Ursa Minor: versatile cluster-based storage

Gregory R. Ganger, Michael Abd-El-Malek, Chuck Cranor,
James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad,
Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen,
John D. Strunk, Eno Thereska, Jay J. Wylie

CMU-PDL-05-104
April 2005

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

No single data encoding scheme or fault model is right for all data. A versatile storage system allows these to be data-specific, so that they can be matched to access patterns, reliability requirements, and cost goals. Ursa Minor is a cluster-based storage system that allows data-specific selection of and on-line changes to encoding schemes and fault models. Thus, different data types can share a scalable storage infrastructure and still enjoy customized choices, rather than suffering from "one size fits all." Experiments with Ursa Minor show performance penalties as high as 2–3× for workloads using poorly-matched choices. Experiments also show that a single cluster supporting multiple workloads is much more efficient when the choices are specialized rather than forced to use a "one size fits all" configuration.

Acknowledgements: We thank the members and companies of the PDL Consortium (including EMC, Engenio, Equallogic, Hewlett-Packard, HGST, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We also thank Intel, IBM, and Seagate for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by the Army Research Office, under agreement number DAAD19–02–1–0389. James Hendricks is supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense. This material is based on work supported by a National Science Foundation Graduate Research Fellowship.

Keywords: versatile storage, distributed storage, distributed file system, cluster-based storage

1 Introduction

Today's enterprise storage is provided by large monolithic disk array systems, extensively engineered to provide high reliability and performance in a single system. This approach, however, comes with significant expense and introduces scalability problems, since a given storage enclosure has an upper bound on how many disks it can support. To reduce costs and provide scalability, many are now pursuing cluster-based storage solutions (e.g., [12, 13, 14, 15, 16, 22, 28]). Cluster-based storage replaces the single system with a collection of smaller, lower-performance, less-reliable storage-nodes (sometimes referred to as *storage bricks*). Data and work are redundantly distributed among the bricks to achieve higher performance and reliability. The argument for the cluster-based approach to storage follows from both the original RAID argument [33] and arguments for cluster computing over monolithic supercomputers.

Cluster-based storage has scalability and cost advantages, but most designs lack the versatility expected from high-end storage solutions. By *versatility*, we mean that first-order data distribution choices (e.g., data encoding, fault tolerance, and data location) can be specialized to individual data (and their associated goals and workloads) stored within a shared infrastructure. Such versatility is crucial for addressing the varied demands of different classes of data. Failing to provide versatility forces all data into a single point of the performance–reliability–cost trade-off space. Versatility also addresses the impact of access patterns on the performance of different data distributions. For example, large sequentially-accessed data could be erasure-coded to reduce the space and bandwidth costs of fault tolerance, while randomly-accessed data might be replicated to avoid the read-modify-write penalties associated with erasure coding schemes.

This paper describes Ursa Minor, a cluster-based storage system designed to allow per-"data object" selection of, and on-line changes to, erasure coding scheme, block size, data location, and fault model. Ursa Minor achieves its versatility by using a *protocol family*, storing variably-sized data-fragments at individual storage-nodes, and maintaining per-object data distribution descriptions. A protocol family shifts the decision of which types of faults to mask from implementation time to data creation time, allowing each object to be protected from different types and numbers of faults. Ursa Minor's protocol family supports per-object choice of *data distribution* consisting of data encoding (replication or erasure coding), block size, storage-node fault type (crash, crash-recovery, Byzantine), number of storage-node faults to tolerate, timing model (synchronous or asynchronous), and data location. Storage-nodes treat all objects similarly, regardless of the object's data distribution.

Experiments with our implementation of Ursa Minor validate both the importance of versatility and Ursa Minor's ability to provide it. As illustrated in Figure 1, significant performance benefits are realized when data distribution choice is specialized to data access patterns and fault tolerance requirements. These benefits remain when multiple workload types share a storage cluster. Capacity benefits are also realized when erasure coding is used, instead of replication, to provide fault tolerance. For example, the data distribution for the Trace workload uses erasure coding to reduce space consumption by 50% while tolerating two crash failures; only a 7% performance penalty is paid for doing this, because this workload is highly sequential. Similarly, specializing the fault model ensures that costs for fault tolerance are incurred in accordance with acceptable risks, increasing throughput for data with lesser reliability requirements (e.g., the Scientific workload) by a factor of two over a reasonable "one size fits all" configuration.

Ursa Minor's ability to support on-line data distribution changes is also demonstrated. The ability to reconfigure data distributions on-line enables tuning based on observed usage rather than expected usage. This simplifies tuning, whether by administrators or automated tuning engines, since pre-deployment expertise about an application's access patterns becomes unnecessary. Reducing the amount of expertise and pre-deployment planning in such ways is important to reducing the excessive administration effort required with today's storage infrastructures.

This paper makes the following contributions. First, it makes a case for versatile cluster-based storage, arguing that versatility is needed to avoid significant performance, reliability, and/or cost penalties when

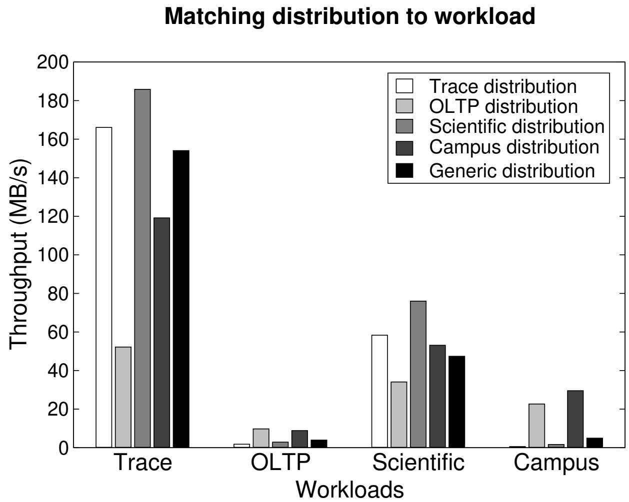

Figure 1: **Matching data distribution to workload.** This graph shows the performance of four workloads run on Ursa Minor as a function of the data distribution. For each workload, five distributions are evaluated: the best distribution for each of the four workloads and a generic "middle of the road" choice for the collection of workloads. Section 4.3 details the workloads and data distributions. "Trace" corresponds to read-only streaming of large trace files. "Scientific" corresponds to large writes and reads of temporary data. "OLTP" and "Campus" correspond to transaction processing and campus file system workloads respectively. The OLTP and Trace data are viewed as important, requiring tolerance of two storage-node crash failures. Campus must tolerate one, and Scientific none.

storage is shared among different classes of data. Second, it describes the design and implementation of a cluster-based storage system, Ursa Minor, that is versatile; we are aware of no existing cluster storage system that provides nearly as much versatility, including the ability to specialize fault models and to change data distributions on-line. Third, it presents measurement results of the Ursa Minor prototype that demonstrate the value of versatility for specializing to access patterns, specializing to reliability requirements, and allowing on-line changes of data distributions.

