, the numbers of users of the repository can be scaled easily, and different access workloads investigated.

The next task will be to simulate the content management and dispersal aspect. Particular focus here is on the effect of the striping (local versus network) policy. A major goal is to identify whether the bottleneck lies within the nodes, or in the interconnection network. Although we will not implement the on-disk layout mechanism in detail in the cluster-level simulation, we will simulate its worst-case and average-case guarantees of contiguity to estimate seek and transfer times.

As mentioned previously, we will not address the issues of reliability, fault-tolerance, authentication, and security in this research. They are left as topics for future examination. However, the "closeted" nature of a private storage area network cluster does mitigate the ill effects of discounting the areas of security and authentication somewhat.

Table 1 shows the dependent variables used and independent variables measured in our simulation experiments. The access latency is the amount of time between a client issuing a

Dependent variables

Name	Levels
Collection size (TB)	0.1, 1, 100, 1000
Disk nodes	100, 1000, 10000, 1000000
Disk size (mean GB)	10, 40, 100
Network speed	10 Mbps, 1 Gbps
Striping threshold (KB)	TBD
Workload	TBD

Independent variables

Access latency
Disk bandwidth usage
Network bandwidth usage
Node CPU utilisation
Node disk utilisation
Number of network messages

Table 1: Simulation variables

request and receiving the first result byte. Disk bandwidth usage is the percentage of time spent transferring actual data against the total amount of time spent on a disk operation. Hence, if a disk transfer operation spends 9 ms seeking and 1 ms transferring data, the operation is utilising only 10% disk bandwidth; if it spends 9 ms seeking and 81 ms transferring data in fulfilling a disk transfer, it is utilising 90% disk bandwidth.

Network bandwidth usage is the percentage of available network capacity used, analogous to disk bandwidth usage. Node CPU usage is the percentage of time a node spends actively processing requests (from clients or other cluster nodes) versus the time it spends idle awaiting work to do. Node disk utilisation is the proportion of disk space allocated in a node against the total available disk space in that node. (So, if a node currently uses 5 GB of a 20 GB disk, it is at 25% node disk utilisation.) Number of network messages is the number of control and transfer messages sent over the network (internal and external) in the fulfillment of a client request.

Our experiments will revolve around a simulation of a digital libraries object repository storage cluster based upon techniques and principles described in Section 5. The simulation will be instrumented so that its performance and functionality may be measured and modified.

Series 1: Object/Workload Distribution

Vary: Collection size; number of nodes

Measure: Node CPU utilisation; node disk utilisation

Objective: Test equitable distribution of object metadata and client workload across cluster

Series 2: Striping Policy

Vary: Collection size; number of nodes; striping threshold/policy

Measure: Node CPU utilisation; node disk utilisation; network bandwidth utilisation

Objective: Measure the effect of striping threshold on disk and network usage

Series 3: Metadata-driven Layout

Vary: Collection size; number of nodes; metadata layout policy

Measure: Node CPU utilisation; node disk utilisation; network bandwidth utilisation; access latency; disk bandwidth

Objective: Determine the effects of metadata-driven layout specialisation

Table 2: Series of experiments for simulation

The simulation is intended as a "test bed" for the ideas expressed herein.

We intend to run three main series of experiments to assess the impact of various key ideas put forth in this document. This series is listed in Table 2. The first two series of experiments are largely static tests of storage distribution, as described previously. The third series is intended to explore the effects of using metadata hints on layout policy. Here, we will vary the striping threshold on an object type basis. For example, a different striping threshold will be used for MPEG files than for PDF files, instead of using the same for both.

Assessing the relative performance of the system is somewhat difficult given the scaling and domain-specific nature of the area. When varying the number of nodes, it is important to ensure that it scales smoothly and is free of "hot spots" or overloaded nodes. Performance will be assessed relative to the maximum theoretical, to make it somewhat independent of actual given parameter values. Chen and Patterson [33] offer possible guidance on designing self-scaling benchmarks that are meaningful across platforms.

There are several expected contributions of this research. Firstly, it will provide important metric data in a field that, to this date, has largely ignored it. (Digital libraries "live and breathe" on storage, yet extant metrics work has focused on user interfaces, queries, relevance, recall, and precision. Collection storage analysis is conspicuously missing.)

Secondly, and importantly, it will address storage from a digital libraries point of view. It has long been recognised that digital libraries are not file systems, yet at a low level they reside on them. Not only does this rob them of performance opportunities, but it forces digital library functionality through the needle eye of a file system API. This research will provide a digital object, respository-based approach to digital library storage.

Thirdly, the research addresses the critical need for scalability and evolution in digital library storage. With content growing at an alarming rate, storage architectures must meet that growth. This necessitates a heterogeneous, incrementally growing storage architecture: a demand this research addresses.

Finally, another significant contribution of this research is a flexible handling of metadata at the storage level. In addition, it contributes an incorporation and usage of active storage performance data in ongoing archival operations, along with an exploitation of semantic content in improving storage performance.

References

[1] Acharya, A., Uysal, M., and Saltz, J. Active disks: Programming model, algorithms and evaluation. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS) VIII* (Oct. 1998), ACM, pp. 81–91.

[2] Adamic, L. A., and Huberman, B. A. The Web's hidden order. *Commun. ACM 44*, 9 (Sept. 2001), 55–60.

[3] Alon, N., and Luby, M. A linear time erasure-resilient code with nearly optimal recovery. *IEEE Transactions on Information Theory 42*, 6 (Nov. 1996), 1732–1736.

[4] Alvarez, G. A., Borowsky, E., Go, S., Romer, T. H., Becker-Szendy, R., Golding, R., Merchant, A., Spasojevic, M., Veitch, A., and Wilkes, J. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.* (2001). to appear.

[5] Anderson, D. Consideration for smarter storage devices. Presentation given at the National Storage Industry Consortium (NSIC) Network-Attached Storage Device working group meeting, June 1998. Available from `http://www.nsic.org/nasd/`.

[6] Anderson, T. E., Culler, D. E., Patterson, D. A., and the NOW team. A case for NOW (Network Of Workstations). Tech. rep., University of California at Berkeley, 1994.

[7] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. Serverless network file systems. Tech. rep., University of California at Berkeley, 1995.

[8] Arpaci-Dusseau, A. C. *Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems*. PhD thesis, University of California at Berkeley, 1998.

[9] Arpaci-Dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. Cluster I/O with River:

Making the fast case common. In *Proceedings of IOPADS '99: Input/Output for Parallel and Distributed Systems* (Atlanta, Georgia, May 1999).

[10] Asami, S., Talagala, N., and Patterson, D. A. Designing a self-maintaining storage system. In *Proceedings of the 1999 IEEE Symposium on Mass Storage Systems* (1999).

[11] Aversa, L., and Bestavros, A. Load balancing a cluster of web servers using distributed packet rewriting. Tech. Rep. 1999-001, Boston University, 6 Jan. 1999. `urn:hdl:ncstrl.bu_cs/1999-001`.

[12] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. Measurements of a distributed filesystem. In *Proceedings of Symposium on Operating Systems Principles (SOSP) 13* (Oct. 1991), pp. 198–212.

[13] Barclay, T., Gray, J., and Slutz, D. Microsoft TerraServer: A spatial data warehouse. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Austin, Texas, USA, May 2000).

[14] Baru, C., Moore, R., Rajasekar, A., and Wan, M. The SDSC Storage Resource Broker. In *Proceedings of CASCON '98* (Toronto, Canada, 30 Nov.–3 Dec. 1998), IBM Conference. Available via `http://npaci.edu/DICE/Pubs/srb.ps`.

[15] Barve, R., Shriver, E., Gibbons, P. B., Hillyer, B. K., Matias, Y., and Vitter, J. S. Modeling and optimizing I/O throughput of multiple disks on a bus. In *Proceedings of the 1999 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Altanta, Georgia, USA, May 1999), pp. 83–92.

[16] Beck, M., and Moore, T. The Internet2 Distributed Storage Infrastructure project: An architecture for Internet content channels. *Computer Networking and ISDN Systems 30*, 22-23 (1998), 2141–2148.

[17] Benner, A. *Fibre Channel: Gigabit I/O and Communications for Computer Networks*. McGraw-Hill, New York, NY, 1996.

[18] Bennett, J. M., Bauer, M. A., and Kinchlea, D. Characteristics of files in NFS environments. In *ACM Symposium on Small Systems* (1991), pp. 33–40.

[19] Berenbrink, P., Brinkmann, A., and Scheideler, C. Design of the PRESTO multimedia storage network. In *Proceedings of the Workshop on Communication and Data Management in Large Networks (INFORMATIK 99)* (Oct. 1999).

[20] Berners-Lee, T., Fielding, R. T., and Masinter, L. Uniform resource identifiers (URI): Generic syntax. RFC 2396, Aug. 1998. `http://www.ietf.org/rfc/rfc2396.txt`.

[21] Berners-Lee, T., Fielding, R. T., and Nielsen, H. F. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, May 1996. `http://www.ietf.org/rfc/rfc1945.txt`.

[22] Berners-Lee, T., Masinter, L., and McCahill, M. Uniform resource locators (URL). RFC 1738, Dec. 1994. `http://www.ietf.org/rfc/rfc1738.txt`.

[23] Berson, S., Muntz, R., and Wong, W. Randomized data allocation for real-time disks. In *COMPCON '96* (1996), pp. 286–290.

[24] Borowsky, E., Golding, R., Jacobson, P., Merchant, A., Schreier, L., Spasojevic, M., and Wilkes, J. Capacity planning with phased workloads. In *WOSP '98* (Santa Fe, New Mexico, USA, Oct. 1998).

[25] Brandt, C., Kyriakaki, G., Lamotte, W., Lüling, R., Maragoudakis, Y.,

Mavraganis, Y., Meyer, K., and Pappas, N. The SICMA multimedia server and virtual museum application. In *Proceedings of the Third European Conference on Multimedia Applications, Services and Techniques* (1998), pp. 83–96.

[26] Brodowicz, M., and Johnson, O. PARADISE: An advanced featured parallel file system. Tech. rep., University of Houston, 1998.

[27] Calzarossa, M., and Serazzi, G. Workload characterization: A survey. *Proceedings of the IEEE 81*, 8 (Aug. 1993), 1136–1150.

[28] Cao, P., Lim, S., Venkataraman, S., and Wilkes, J. The TickerTAIP parallel RAID architecture. In *Proceedings of the 20th Symposium on Computer Architecture* (May 1993), pp. 52–63.

[29] Cao, P., Lim, S. B., Venkataraman, S., and Wilkes, J. The TickerTAIP parallel RAID architecture. *ACM Trans. Comput. Syst. 12*, 3 (Aug. 1994), 236–269.

[30] Chandy, J., and Reddy, A. L. N. Failure evaluation of disk array organizations. In *Proceedings of the International Conference on Distributed Computing Systems* (1993).