The remainder of this paper is organized as follows. Section 2 motivates versatile cluster-based storage and discusses related work. Section 3 describes Ursa Minor, our versatile cluster-based storage system. Section 4 evaluates Ursa Minor, focusing on how its versatility enables data-specific specialization and the resulting benefits. Section 5 summarizes this paper's contributions and discusses future work.

2 Versatile cluster-based storage

This section makes a case for versatile cluster-based storage. It explains the potential of cluster-based storage, describes desired forms of versatility, and discusses related work.

2.1 Cluster-based storage

Today's enterprise storage systems are monolithic and very expensive, based on special-purpose, hot-swappable components with comprehensive internal redundancy. These systems are engineered and tested to tolerate harsh physical conditions and continue operating under almost any circumstance. They provide high-performance and high-reliability, but do so at great monetary expense.

Cluster-based storage is a promising alternative to today's monolithic storage systems [5, 12, 13, 14, 15, 16, 22, 28]. The concept is that, appropriately coordinated, collections of smaller storage-nodes should be able to provide performance and reliability competitive with today's solutions, but at much lower cost. The cost reductions would come from using commodity components for each storage-node and exploiting economies of scale. Each would provide a small amount of the performance needed and lower reliability than required. As with previous arguments for RAID and cluster computing, the case for cluster-based storage anticipates that high levels of reliability and performance can be obtained by appropriate inter-storage-node redundancy and workload distribution. If successful, cluster-based storage could be much less expensive (per terabyte) than today's enterprise storage systems [14].

Cluster-based storage also helps with the scaling challenges inherent to monolithic storage systems. In particular, once the limit on the number of disks that can be inserted into a large storage system's enclosures is reached, a second large system must be purchased and data must be redistributed across the enclosures. Avoiding this drastic step-function in effort and capital expense can push administrators to purchase oversized (but mostly empty) systems. Most cluster-based storage designs allow growth of capacity and performance through the incremental addition of storage-nodes, with automated balancing of the data to utilize the new resources.

2.2 Versatility in cluster-based storage

Cluster-based storage has cost and scalability advantages over traditional monolithic storage systems. To replace them effectively, however, cluster-based storage must provide similar versatility. It must be possible to specialize the data distribution, protection, and access schemes for different classes of data. Data distribution choices involve significant trade-offs among performance, reliability, and cost. No single choice of data distribution is right for all data.

This section describes several choices that should be specialized to individual data based on application requirements (e.g., risk tolerance and performance goals), access patterns, and cost restrictions. Almost all modern disk array systems allow the redundancy scheme (e.g., RAID 5 vs. RAID 0+1) and stripe unit size to be chosen on a per-volume basis. Cluster-based systems should have similar versatility. In addition, cluster-based storage introduces questions of fault model choice that have greater impact than in the centralized controller architecture of monolithic storage systems.

Data distribution choices: Data can be spread across cluster storage-nodes in multiple ways to address two primary concerns: fault tolerance and load balancing. In most cluster-based storage designs, assignment of data to storage-nodes is internally adapted to automatically balance load. The approach to fault tolerance, on the other hand, is often fixed for all data.

There are two common redundancy schemes for cluster-based storage. First, data can be *replicated* such that each block is stored on two or more storage-nodes. Second, data can be *erasure coded*; for example, an m-of-n erasure code encodes a data block into n fragments such that any m can be used to reconstruct it. (RAID 5 is an m-of-$(m+1)$ scheme.) The trade-off between these schemes is similar to that of RAID 5 versus RAID 0+1 in disk array systems. Replicated data generally supports higher out-of-cache throughput for non-sequential accesses, because fewer disk accesses are involved in servicing each request. Erasure coded data, on the other hand, can tolerate failures (especially multiple failures) with less network bandwidth and storage space [44, 46]. For sequentially accessed data, these benefits can be realized without significant disk access penalties.

Most modern disk array systems use data distributions that can tolerate a single disk failure. This is unlikely to be sufficient in cluster-based storage systems that use less robust disks designed for desktop systems. Further, other components (e.g., fans and power supplies) that can fail and be hot-swapped in high-end storage systems will translate into storage-node failures in cluster-based storage systems. The expectation, therefore, is more frequent storage-node failures. Even with traditional systems, manufacturers have recognized that it is becoming important to tolerate multiple disk failures [8]. In cluster-based storage, tolerating two or more storage-node failures is likely to be required for important data. Because of the performance and capacity costs involved with tolerating more failures, however, the number of failures tolerated will need to be configurable.

Fault model choices: Most cluster-based storage designs are decentralized systems, as contrasted with monolithic disk systems with their centralized controllers. Centralized controllers provide a serialization point, a single restart location, and unambiguous storage-node (i.e., disk) failure indication. Decentralized cluster-based storage enjoys none of these features. As a result, carefully designed data access protocols are utilized to provide data consistency in the face of storage-node failures, communication delays, client failures, and concurrent access.

The overheads associated with these protocols depend significantly on their underlying fault model assumptions, for which there are many choices. For example, one might assume that faulty storage-nodes only ever crash or that they might behave more arbitrarily (e.g., corrupting data or even exhibiting Byzantine behavior). One might assume that clocks are synchronized and communication delays are bounded (i.e., a *synchronous* timing model) or that storage-node reboots and transient network delays/partitions make timing assumptions unsafe (i.e., an *asynchronous* timing model). Weakening failure and timing assumptions generally makes a system more robust at the expense of the amount of data redundancy and communication required. Section 3.2 and our experimental results clarify these trade-offs.

It is tempting to assume that tolerating storage-node crashes is sufficient, and that good engineering can prevent Byzantine failures and timing faults. However, given the amount of software involved and the consumer-quality components that are likely to be integrated into cluster-based storage systems, there is real risk associated with that assumption. Even in today's high-end storage systems, there are mechanisms designed to mask non-crash communication and firmware failures within the controller and the disks. For example, we have been told (e.g., [24]) that disks occasionally write data sectors to the wrong location.[1] Such a fault corrupts two pieces of data: the old version of the updated data goes unmodified (an "omission failure") and some unassociated data is replaced. Non-crash failures can be expected to increase in frequency when using less robust components to construct a system.

Ability to change choices on-line: Most cluster-based storage designs adaptively modify the assignments of data replicas/fragments to storage-nodes based on access patterns and storage-node availability. We believe that it is desirable for other data distribution choices to be adaptable as well. If modifying such choices were easy, administrators could worry less about getting the initial configuration choice perfect, especially with regards to tuning to match access patterns. Instead, applications and their storage could be deployed, and then the data distribution choices adjusted based on the observed access pattern. Even the number and type of faults tolerated could be changed based on the problems observed in practice.[2] By allowing changes based on observed behavior, a system can save storage administrators from having to gain expertise in the impacts of each physical environment and the storage behavior of each major application before deploying it—trial-and-error approaches could be used to arrive at an acceptable system configuration.