[31] Chen, P., Lee, E., Gibson, G., Katz, R., and Patterson, D. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys 26*, 2 (June 1994), 145–188.

[32] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. The Rio file cache: Surviving operating system crashes. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS) VII* (Oct. 1996), pp. 74–83.

[33] Chen, P. M., and Patterson, D. A. A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performance. *ACM Trans. Comput. Syst. 12*, 4 (Nov. 1994), 308–339.

[34] Chiueh, T. Trail: a track-based logging disk architecture for zero-overhead writes. In *Proceedings of the 1993 IEEE International Conference on Computer Design (ICCD '93)* (Cambridge, MA, USA, 3–6 Oct. 1993), pp. 339–343.

[35] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., Snir, M., Traversat, B., and Wong, P. Overview of the MPI-IO parallel I/O interface. In *Input/Output in Parallel and Distributed Computer Systems*, R. Jain, J. Werth, and J. C. Browne, Eds., vol. 362 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer, Amsterdam, 1996.

[36] Corbett, P. F., and Feitelson, D. G. The Vesta parallel file system. *ACM Transactions on Computer Systems 14*, 4 (Aug. 1996), 225–264.

[37] Corbett, P. F., Prost, J.-P., Demetriou, C., Gibson, G., Reidel, E., Zelenka, J., Chen, Y., Felten, E., Li, K., Hartman, J., Peterson, L., Bershad, B., Wolman, A., and Aydt, R. Proposal for a common parallel file system programming interface. Tech. rep., Department of Computer Science, University of Arizona, Tucson, Arizona, USA, 1996. Available via the WWW at `http://www.cs.arizona.edu/sio/api1.0.ps`.

[38] Courtright, W. V., Gibson, G., Holland, M., and Zelenka, J. RAIDframe: Rapid prototyping for disk arrays. In *SIGMETRICS '96* (May 1996), pp. 268–269.

[39] Coyne, R. A., and Hulen, H. An introduction to the Storage System Reference Model, version 5. In *Proceedings of*

the Twelfth IEEE Symposium on Mass Storage (Apr. 1993).

[40] COYNE, R. A., HULEN, H., AND WATSON, R. The High Performance Storage System. In *Proceedings of the Conference on Supercomputing '93* (Portland, Oregon, USA, 15–19 Nov. 1993), Association for Computing Machinery, pp. 83–92.

[41] CRANE, G., CHAVEZ, R. F., MAHONEY, A., MILBANK, T. L., RYDBERG-COX, J. A., AND SMITH, D. A. Drudgery and deep thought: Designing digital libraries for the humanities. *Commun. ACM 44*, 5 (May 2001), 34–40.

[42] CRESPO, A., AND GARCÍA-MOLINA, H. Archival storage for digital libraries. In *Proceedings of Digital Libraries 98* (Pittsburgh, PA, 1998), ACM, p. 6978.

[43] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the 2001 Symposium on Operating Systems Principles (SOSP)* (Banff, Canada, 21–24 Oct. 2001), Association for Computing Machinery.

[44] DANIEL, R. A trivial convention for using HTTP in URN resolution. RFC 2169, June 1997. `http://www.ietf.org/rfc/rfc2169.txt`.

[45] DANIEL, JR., R., AND LAGOZE, C. Distributed Active Relationship in the Warwick Framework. In *Proceedings of the Second IEEE Metadata Conference* (Silver Spring, Maryland, USA, 16–17 Sept. 1997).

[46] DAVIS, J. R., AND LAGOZE, C. A protocol and server for a distributed digital technical report library. Tech. rep., Cornell University, June 1994. `urn:hdl:ncstrl.cornell/TR94-1418`.

[47] DEVINE, R. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO)* (1993).

[48] DEVLIN, B., GRAY, J., LAING, B., AND SPIX, G. Scalability terminology: Farms, clones, partitions and packs: RACS and RAPS. Tech. Rep. MS-TR-99-85, Microsoft Research, Dec. 1999.

[49] DOUCEUR, J. R., AND BOLOSKY, W. J. A large-scale study of file-system contents. In *SIGMETRICS '99* (Atlanta, Georgia, USA, 29 Apr.–6 May 1999), pp. 59–70.

[50] DRUSCHEL, P., AND ROWSTRON, A. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)* (Elmau/Oberbayern, Germany, May 2001), pp. 65–70.

[51] DUBLIN CORE DIRECTORATE. Dublin Core metadata initiative. WWW page, 2000. `http://purl.org/dc`.

[52] DUNN, J. W., AND MAYER, C. A. VARIATIONS: A digital music library system at Indiana University. In *Proceedings of the Fourth ACM Conference on Digital Libraries* (Berkeley, California, USA, Aug. 1999).

[53] DURFEE, E., KISKIS, D., AND BIRMINGHAM, W. The agent architecture of the University of Michigan Digital Library. *IEE/British Computer Society Proceedings on Software Engineering 144*, 1 (Feb. 1997).

[54] EBLING, M. R., AND SATYANARAYANAN, M. SynRGen: An extensible file reference generator. In *SIGMETRICS '94* (Santa Clara, CA, May 1994), pp. 108–117.

[55] ECS DATA MANAGEMENT ORGANISATION. ECS Data Handling System. World Wide Web site, Nov. 1999. `http://edhs1.gsfc.nasa.gov/`.

[56] FIBRE CHANNEL INDUSTRY ASSOCIATION. Fibre Channel Industry Associa-

tion. World Wide Web page, 2000. `http://www.fibrechannel.org/`.

[57] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, Springfield, Virginia, USA, Apr. 1995.