2.3 Related work

There is a large body of previous work in cluster-based storage and in adaptive storage systems. This section overviews some high-level relationships to Ursa Minor's goal of versatile cluster-based storage. Related

[1] Exact reasons for this sort of problem are rarely reported, but the observed behavior is not limited to a single disk make or model. It could be caused by bugs in firmware or by hardware glitches induced by vibration, heat, or other physical effects.

[2] The physical challenges of data centers, such as heat dissipation and vibration, make storage fault tolerance less uniform across instances of a system. A deployment in an environment that struggles more with these issues will encounter more failures and perhaps more non-crash failures than one in a more hospitable environment.

work for specific mechanisms are discussed with those mechanisms.

Many scalable cluster-based storage systems have been developed over the years. Petal [28], xFS [5], and NASD [17] are early systems that laid the groundwork for today's cluster-based storage designs, including Ursa Minor's. More recent examples include Farsite [2], FAB [38], EMC's Centera [12], Equallogic's PS series product [13], and the Google file system [16]. All of these distributed storage solutions provide the incremental scalability benefits of cluster-based storage, as well as some provisions for fault tolerance and load balancing. Each, however, hard codes most data distribution choices for all storage in the system. For example, Petal replicates data for fault tolerance, tolerates only server crashes (among types of failures), and uses chained declustering to spread data and load across nodes in the cluster; these choices apply to all data. xFS uses parity-based fault tolerance for server crashes and data striping for load spreading, with one choice for the entire system. Ursa Minor's design builds on previous cluster-based storage systems to additionally provide versatility; its single design and implementation supports a wide variety of data encoding schemes, fault models, and data distribution choices, all selectable and changeable on-line on a per-object basis.

FAB [38] and RepStore [48] offer two encoding scheme choices (replication and erasure coding) for data rather than just one. FAB allows the choice to be made on a per-volume basis at volume creation time. RepStore, which has been designed and simulated [48], uses AutoRAID-like algorithms to adaptively select which to use for which data. Reported experiments with the FAB implementation and the RepStore simulator confirm our experiences regarding the value of this one form of versatility. Ursa Minor goes beyond both in supporting a much broader range of configuration choices for stored data, including fault models with non-crash failures. Compared to FAB, Ursa Minor also supports on-line re-encoding of data, allowing its configuration choices to be modified on-line.

Pond [36] uses both replication and erasure coding for data in an effort to provide Internet-scale storage with long-term durability. It uses replication for active access and erasure coding for long-term archiving. Although it does provide incremental scalability, it is designed for wide-area deployment rather than single-data-center cluster-based storage. Partly as a consequence, it does not provide most of the versatility options of Ursa Minor.

An alternative approach to cluster-based storage, as we have presented it, is to provide scalability by interposing a proxy [39], such as Slice [3], Cuckoo [25], or Anypoint [47]. Such proxies can spread data and requests across servers like a disk array controller does with its disks. Most such designs rely on the back-end storage servers to provide redundancy, though redundancy could be orchestrated by the proxy. Versatility could be provided by the proxy or by selectively choosing back-ends and their configurations. This approach to storage infrastructure maintenance represents a middle-ground between traditional storage and cluster-based storage. The best choice for any particular circumstance remains an open question; our work serves to extend the reach of cluster-based storage by enabling versatility.

AutoRAID [45] provides versatile storage in a monolithic disk array controller. Most disk array controllers allow specialized choices to be made for each volume. AutoRAID goes beyond this by internally and automatically adapting the choice for a data block (between RAID 5 and mirroring) based on usage patterns. By doing so, it can achieve much of the best of both encodings: the cost-effectiveness of RAID 5 storage for infrequently used data and the performance of mirroring for popular data. Ursa Minor brings versatility and the ability to select and change data distributions on-line to distributed cluster-based storage. To achieve AutoRAID's automatic adaptivity, Ursa Minor's versatility should be coupled with similar workload monitoring and decision-making logic.

3 Ursa Minor

Ursa Minor is a highly versatile cluster-based storage system. This section describes its design and implementation, focusing on aspects that enable its versatility.

3.1 Architecture and goals

Ursa Minor provides storage of *objects* in the style of NASD [17] and the emerging OSD standard [31]. In general, an object consists of basic attributes (e.g., size and ACLs) and byte-addressable data. Each object has a numerical identifier (an *object ID*) in a flat name space. The system provides file-like access operations, including object CREATE and DELETE, READ and WRITE, GET_ATTRIBUTES and SET_ATTRIBUTES, and so on. The primary difference from file systems is that there are no ASCII names or directories.

The primary advantage of object-based storage is that it explicitly exposes more information about data stored in the system than a purely block-based storage interface like SCSI or ATA, while avoiding the specific naming and other semantics of any individual file system. Specifically, it exposes the set and order of data that make up each object, as well as some attributes. This information simplifies the implementation of secure direct access by clients to storage-nodes—this was the primary argument for the NASD architecture [17]. It is also valuable for accounting and other administrative activities. For Ursa Minor, it also facilitates the manipulation of data distribution choices for individual objects.

Like NASD and other object-based storage systems, Ursa Minor allows direct client access to storage-nodes as illustrated in Figure 2. Clients first consult with the object manager to discover the data distribution (including the list of storage-nodes) used for their data. Afterward, they can interact directly with the storage-nodes to access it. The object manager provides metadata and authorization that enable clients to read and write data at storage-nodes. Metadata operations, such as object creation and deletion, are done through the metadata service. Metadata-affecting data operations, such as object extension, require an extra interaction with the object manager (to record the new length, for object extension).

Much of Ursa Minor's versatility is enabled by the protocol family it uses for data access. A *protocol family* supports different fault models in the same way that most access protocols support varied numbers of failures: by changing the number of storage-nodes accessed for reads and writes. Ursa Minor's protocol family operates on arbitrarily-sized blocks of data and is overviewed in Section 3.2, together with the client-server interface and the storage-node implementation. The protocol family allows each piece of data to use any of many data encoding schemes and conform to any of many fault and timing models.

Each object's data is stored as one or more ranges of bytes, called *slices*. Each slice is stored as a sequence of blocks with a common block size, encoding scheme, data-fragment locations, and protocol family configuration. Different slices can have different values for any of these choices. Slices allow large objects to be partitioned across multiple sets of storage-nodes (while also being striped within each set). Slices also allow for incremental re-encode. Although slices are integral to the Ursa Minor design, to simplify discussion, most of the paper refers to the data distribution of objects rather than that of a slice of an object.

Section 3.3 describes three additional Ursa Minor components: the client library that translates between a byte-based view of objects and the protocol's block-based operation, the object manager that hold per-object attributes and slice information, and the NFS server used to enable access for unmodified clients.

On-line change of an object's data distribution is arbitrated by the object manager. The data distribution can be changed for granularities as small as a block. As well, clients are not prevented from accessing object data during on-line changes of the object's data distribution. Such a data distribution change can alter the storage locations, encoding, fault model, or block size that define the backing-store for bytes of an object. Section 3.4 describes this process in detail.