[58] FLOURIS, M. D., AND MARKATOS, E. P. Network RAM. In *High Performance Cluster Computing*, B. Rajkumar, Ed. Prentice Hall, 1999, pp. 383–408.

[59] GAIT, J. Phoenix: A safe in-memory file system. *Commun. ACM 33*, 1 (Jan. 1990), 81–86.

[60] GANGER, G. R. Generating representative synthetic workloads: An unsolved problem. In *Proceedings of the Computer Measurement Group (CMG) Conference* (Dec. 1995), pp. 1263–1269.

[61] GANGER, G. R., AND PATT, Y. N. The process-flow model: Examining I/O performance from the system's point of view. In *Proceedings of the 1993 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (May 1993), pp. 86–97.

[62] GANGER, G. R., AND PATT, Y. N. Soft updates: A solution to the metadata update problem in file systems. Tech. Rep. CSE-TR-254-95, University of Michigan, 1995.

[63] GANGER, G. R., WORTHINGTON, B. L., AND PATT, Y. N. The DiskSim simulation environment version 1.0 reference manual. Tech. Rep. CSE-TR-358-98, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 27 Feb. 1998.

[64] GHANDEHARIZADEH, S., AND IERARDI, D. An algorithm for disk space management to minimize seeks. Tech. Rep. 95–612, Department of Computer Science, University of Southern California, Los Angeles, California, 1995.

[65] GHANDEHARIZADEH, S., IERARDI, D., AND ZIMMERMANN, R. Management of space in hierarchical storage systems. Tech. Rep. 94–598, Department of Computer Science, University of Southern California, Los Angeles, California, 1994.

[66] GHANDEHARIZADEH, S., IERARDI, D., AND ZIMMERMANN, R. An online algorithm to optimize file layout in a dynamic environment. Tech. rep., Department of Computer Science, University of Southern California, Los Angeles, California, 1995.

[67] GIBSON, G. A. *Redundant Disk Arrays: Reliable Parallel Secondary Storage*. PhD thesis, University of California at Berkeley, 1990.

[68] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A cost-effective, high-bandwidth storage architecture. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS) VIII* (Oct. 1998), ACM, pp. 92–103.

[69] GIBSON, G. A., NAGLE, D. F., COURTRIGHT II, W., LANZA, N., MAZAITIS, P., UNANGST, M., AND ZELENKA, J. NASD scalable storage systems. In *Proceedings of USENIX 1999* (Monterey, California, USA, 9–11 June 1999).

[70] GIBSON, T. J., AND MILLER, E. L. Long term file activity patterns in a unix workstation environment. In *Proceedings of the Fifteenth IEEE Symposium on Mass Storage Systems* (College Park, Maryland, USA, 23–26 Mar. 1998), pp. 355–371.

[71] GLADNEY, H. M., AND LOTSPIECH, J. B. Safeguarding digital library contents and users: Assuring convenient security and data quality. *D-Lib Magazine* (May 1997). `urn:hdl:cnri.dlib/may97-gladney`.

[72] Golding, R., Shriver, E., Sullivan, T., and Wilkes, J. Attribute-managed storage. In *Workshop on Modeling and Specification of I/O (MSIO)* (San Antonio, Texas, USA, 26 Oct. 1995).

[73] Gray, J., and Shenoy, P. Rules of thumb in data engineering. Tech. Rep. MS-TR-99-100, Microsoft Research, Advanced Technology Division, Dec. 1999.

[74] Grochowski, E., and Hoyt, R. F. Future trends in hard disk drives. *IEEE Transactions on Magnetics 32*, 3 (May 1996), 1850–1854.

[75] Hartman, J. H., Murdock, I., and Spalink, T. The Swarm scalable storage system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems* (Austin, Texas, USA, 31 May–4 June 1999), IEEE Computer Society, pp. 74–81.

[76] Hartman, J. H., and Ousterhout, J. K. The Zebra striped network file system. *ACM Trans. Comput. Syst. 13*, 3 (1995), 274–310.

[77] Haskin, R. L., and Lorie, R. A. On extending the functions of a relational database system. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data* (Orlando, Florida, USA, 2–4 June 1982), M. Schkolnick, Ed., ACM Press, pp. 207–212.

[78] Hirschberg, D. S. A class of dynamic memory allocation algorithms. *Commun. ACM 16*, 10 (Oct. 1973), 615–618.

[79] Holland, M., and Gibson, G. A. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS) V* (1992), pp. 23–35.

[80] Hu, Y., and Yang, Q. DCD—Disk Caching Disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd International Symposium on Computer Architecture* (Philadelphia, PA, USA, May 1996), pp. 169–178.

[81] Hunt, G., Nahum, E., and Tracey, J. Enabling content-based load distribution for scalable services. Tech. rep., IBM TJ Watson Research Center, Yorktown Heights, NY 10598, 1998.

[82] IEEE Storage System Standards Working Group. Mass storage reference model version 5. Tech. rep., IEEE Computer Society Mass Storage Systems and Technology Committee, 5 Aug. 1993. Unapproved draft.

[83] IETF Secretariat. Uniform Resource Names (URN) charter. World Wide Web page, Sept. 2000. `http://www.ietf.org/html.charters/urn-charter.html`.

[84] Ihaka, R., and Gentleman, R. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics 5*, 3 (1996), 299–314.

[85] International Business Machines. IBM Digital Library, 1998. `http://www.software.ibm.com/is/dig-lib/`.

[86] International Business Machines. IBM Deskstar 25GP and Deskstar 22GXP hard disk drives. Product Data Sheet, Apr. 1999. TECHFAX 7103.