3.2 Protocol family for versatile access

Data access in Ursa Minor builds on a protocol family [19] that supports consistent read/write access to data blocks. Each protocol family member conforms to one of two timing models, one of several fault models, and supports any threshold erasure coding scheme for data. Member implementations are distinguished by

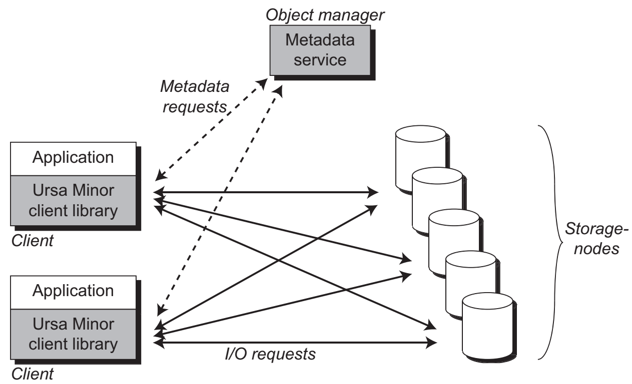

Figure 2: **Ursa Minor high-level architecture.** Clients use the storage system via the Ursa Minor client library. The metadata needed to access objects is retrieved from the object manager. Requests for data are then sent directly to storage-nodes.

choices enacted in client-side software regarding the number of storage-nodes accessed and the logic employed during a read operation. The storage-node implementation and client-server interface is the same for all members. This section overviews the protocol family, its guarantees, its operation and implementation, and the storage-node interface and implementation.

3.2.1 Protocol family overview

This section overviews the operation and properties of the read/write protocol family used in Ursa Minor. Pseudo-code and proofs of correctness are available in a separate technical report [19].

The fault-tolerance of each member of the protocol family is determined by three independent parameters: the timing model, the storage-node failure model, and the client failure model.

Timing model: Protocol family members are either asynchronous or synchronous. Asynchronous members rely on no timeliness assumptions (i.e., no assumptions about message transmission delays or execution rates). In contrast, synchronous members assume known bounds on message transmission delays between correct clients and storage-nodes as well as their execution rates. By assuming a synchronous system, storage-nodes that crash are detectable via timeouts, providing useful information to the client. In an asynchronous system, on the other hand, storage-node crashes are indistinguishable from slow communication.

Storage-node failure model: Each family member supports a hybrid storage-node failure model [42]. Up to t storage-nodes may fail, $b \leq t$ of which may be Byzantine faults [27]; the remainder can only crash. Such a model can be configured to a wholly crash (i.e., $b = 0$) or wholly Byzantine (i.e., $b = t$) model for storage-node failures.

Client failure model: Every member of the protocol family tolerates crash client failures and some also tolerate Byzantine client failures. Client crash failures during write operations can result in subsequent read operations (by other clients) observing an incomplete write operation. As in any general storage system, an authorized Byzantine client can write arbitrary values to storage. This affects the value of the data, not its consistency. Mechanisms for detecting and protecting against Byzantine client failures during write operations (e.g., "poisonous writes" [29]) are described in [19]. Although the implementation supports them, we do not employ the Byzantine client mechanisms in Ursa Minor—we consider the value of preventing this consistency attack (as opposed to actual data corruption) to be of minor practical value.

3.2.2 Protocol guarantees and constraints

All members of the protocol family guarantee linearizability [21] and wait-freedom [20, 23] of all correct operations. Of course, to do so, the number of storage-nodes (and thus fragments, n) must conform to constraints with regard to b and t (from the hybrid model of storage-node failure). For asynchronous members, the constraint is $2t+2b+1 \leq n$. For synchronous members, the constraint is $t+b+1 \leq n$. Full development and proof sketches for these and other relevant constraints (e.g., read classification rules and m) are available in [19].

3.2.3 Protocol operation and implementation

Each protocol family member supports read and write operations on arbitrarily-sized blocks. The Ursa Minor client library contains a protocol library that accepts, as input for each operation, an object ID, data block, block number, and data distribution (i.e., the fault model, data encoding scheme, block size, and list of storage-nodes to use). This library handles the protocol execution on behalf of the client.

To write a block, the client encodes it into n fragments; any threshold-based (i.e., m-of-n) erasure code (e.g., information dispersal [35], short secret sharing [26], or replication) could be used. *Logical timestamps* are used to totally order all write operations and to identify fragments from the same write operation across storage-nodes. For each correct write, a client constructs a logical timestamp that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp). Forming this timestamp either requires GET_LOGICAL_TIME requests to storage-nodes (for asynchronous members) or reads of the local clock (for synchronous members). A write operation is complete when enough storage-nodes have executed write requests to guarantee that no subsequent read operation can return a previously written value. Storage-nodes provide fine-grained versioning; a correct storage-node stores a fragment version (indexed by logical timestamp) for each write request it executes.

To read a block, a client issues read requests to a subset of the listed storage-nodes. From the responses, the client identifies the *candidate*, which is the fragment version returned with the greatest logical timestamp. The read operation classifies the candidate as *complete*, *incomplete* or *repairable* based on the number of read responses that share the candidate's timestamp. If the candidate is classified as complete, then the read operation is done, and the value of the candidate is returned. This is by far the most common case. Only in certain cases of failures or concurrency are incomplete or repairable candidates observed. If it is classified as incomplete, the candidate is discarded, another read phase is performed to collect previous versions of fragments, and classification begins anew. This sequence may be repeated. If the candidate is repairable, it is repaired by writing it back to storage-nodes that do not have it (with the logical timestamp shared by the repairable fragments). Then, it is returned.

Byzantine storage-nodes can corrupt their data-fragments. As such, it must be possible to detect and mask up to b storage-node integrity faults. Cross checksums [18] are used to detect corrupt data-fragments: a cryptographic hash of each data-fragment is computed, and the set of N hashes are concatenated to form the *cross checksum* of the data-item. The cross checksum is stored with each data-fragment, as part of the timestamp, enabling corrupted data-fragments to be detected by clients during reads. (Although not enabled, Byzantine clients are tolerated by having clients regenerate the cross checksum during read operations, to verify that the original write was well-formed.) Our implementation uses a publicly available implementation of MD5 [1] for all hashes.