[87] International Business Machines. IBM Ultrastar 36LZX hard disk drives. Product Data Sheet, Mar. 1999. TECHFAX 7111.

[88] Kahn, R., and Wilensky, R. A framework for distributed digital object services. `urn:hdl:cnri.dlib/tn95-01`, 13 May 1995.

[89] Karger, D. R., Lehman, E., Leighton, F. T., Levine, M. S., Lewin, D., and Panigrahy, R. Consistent hashing and

random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (El Paso, Texas, USA, 4–6 May 1997), pp. 654–663.

[90] KARLSSON, J. S., LITWIN, W., AND RISCH, T. LH*lh: A scalable high performance data structure for switched multicomputers. IDA Technical Report 1995 LiTH-IDA-R-95-25, Department of Computer and Information Science, Linköping University, S-581 83 Linkping, Sweden, 1995. ISSN-0281-4250.

[91] KEETON, K., PATTERSON, D. A., AND HELLERSTEIN, J. M. A case for intelligent disks (IDISKs). *SIGMOD Record 27*, 3 (Aug. 1998), 42–52.

[92] KNOWLTON, K. C. A fast storage allocator. *Commun. ACM 8*, 10 (Oct. 1965), 623–625.

[93] KOBLER, B., AND BERBERT, J. NASA Earth Observing System Data Information System (EOSDIS). In *Proceedings of the Eleventh IEEE Symposium on Mass Storage Systems* (Monterey, California, USA, 7–10 Oct. 1991), pp. 18–19.

[94] KOCH, P. D. L. Disk file allocation based on the buddy system. *ACM Trans. Comput. Syst. 5*, 4 (Nov. 1987), 352–370.

[95] KOHL, U., LOTSPIECH, J. B., AND KAPLAN, M. A. Safeguarding digital library contents and users: Protecting documents rather than channels. *D-Lib Magazine* (Sept. 1997). `urn:hdl:cnri.dlib/september97-lotspiech`.

[96] KOTZ, D. Disk-directed I/O for MIMD multiprocessors. *ACM Trans. Comput. Syst. 15*, 1 (Feb. 1997), 41–74.

[97] KRIEGER, O., AND STUMM, M. HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Trans. Comput. Syst. 15*, 3 (Aug. 1997), 286–321.

[98] KRÖLL, B., AND WIDMAYER, P. Distributing a search tree among a growing number of processors. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA, 24–27 May 1994), pp. 265–276.

[99] KRONENBERG, N. P., LEVY, H. M., AND STRECKER, W. D. VAXcluster: a closely-coupled distributed system. *ACM Trans. Comput. Syst. 4*, 2 (May 1986), 130–146.

[100] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS* (Cambridge, Massachusetts, USA, Nov. 2000).

[101] LAGOZE, C. A secure repository design for digital libraries. *D-Lib Magazine* (Dec. 1995). `urn:hdl:cnri.dlib/december95-lagoze`.

[102] LAGOZE, C., LYNCH, C. A., AND DANIEL, JR., R. The Warwick framework: A container architecture for aggregating sets of metadata. Computer Science Technical Report TR96-1593, Cornell University, June 1996.

[103] LAGOZE, C., AND PAYETTE, S. An infrastructure for open-architecture digital libraries. Tech. Rep. TR98-1690, Department of Computer Science, Cornell University, 1998. `urn:hdl:ncstrl.cornell/TR98-1690`.

[104] LAGOZE, C., SHAW, E., DAVIS, J. R., AND KRAFFT, D. B. Dienst: Implementation reference manual. Tech. rep., Cornell University, May 1995. `urn:hdl:ncstrl.cornell/TR95-1514`.

[105] LARSON, P. Dynamic hashing. *BIT* (1978), 184–201.

[106] LARSON, P. Dynamic hash tables. *Communications of the ACM 31*, 4 (Apr. 1988), 446–457.

[107] LARSON, P. Linear hashing with separators—a dynamic hashing scheme acheiving one-access retrieval. *ACM Trans. Database Syst. 13*, 3 (Sept. 1988), 366–388.

[108] LEE, E. K., AND KATZ, R. H. Performance consequences of parity placement in disk arrays. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS) IV* (1991), pp. 190–199.

[109] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS) VII* (Oct. 1996), pp. 84–92.

[110] LEINER, B. M. Metrics and the digital library. `urn:hdl:cnri.dlib/july98-editorial`, July 1998. Guest editorial.

[111] LEINER, B. M. The NCSTRL approach to open architecture for the confederated digital library. *D-Lib Magazine* (Dec. 1998). `urn:hdl:cnri.dlib/december98-leiner`.

[112] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the Harp file system. In *Proceedings of the 1991 Symposium on Operating Systems Principles (SOSP)* (Oct. 1991), pp. 226–238.

[113] LITWIN, W. Linear hashing: A new tool for file and table addressing. In *Proceedings of Very Large Databases* (Montreal, Canada, 1980).

[114] LITWIN, W., MENON, J., AND RISCH, T. LH* schemes with scalable availability. Research Report RJ 1021 (91937), IBM Almaden, May 1998.

[115] LITWIN, W., MENON, J., RISCH, T., AND SCHWARZ, T. J. E. Design issues for scalable availability LH* schemes with record grouping. In *DIMACS Workshop on Distributed Data and Structures* (Princeton University, May 1999), Carleton Scientific.

[116] LITWIN, W., AND NEIMAT, M.-A. High-availability LH* schemes with mirroring. In *Proceedings of the Conference on Cooperative Information Systems (COOPIS '96)* (Jan. 1996).