The protocol implementation includes a number of performance enhancements that exploit its threshold nature. For example, to improve the responsiveness of write operations, clients return as soon as the minimum number of required success responses are received; the remainder of the requests complete in the background. To improve the read operation's performance, only m read requests fetch the actual contents stored for the fragment, while all fetch version histories (some of which act as witnesses). This makes

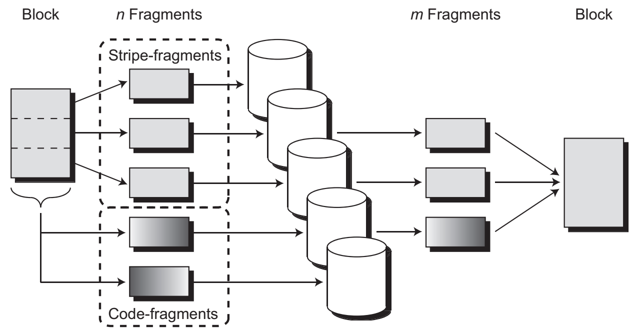

Figure 3: **Erasure coding example.** This example shows a 3-of-5 erasure encoding WRITE and then READ of a block. On WRITE, the original block is broken into three stripe-fragments and two code-fragments and stored on five separate storage-nodes. On READ, only three fragments (stripe or code) need to be retrieved from the storage-nodes to reconstruct the original block.

the read operation more network-efficient. If necessary, after classification, extra fragments are fetched according to the candidate's timestamp.

Our implementation supports both replication and an m-of-n erasure coding scheme. If $m = 1$, then replication is employed. Otherwise, our base erasure code implementation stripes the block across the first m fragments; each *stripe-fragment* is $\frac{1}{m}$ the length of the original block. Thus, concatenation of the first m fragments produce the original block. (Because "decoding" with the m stripe-fragments is computationally less expensive, the implementation always tries to read from the first m storage-nodes for any block.) The stripe-fragments are used to generate the *code-fragments* that provide the necessary redundancy via polynomial interpolation within a Galois Field. Our implementation of polynomial interpolation was originally based on [10] (which conceptually follows [35]). We modified the source to use stripe-fragments and added an implementation of Galois Fields of size 2^8 that use a lookup table for multiplication. Figure 3 illustrates how stripe- and code-fragments are stored.

3.2.4 Storage-node interface and implementation

Storage-nodes expose the same interface, regardless of the protocol family member being employed—write and read requests for all protocol family members are serviced identically. Clients communicate with storage-nodes via a TCP-based RPC interface. Storage-nodes provide interfaces to write a fragment at a specific logical time, to query the greatest logical time of a fragment, to read the fragment version with the greatest logical time, and to read the fragment with the greatest logical time before some logical time. By default, a read request returns the most current fragment version, ordered by logical timestamp. Reads may also request the fragment corresponding to a specific logical timestamp, or just part of the version history (sequence of logical timestamps) associated with a fragment.

Storage-node requests address data fragments by block number because the fragment size is not fixed. Fragment sizes vary for three reasons. First, the block size (for protocol read/write operations) is configurable and should be chosen based on data access patterns (e.g., to match the page size for database activity). Second, erasure coding results in $\frac{blocksize}{m}$ bytes per fragment. Third, the storage-node will sometimes be asked to hold information (the "back-pointers" described in Section 3.4) about in-progress distribution changes instead of data. On a write, the storage-node accepts whatever number of bytes the client sends and records it, indexed by the specified object ID and block number. On a read, the storage-node returns whatever content it holds for the specified object ID and block number.

Each write request implicitly creates a new version of the fragment (indexed by its logical timestamp) at the storage-node. A log-structured organization [37] is used to reduce the disk I/O cost of data versioning.

Multi-version b-trees [6, 40] are used by the storage-nodes to store fragments; all fragment versions are kept in a per-object b-tree indexed by a 2-tuple $\langle blocknumber, timestamp \rangle$. Like previous researchers [34, 41], our experiences indicate that retaining versions and performing local garbage collection come with minimal performance cost (a few percent) and that it is feasible to retain version histories for several days.

Garbage collection of old versions is necessary to prevent capacity exhaustion of the storage-nodes. A storage-node in isolation cannot determine which local fragment versions are safe to garbage-collect, because write completeness is a property of a set of storage-nodes. A fragment version can be garbage-collected only if there exists a later complete write for the corresponding block. Storage-nodes can classify writes by executing the read protocol in the same manner as a client, excluding the actual data fetches.

The storage-node implementation is based on the S4 object store described in [40, 41]. It uses a write-back cache for fragment versions, emulating non-volatile RAM. (Our storage-nodes are battery-backed, but our implementation does not yet retain the cache contents across reboots.) The storage-node additionally maintains a sizeable cache of latest timestamps (including the cross checksums) associated with fragments. The hit rate of the timestamp cache is crucial for performance, as it eliminates disk accesses for storage-nodes that are accessed just to ensure consistency (rather than to access one of *m* data-fragments). Storage-nodes also have two other caches, a data cache for log segments and a metadata cache for version indices.

3.3 Other Ursa Minor components

The protocol and storage-node implementations are discussed in Sections 3.2.3 and 3.2.4, respectively. This section overviews the function and implementation of Ursa Minor's other primary components: object manager, client library, and NFS server.

Object manager: The object manager maintains Ursa Minor metadata about each object, including data distribution and access control information. Clients send RPCs to the object manager to create and delete objects, access attributes, and get distributions and authorizations for accessing data.

To access data, a client sends the object ID and byte offset to the object manager and, if it has appropriate access rights, gets back a *slice descriptor* and a capability. A slice descriptor details the data distribution of the slice containing the specified byte offset, including the byte range, block size, block numbers, encoding scheme, fault model, and list of storage-nodes. The object manager maintains one or more slice descriptors for each object, as needed.

The object manager implementation uses Berkeley DB [32] b-trees, stored in objects, to organize and index the Ursa Minor metadata. To enable crash recovery, db was extended to support shadow paging. The object manager implementation does not currently provide real capabilities; the field is empty and all client requests are serviced by storage-nodes without actual authorization.

Client library: The client library provides a byte-addressed object interface to application code, hiding the details of Ursa Minor. It includes the protocol library discussed earlier, as well as code for interacting with the object manager and dealing with other Ursa Minor details (e.g., block-based storage-node access and in-progress distribution change). The client library is a convenience for programmers, and it is not trusted by storage-nodes or object managers any more than application code.

NFS server: Access to data stored in Ursa Minor clearly involves non-standard protocols. To support unmodified clients, we have implemented an NFS server that exports files and directories stored as objects in Ursa Minor. It bridges between UDP-based NFS version 3 and Ursa Minor's internal protocols. File and directory contents are stored as object data, and the NFS_ATTR structure for each is stored in the first block of the corresponding object. Directories map file names to Ursa Minor object IDs, which in turn are used as NFS file handles so as to allow direct processing of each request.

Such an NFS server is clearly not intended as the primary method of access to a cluster-based storage system like Ursa Minor. But, it is convenient for early and incremental deployment. Our implementation executes as a user-level process and is not tuned, but it suffices for its purpose.

3.4 On-line change of data distribution

In addition to create-time versatility, Ursa Minor supports on-line change of an object's data distribution. This permits an administrator or automated tuning tool to correct poorly chosen distributions and to change distributions as access patterns evolve.