[117] LITWIN, W., NEIMAT, M.-A., LEVY, G., NDIAYE, S., AND SECK, T. LH*s: A high-availability and high-security scalable distributed data structure. In *Proceedings of IEEE RIDE '97* (1997).

[118] LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. LH*—linear hashing for distributed files. In *Proceedings of SIGMOD '93* (May 1993), ACM, pp. 327–336.

[119] LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. RP*: A family of order-preserving scalable distributed data structures. In *Proceedings of Very Large Databases* (Sept. 1994).

[120] LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. LH*—a scalable, distributed data structure. *ACM Transactions on Database Systems 21*, 4 (Dec. 1996), 480–525.

[121] LITWIN, W., AND SCHWARZ, T. J. E. LH*rs: A high-availability scalable distributed data structure using reed solomon codes. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA, 2000).

[122] LUSTRE DEVELOPERS. Lustre home. WWW page, 2000. `http://www.lustre.org`.

[123] Maly, K., Nelson, M. L., and Zubair, M. Smart Objects, Dumb Archives: A user-centric, layered digital library framework. *D-Lib Magazine 5*, 3 (Mar. 1999). `urn:doi:10.1045/march99-maly`.

[124] Matthews, J. N., Roselli, D., Costello, A. M., Wang, R. Y., and Anderson, T. E. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of Symposium on Operating Systems Principles (SOSP) 16* (Saint-Malo, France, Oct. 1997), pp. 238–251.

[125] McCoy, K. *VMS File System Internals*. Digital Press, 1990.

[126] McKusick, M. K. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3 (1984), 181–197.

[127] McKusick, M. K., and Kowalski, T. J. Fsck_ffs—the UNIX file system check program. In *BSD System Manager's Manual*. Computer Systems Research Group, University of California, Berkeley, July 1985, ch. 3.

[128] Mealling, M. URI resolution services necessary for URN resolution. RFC 2483, Jan. 1999. `http://www.ietf.org/rfc/rfc2483.txt`.

[129] Mealling, M., and Daniel, R. The Naming Authority Pointer (NAPTR) DNS resource record. RFC 2915, Sept. 2000. `http://www.ietf.org/rfc/rfc2915.txt`.

[130] Menon, J., Mattson, D., and Ng, S. Distributed sparing for improved performance of disk arrays. Tech. Rep. RJ 7943, IBM Almaden Research Center, 1991.

[131] Menon, J., Roche, J., and Kasson, J. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing 17* (1993), 129–139.

[132] Merchant, A., and Yu, P. Design and modeling of clustered RAID. In *Proceedings of the International Symposium on Fault Tolerant Computing* (1992).

[133] Miller, E. An introduction to the resource description framework. *D-Lib Magazine* (May 1998). `urn:hdl:cnri.dlib/may98-miller`.

[134] Moats, R. URN syntax. RFC 2141, May 1997. `http://www.ietf.org/rfc/rfc2141.txt`.

[135] Moore, R., Baru, C., Rajasekar, A., Ludaescher, B., Marciano, R., Wan, M., Schroeder, W., and Gupta, A. Collection-based persistent digital archives - part 1. *D-Lib Magazine 6*, 3 (Mar. 2000). `urn:doi:10.1045/march2000-moore-pt1`.

[136] Moore, R., Baru, C., Rajasekar, A., Ludaescher, B., Marciano, R., Wan, M., Schroeder, W., and Gupta, A. Collection-based persistent digital archives - part 2. *D-Lib Magazine 6*, 4 (Apr. 2000). `urn:doi:10.1045/april2000-moore-pt2`.

[137] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., and Lyon, B. Breaking through the NFS performance barrier. In *Proceedings of the EUUG Spring 1990* (Apr. 1990).

[138] Nelson, M. L., Maly, K., Shen, S. N. T., and Zubair, M. NCSTRL+: Adding multi-discipline and multi-genre support to the Dienst protocol using clusters and buckets. In *Proceedings of the IEEE Forum on Reasearch and Technology Advances in Digital Libraries, IEEE ADL '98* (Santa Barbara, California, USA, 22–24 Apr. 1998), IEEE Computer Society, pp. 128–136. ISBN 0-8186-8464-X.

[139] Ng, S. W. Crosshatch disk array for improved reliability and performance. In

Proceedings of the 21st International Symposium on Computer Architecture (1994), pp. 255–264.

[140] Nieuwejaar, N., and Kotz, D. Performance of the Galley parallel file system. In *Proceedings of IOPADS '96: Input/Output for Parallel and Distributed Systems* (Philadelphia, Pennsylvania, USA, 1996), pp. 83–94.

[141] Open Software Foundation. File systems in a distributed computing environment: A white paper. Tech. rep., Open Software Foundation, Cambridge, MA, 1991.

[142] Oracle Corporation. Oracle Internet File System (*i*FS). Product Overview, 2000. `http://www.oracle.com/database/options/ifs/iFSFO.html`.

[143] Ousterhout, J. K., Cherenson, A., Douglis, F., Nelson, M., and Welch, B. The Sprite network operating system. *IEEE Computer* (Feb. 1988), 23–36.

[144] Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. A trace-driven analysis of the UNIX 4.2 BSD file system. *Operating System Review 19*, 4 (1985), 15–24.

[145] Ousterhout, J. K., and Douglis, F. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review 23*, 1 (1989), 11–27.

[146] Paepcke, A., Baldonado, M. Q. W., Chang, C.-C. K., Cousins, S., and García-Molina, H. Using distributed objects to build the Stanford digital library Infobus. *IEEE Computer 32*, 2 (Feb. 1999), 80–87.

[147] Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepol, W., and Nahum, E. Locality-aware request distribution in cluster-based network servers. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS) VIII* (San Jose, California, USA, 2–7 Oct. 1998), pp. 205–216.

[148] Pai, V. S., Druschel, P., and Zwaenepoel, W. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)* (New Orleans, Louisiana, USA, 22–25 May 1999), pp. 15–28.

[149] Paskin, N. DOI: Current status and outlook. *D-Lib Magazine* (May 1999). `urn:doi:10.1045/may99-paskin`.

[150] Paskin, N. Toward unique identifiers. *Proceedings of the IEEE 87*, 7 (July 1999).

[151] Payette, S., and Lagoze, C. Flexible and Extensible Digital Object and Repository Architecture (FEDORA). In *Second European Conference on Research and Advanced Technology for Digital Libraries* (Heraklion, Crete, 21–23 Sept. 1998), vol. 1513 of *Lecture Notes in Computer Science*, Springer. `http://www2.cs.cornell.edu/payette/papers/ECDL98/FEDORA.html`.

[152] Pierce, P. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications* (1989), pp. 155–160.

[153] Plaxton, C. G., Rajaraman, R., and Richa, A. W. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures* (June 1997), pp. 311–320.

[154] Powell, A. Resolving DOI based URNs using Squid: An experimental system at UKOLN. *D-Lib Magazine* (June 1998). `urn:hdl:cnri.dlib/june98-powell`.

[155] RABIN, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM 36*, 2 (Apr. 1989), 335–348.

[156] RAMAKRISHNAN, K. K., BISWAS, P., AND KAREDLA, R. Analysis of file I/O traces in commercial computing environments. *Performance Evaluation Review 20*, 1 (June 1992), 78–90.

[157] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *SIGCOMM '01* (San Diego, California, USA, 27–31 Aug. 2001), Association for Computing Machinery, pp. 161–172.

[158] RIEDEL, E., AND GIBSON, G. A. Active disks - remote execution for network-attached storage. Tech. Rep. CMU-CS-97-198, Carnegie Mellon University, Dec. 1997.

[159] ROSENBLUM, M. *The Design and Implementation of a Log-Structured File System*. Kluwer, 1995.

[160] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *Proceedings of the thirteenth Symposium on Operating Systems Principles (SOSP)* (Pacific Grove, California, USA, 13–16 Oct. 1991), pp. 1–15.

[161] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (Mar. 1994), 17–29.

[162] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the Sun Network File system. In *Proceedings of the Summer USENIX Conference* (1985), pp. 119–130.

[163] SANTOS, J. R., AND MUNTZ, R. Performance analysis of the RIO multimedia storage system with heterogeneous disk configurations. In *Proceedings of the sixth ACM international conference on Multimedia* (Bristol, United Kingdom, 13–16 Sept. 1998), Association for Computing Machinery, pp. 303–308.

[164] SANTOS, J. R., MUNTZ, R., AND BENSON, S. A parallel disk storage system for real-time multimedia applications. In *Special Issue on Multimedia Computing Systems of the International Journal of Intelligent Systems* (1998).

[165] SATYANARAYANAN, M. A study of file sizes and functional lifetimes. In *Proceedings of Symposium on Operating Systems Principles (SOSP) 8* (Dec. 1981), Association for Computing Machinery, pp. 96–108.

[166] SATYANARAYANAN, M. Coda: A highly available file system for a distributed workstation environment. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems* (Sept. 1989).

[167] SATYANARAYANAN, M. Scalable, secure, and highly available distributed file access. *IEEE Computer* (May 1990), 9–20.

[168] SCHINDLER, J., AND GANGER, G. R. Automated disk drive characterization. In *Proceedings of the 2000 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Santa Clara, California, USA, June 2000), pp. 112–113.

[169] SCHULZE, M., GIBSON, G., KATZ, R., AND PATTERSON, D. How reliable is a RAID? In *Proceedings of the 34th IEEE Computer Society International Conference* (1989), pp. 118–123.

[170] SELTZER, M. *File System Performance and Transaction Support*. PhD thesis, University of California, Berkeley, 1992.

[171] SELTZER, M., BOSTIC, K., MCKUSIC, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *1993 Winter USENIX Conference* (San Diego, CA, 25–29 Jan. 1993).

[172] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison.

[173] SELTZER, M., AND STONEBRAKER, M. Read optimized file system designs: A performance evaluation. In *Proceedings of the 7th International Conference on Data Engineering* (Kobe, Japan, 8–12 Apr. 1991).

[174] SHAFER, K., WEIBEL, S., JUL, E., AND FAUSEY, J. Introduction to Persistent Uniform Resource Locators. WWW page, 1996. `http://purl.oclc.org/OCLC/PURL/INET96`.

[175] SHIERS, J. Building a database for the LHC: The exabyte challenge. In *Proceedings of the Fifteenth IEEE Symposium on Mass Storage Systems* (College Park, Maryland, USA, 23–26 Mar. 1998), pp. 385–395.

[176] SHRIVER, E. A formalization of the attribute mapping problem. Tech. Rep. HPL-SSP-95-10 Rev D, Hewlett-Packard Laboratories, Palo Alto, California, USA, 15 July 1996.

[177] SHRIVER, E. *Performance Modeling for Realistic Storage Devices*. PhD thesis, Department of Computer Science, New York University, New York, New York, USA, May 1997.