To transition between data distributions, Ursa Minor makes use of *back-pointers*. A back-pointer is a copy of an old data distribution stored as the zeroth version of blocks in a new data distribution. By providing a link between the new distribution and the old, back-pointers obviate the need to halt client access to an object during the data re-encode step (which can occur in the background). A reader can follow the back-pointer to the last data written in the old distribution, if no data has yet been written to the new.

A distribution change proceeds in four steps. First, the object manager blocks client write requests to the affected blocks by revoking their authorizations. Second, the object manager installs back-pointers to the old distribution by writing them to the storage-nodes that store the new distribution. One back-pointer is written for each new block. Third, the object manager updates its metadata with the new distribution and revokes remaining (read-only) authorizations to the re-encoded data. Clients learn of the new distribution when they ask the object manager for access. Fourth, clients access data according to the new distribution while it is copied, in the background, from the old to the new distribution.

During step four, clients write directly to the new distribution. When a client reads data from the new distribution, it may encounter either a back-pointer or data. If it encounters a back-pointer, the client library will proceed to access the identified old distribution. Once it encounters data, it proceeds as normal. Note that the data read by a client in step four may have been copied from the old distribution or it may be newly written data originating since the distribution was changed.

The Ursa Minor component that transitions data from the old distribution to new (step four) is called a *distribution coordinator*. It copies data incrementally (e.g., a block at a time), in the background, taking care not to write over data already written by a client to the new distribution. To ensure this behavior, the coordinator must set the timestamp for data it writes to be after the timestamp of the back-pointer but before the timestamp of any new client writes. (Write capabilities after a distribution change reserve a timestamp value after the back-pointer to enable this function.)

One of the trickier aspects of data distribution change arises when the data block size is changed. Changes in data-fragment sizes are not a problem, because storage-nodes are designed to cope with such variation. Changes in the block size (used to break up the byte stream into blocks on which the protocol operates), however, will change the number of blocks needed for a given byte range. This can cause conflicts between block numbers wanted for different ranges of data bytes in an object. This problem is addressed by decoupling the block numbers used for storage from the byte offsets accessed by clients—a slice descriptor identifies the block numbers explicitly rather than having clients compute them. When a block size change is performed, a new range of block numbers within the object is used for the new distribution, eliminating any conflict.

The current implementation of on-line data distribution change departs from the design in a couple of ways, both stemming from the lack of real support for capabilities. First, the revocation steps are emulated with call-backs to clients rather than communication with storage-nodes. Second, there are no reserved timestamps, meaning that coordinator writes can overwrite subsequent client writes. Although not acceptable for deployment, these shortcomings should not affect performance experiments with the mechanisms.

Ursa Minor's approach to on-line distribution change minimizes blocking of client accesses and allows incremental application of change. Further, the notion of slices allows back-pointer insertion for change to a large object to be performed piecemeal rather than all at once, reducing the duration of time during which writes are blocked (for back-pointer application). In addition, the coordinator can move data to the new distribution at whatever rate is appropriate.

4 Evaluation

This section evaluates Ursa Minor and its versatility in four specific areas: (i) we demonstrate that the baseline performance of NFS over Ursa Minor reasonable; (ii) we demonstrate that the versatility provided by Ursa Minor provides significant benefits for different synthetic workloads; (iii) we demonstrate that the Ursa Minor prototype can perform on-line changes of an object's data distribution; and, (iv) we demonstrate, via trace-based analysis, that stored data exhibits distinct characteristics that, when matched to appropriate distributions, can provide significant storage capacity benefits.

4.1 Experimental setup

All experiments are run using Dell PowerEdge 650 machines equipped with a single 2.66 GHz Pentium 4 processor, 1 GB of RAM, and Seagate ST33607LW, 36 GB, 10K rpm SCSI disks. The network configuration consists of a single Intel 82546 gigabit Ethernet adapter in each machine, connected via a Dell PowerConnect 5224 switch. The machines run the Debian "testing" distribution and use Linux kernel version 2.4.22. The same machine types are used both as clients and storage-nodes in experiments.

In all experiments, the working set is larger than client and storage-node caches. Indeed, to ensure that the measurements are of disk-bound activities, storage-nodes are configured with smaller caches than the 1 GB of RAM on each storage-node allows. Specifically, storage-nodes use a 16 MB data cache and a 32 MB metadata cache.

4.2 Baseline prototype performance: Ursa Minor NFS

This section uses application benchmarks to show that Ursa Minor achieves reasonable performance. To do so, Ursa Minor NFS server's performance is compared to that of a Linux kernel-level NFSv3 server. The Linux NFS server is configured to communicate with clients using UDP. In both cases, the NFS servers run on dedicated machines. The Linux NFS server exports an ext3 partition that resides on one of its local disks and uses a variable amount of memory for its buffer cache. The Ursa Minor NFS server exports data stored on a single storage-node (i.e., the versatility of Ursa Minor NFS is not exploited) with the object manager running on the same machine as the NFS server. Ursa Minor NFS is configured to use 384 MB of data cache and 32 MB of attribute cache. Additionally, since Ursa Minor NFS assumes NVRAM on storage-nodes, the Linux NFS server is configured with the "async" flag, allowing it to reply to requests before the requested data reaches stable storage.

For the experiments, we run the TPC-C and Iozone benchmarks. The TPC-C benchmark [9] simulates an on-line transaction processing database workload, where each transaction consists of a few read-modify-write operations to a small number of records. The disk locations of these records exhibit little locality. We ran TPC-C on the Shore database storage manager [7]. We configured Shore and TPC-C to use 8 KB pages, 10 warehouses and 10 clients, giving it a 5 GB footprint. The Shore volume is a file stored on either the Linux NFS server or the Ursa Minor NFS server. Performance of a TPC-C benchmark is measured in TPC-C transactions completed per minute (TpmC).

Iozone is a general, user-level file system benchmarking tool [30]. It tests many different I/O types and access patterns. For these experiments, we show the performance for 64 KB sequential writes and reads to a single 2 GB file.

Figure 4 shows performance for small, random I/O (using TPC-C) and sequential I/O (using Iozone). Because Ursa Minor adds additional functionality in the back-end of its NFS server, including a network hop, we expect its performance to be lower than that of a local NFS server. In addition, because we are running a prototype system, we expect that the tuned Linux NFS server will have better performance at this stage. As expected, the Ursa Minor NFS prototype server's performance is between 56% and 82% that of

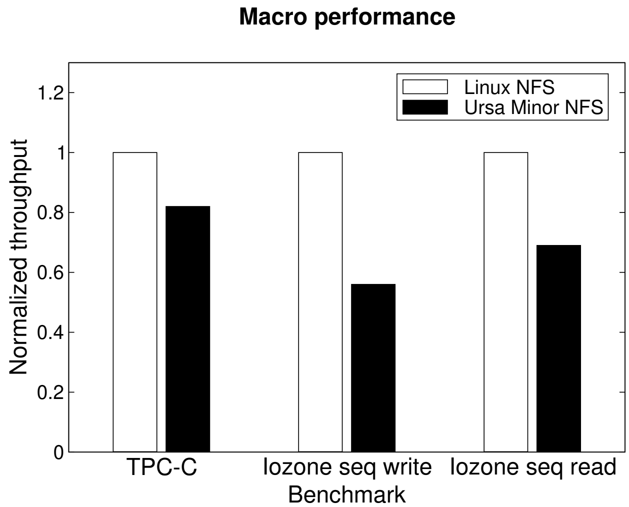

Figure 4: **Macro-benchmark performance.** This figure shows the normalized performance of the Ursa Minor NFS prototype against the Linux kernel NFS server in asynchronous mode. The Linux NFS server achieved 831 TpmC for TPC-C, 32 MB/s write throughput for Iozone, and 49 MB/s read throughput for Iozone.

the Linux server's. We view this difference as reasonable for experimentation. Specifically, the prototype implementation is suitable for experimentation with regard to the value of versatility.