[178] SHRIVER, E., MERCHANT, A., AND WILKES, J. An analytical behaviour model for disk drives with readahead caches and request reordering. In *Proceedings of the 1998 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (June 1998), pp. 182–191.

[179] SIBERT, O., BERNSTEIN, D., AND VAN WIE, D. Securing the content, not the wire, for information commerce. Tech. rep., InterTrust Technologies Corp., 1996. `http://www.intertrust.com/architecture/stc.html`.

[180] SOLLINS, K. Architectural principles of uniform resource name resolution. RFC 2276, Jan. 1998. `http://www.ietf.org/rfc/rfc2276.txt`.

[181] SOLLINS, K., AND MASINTER, L. Functional requirements for uniform resource names. RFC 1737, Dec. 1994. `http://www.ietf.org/rfc/rfc1737.txt`.

[182] SOLTIS, S. R. *The Design and Implementation of a Distributed File System based on Shared Network Storage*. PhD thesis, University of Minnesota, Aug. 1997.

[183] STODOLSKY, D., AND GIBSON, G. A. Parity logging: overcoming the small write problem in redundant disk arrays. In *Proceedings of the 1993 Symposium on Operating Systems Principles (SOSP)* (1993).

[184] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01* (San Diego, California, USA, 27–31 Aug. 2001), Association for Computing Machinery.

[185] STORAGE NETWORKING INDUSTRY ASSOCIATION. A dictionary of storage networking terminology. World Wide Web page, 2000. `http://www.snia.org/English/Resources/Dictionary.html`.

[186] SUN, S. X., AND LANNOM, L. Handle system overview. IETF Draft, Aug. 2000. `http://www.ietf.org/internet-drafts/draft-sun-handle-system-05.txt`.

[187] SUN, S. X., REILLY, S., AND LANNOM, L. Handle system namespace and service definition. IETF Draft, Aug. 2000. `http://www.ietf.org/internet-drafts/draft-sun-handle-system-def-03.txt`.

[188] Szalay, A. S., Kunszt, P., Thakar, A., Gray, J., Slutz, D., and Brunner, R. J. Designing and mining multi-terabyte astronomy archives: The Sloan Digital Sky Survey. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Austin, Texas, USA, May 2000).

[189] Talagala, N., Asami, S., Anderson, T., and Patterson, D. Tertiary disk: Large scale distributed storage. Tech. rep., University of California at Berkeley, 1996.

[190] Talagala, N., Asami, S., Patterson, D. A., Hart, D., and Futernick, B. The art of massive storage: A case study of a web archive. *IEEE Computer* (Feb. 1999). submitted for publication.

[191] The Naming System Venture. ReiserFS. World Wide Web, Jan. 2001. `http://www.reiserfs.org/`.

[192] Thekkath, C. A., Mann, T., and Lee, E. K. Frangipani: A scalable distributed file system. In *Proceedings of Symposium on Operating Systems Principles (SOSP) 16* (Oct. 1997), pp. 224–237.

[193] Thiele, H. The Dublin Core and Warwick Framework. *D-Lib Magazine* (Jan. 1998). `urn:hdl:cnri.dlib/january98-thiele`.

[194] Upfal, E., and Widgerson, A. How to share memory in a distributed system. *J. ACM 34*, 1 (Jan. 1987), 116–127.

[195] URN Implementors. Uniform resource names: A progress report. *D-Lib Magazine* (Feb. 1996). `urn:hdl:cnri.dlib/february96-urn_implementors`.

[196] Uysal, M., Alvarez, G. A., and Merchant, A. A modular, analytical throughput model for modern disk arrays. In *Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2001)* (Cincinnati, Ohio, USA, 15–18 Aug. 2001).

[197] Vahdat, A., Anderson, T., and Dahlin, M. Active Names: Programmable location and transport of wide-area resources. Tech. rep., University of California at Berkeley, 1998.

[198] Vingralek, R., Breitbart, Y., and Weikum, G. Distributed file organization with scalable cost/performance. In *Proceedings of ACM-SIGMOD* (May 1994), pp. 253–264.

[199] Virtual Interface Architecture Consortium. Virtual Interface Architecture. World Wide Web page, 2000. `http://www.viarch.org/`.

[200] Vogels, W. File system usage in Windows NT 4.0. In *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP)* (Charleston, USA, 12–15 Dec. 1999), Association for Computing Machinery, pp. 93–109.

[201] Wang, R. Y., and Anderson, T. E. xFS: A wide area mass storage file system. Tech. rep., University of California at Berkeley, 1993.

[202] Wilkes, J. The Pantheon storage-system simulator. Tech. Rep. HPL-SSP-95-14, Hewlett-Packard Laboratories, Palo Alto, California, USA, May 1996.

[203] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. The HP AutoRAID hierarchical storage system. In *Proceedings of Symposium on Operating Systems Principles (SOSP) 15* (Dec. 1995), pp. 96–108.

[204] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. Dynamic storage allocation: A survey and critical review. In *Proceedings of the 1995 International Workshop on Memory Management* (Kinross, Scotland, United Kingdom, 27–29 Sept. 1995).

[205] WINTER, R., AND AUERBACH, K. Giants walk the earth: The 1997 VLDB survey. *Database Programming and Design 10*, 9 (Sept. 1997).

[206] WINTER, R., AND AUERBACH, K. The big time: The 1998 VLDB survey. *Database Programming and Design 11*, 8 (Aug. 1998).

[207] WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. On-line extraction of SCSI disk drive parameters. In *Proceedings of the 1995 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Ottawa, Canada, May 1995), pp. 146–156.