4.3 Ursa Minor: Versatility

This section reports the results of several experiments that demonstrate the value of versatility. This evaluation focuses on accessing Ursa Minor directly through the client library and not through the Ursa Minor NFS server. The first two experiments consider matching distributions to workloads, the third experiment considers matching block size to request size, and the fourth experiment considers the performance cost of different storage-node fault models.

4.3.1 Specializing the data distribution

The performance and reliability of data stored in a cluster-based storage system is heavily influenced by the distribution chosen for that data. By providing versatility, a system allows the data distribution to be matched to the requirements of the workload. Without this versatility, all workloads are forced to use a single distribution that is expected to perform decently on a variety of workloads. Such compromise can lead to a significant decrease in performance, fault tolerance, or other properties.

In order to explore the trade-offs in choosing data distributions, we use four synthetic workloads to illustrate some common workloads:

Trace This represents a workload of trace analysis, common in research environments. It consists of large streaming reads with a request size of 96 KB. We assume that this workload's data must be able to tolerate two storage-node crash failures.

OLTP This workload simulates an OLTP database workload. It consists of random 8 KB reads and writes in a 1:1 ratio. We assume that this workload data must tolerate two storage-node crash failures, under the assumption that such information is costly to lose.

Name	Encoding	*m*	*t*	*n*	Block size	Storage efficiency
Trace	Erasure coding	2	2	4	96 KB	0.50
OLTP	Replication	1	2	3	8 KB	0.33
Scientific	Replication	1	0	1	96 KB	1.00
Campus	Replication	1	1	2	8 KB	0.50
Generic	Replication	1	2	3	24 KB	0.33

Table 1: **Distributions.** This table describes the distributions for encoding data for the experimental results in Figures 1 and 5. The choice for each specific workload is the best-performing option that meets its reliability requirements and uses four or fewer storage-nodes. The "generic" distribution meets each workload's fault tolerance requirements and performs well across the set of workloads.

Scientific This workload simulates the temporary data generated during large scientific calculations. It consists of sequential reads and writes with a 1:1 ratio and 96 KB requests. Because this data is easy to reconstruct, it does not need to tolerate any failures.

Campus This workload is based on the analysis of the Harvard CAMPUS NFS trace, a mainly email workload, by Ellard et al. [11]. As described by Ellard et al., the workload contains mostly sequential accesses ($\approx 90\%$) with some random accesses, and approximately equal ratio of reads and writes ($\approx 55\%$ reads). We assume that this workload's data must tolerate one storage-node crash failure.

These workloads represent environments with different access patterns and different concerns about reliability and storage space.

Figure 1 on page 2 shows the trade-off in distribution choice for these four workloads. We ran an experiment for each $\langle$workload, distribution$\rangle$ pair. In each such experiment, four storage-nodes are employed, and four clients run the given workload with the specified distribution. Each client accesses a single 300 MB object. Because storage-nodes have a 16 MB data cache and 32 MB metadata cache, this experiment illustrates the performance of workloads that are disk-bound. The clients in these experiments access the storage-nodes directly, and do not pass through the Ursa Minor NFS server.

For each workload, we first determine a specialized distribution that provides the workload with the highest performance given a four client and four storage-node system configuration. We measure the throughput of the system in each configuration over ten second intervals. A small configuration is chosen to simplify reasoning about the results; in practice much larger configurations are expected. We also determine a good "generic" distribution that provides good all around performance for the four workloads. These distributions are identified in Table 1, along with their storage efficiencies. *Storage efficiency* corresponds to the space-efficiency of the erasure code or replication used to store data; it is calculated as $\frac{m}{n}$. Note that n is less then 4 for some of these distributions; in these cases, clients access different subsets of n of the 4 storage-nodes, depending on the block being accessed. A simple function of the block identifier is used to assign blocks to sets of n storage-nodes with a uniform distribution.

Figure 1 shows each workload run on each of the five distributions in Table 1. As expected, the results show that specializing the distribution to the workload yields increased performance. The performance of a workload on a distribution specialized to another workload is poor, resulting in up to a factor of ten drop in performance. The generic distribution leads to more than a factor of two drop in performance for many of the workloads. Even for the Trace workload, for which the generic distribution demonstrates its best performance, the generic distribution performs 7% worse than the Trace distribution.

Each of the four workloads does best when using a different data distribution. For example, given a set of four storage-nodes, the best acceptable encoding for the Trace workload is 2-of-4 erasure coding because it provides good space-efficiency as well as good performance. A 1-of-1 scheme (the "Scientific

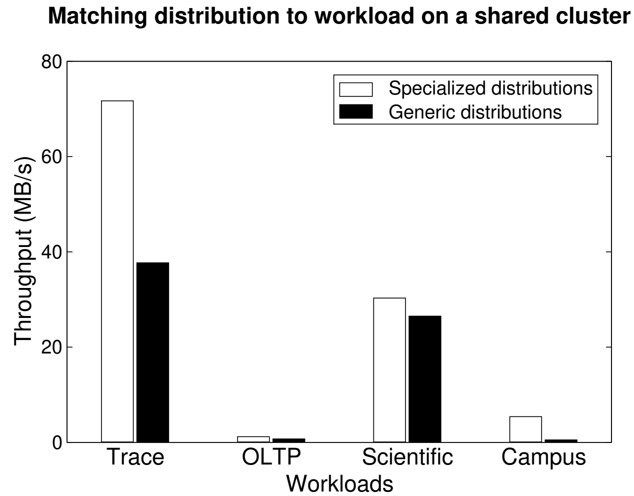

Figure 5: **Matching distribution to workload on a shared cluster.** This experiment shows the performance of the four workloads when they are run concurrently on a shared set of storage-nodes. The results show that by specializing the distribution for each workload, the performance in aggregate as well as the performance for the individual workloads improves markedly.

distribution" bar) provides better performance but does not satisfy the fault-tolerance requirement. A 1-of-3 scheme (3-way replication) provides similar performance, but requires 50% more storage space, a costly change for large data sets like traces. A replicated encoding is best for OLTP because it uses just one storage-node per read request (for data access). The smallest allowable amount of redundancy (i.e., the smallest t) is best, for each workload, both to minimize the capacity overheads and to minimize the cost of writes.

The Scientific workload performs best with a 1-of-1 encoding because this incurs the lowest cost for writes. As well, because of our experimental set-up, with four clients and four storage-nodes, this places each client on a separate storage-node. Isolating clients leads to an even higher performance difference from the other distributions because, currently, the storage-node implementation may intersperse blocks from different sequential writes within the same log segment, reducing the efficacy of subsequent sequential reads. The best encoding for the Campus workload is a 1-of-2 scheme, which incurs the lowest cost in terms of writes while still providing the required fault tolerance.

4.3.2 Sharing the Ursa Minor cluster

The Ursa Minor vision is to provide a single storage infrastructure suitable for hosting many different workloads, potentially at the same time. As such, we perform experiments to determine the impact of sharing a cluster among workloads and of matching distributions to workloads in a shared cluster. In the first experiment, each concurrently-running workload uses the distribution that is specialized for it in isolation. In the second, they all run the generic distribution. Figure 5 shows the performance of each workload when all four are run concurrently on the same set of storage-nodes. Specializing the distribution to the workload gives improvements to all of the workloads, with improvements ranging from 14% for the Scientific workload to $10\times$ for the Campus workload.

This illustrates two important points. First, it shows that the cost of using a one-size-fits-all distribution is high. Moving from the specialized distribution for each workload to the generic distribution for each workload causes the the aggregate throughput of the storage-nodes to drop over 40%, from 109 MB/s to 65.5 MB/s. Second, it shows that specialization of the distribution for each workload, in general, improves

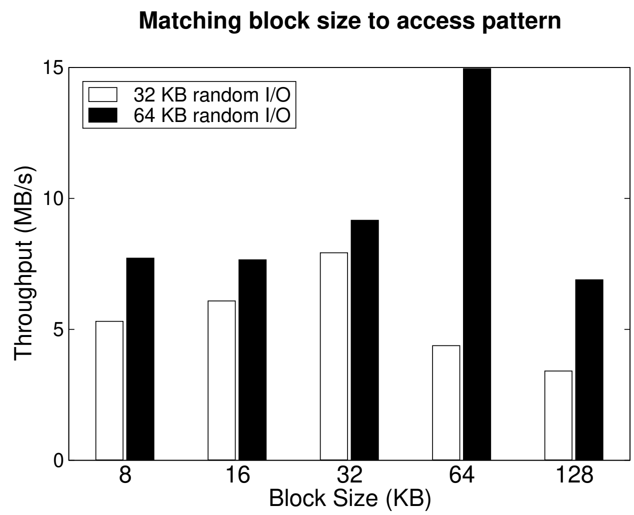

Figure 6: **Block size.** This graph illustrates the importance of matching Ursa Minor block size to application block size. In this experiment, a client performs random read and writes of size 32 KB, aligned on 32 KB block boundaries, or 64 KB aligned on 64 KB boundaries. The block size of the object is varied between 8 KB and 128 KB. This experiment uses a single client and a single storage-node.

the performance of not only the specific workload, but also the other workloads in the system, by freeing up more resources for the other workloads to use.

4.3.3 Specializing the block size

Block size is an important factor in performance. Figure 6 shows the effect of block size on performance of two workloads in Ursa Minor. It shows the throughput for a single client issuing an equal numbers of read and write requests to a single storage-node (m=n=1) when the client request size is kept constant, while the storage block size is varied. The graph shows the performance at each block size for two experiments: one with a client using a 32 KB request size and the other with a client using a 64 KB request size. Performance is best when Ursa Minor uses a block size that matches that of the workload. When the block size is smaller than the client request size, accesses have to be split into multiple requests. When the block size is too large, reads must fetch unnecessary data and writes must perform read-modify-write. Matching block size to request size is an important aspect of matching a distribution to a workload.

4.3.4 Specializing the fault model

Ursa Minor provides fault model versatility, allowing the number and types of failures tolerated to be configured on a per-object basis. Applications that can accept some risk with regard to reliability should not pay the capacity and performance costs associated with high degrees of fault tolerance. Yet, it is important to provide sufficient fault tolerance for important data. Fault model versatility can accommodate both in the same system. For example, most systems tolerate only storage-node crashes, ignoring software failures that corrupt data, lose updates, or otherwise provide incorrect responses to clients. These things do happen, usually due to software or firmware bugs. Tolerating Byzantine storage-node failures will mask such problems and even actively malicious behavior. Similarly, most systems assume synchrony, bounded delays that allow use of synchronized clocks and timeout-based failure identification. But, assuming synchronous systems can lead to corrupt data when requests take too long due to transient network failures/partitions,

Storage-node fault model	Faults tolerated	Timing model	n	OLTP
Byzantine	$t = b = 1$	Asynchronous	5	7 MB/s
Byzantine	$t = b = 1$	Synchronous	3	10 MB/s
Byzantine	$t = b = 2$	Synchronous	5	5 MB/s
Crash	$t = 1, b = 0$	Synchronous	2	13 MB/s

Table 2: **Fault models.** This table lists the aggregate throughput for different types of failures. It shows the number and type of fault, the timing model, n, and the throughput for the OLTP workload. In all cases, replication ($m = 1$) is used.

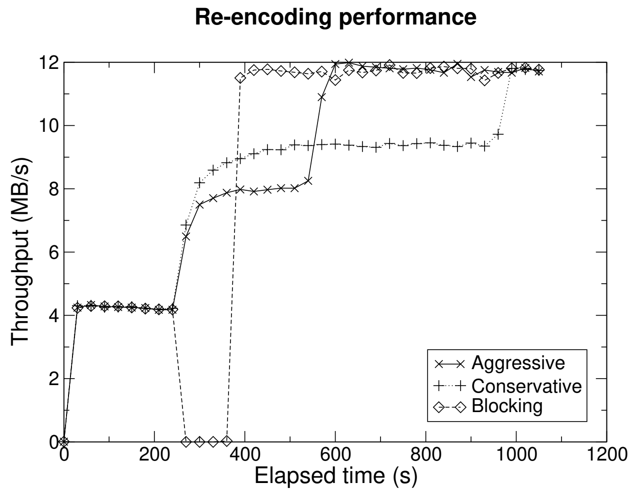

Figure 7: **Migration and re-encoding.** This graph shows the effect of migration and re-encoding as a function of the coordinators aggressiveness.

oversized request queues, server reboots, and the like. Removing the synchrony assumption (i.e., assuming asynchrony) masks such failures. Ursa Minor's protocol family allows these choices to be made on a per-data object basis.

Table 2 shows the performance of the OLTP workload when tolerating different types of faults. This experiment uses 8 clients and 5 storage-nodes. Going from one synchronous Byzantine failure to two synchronous Byzantine failures or one asynchronous Byzantine failure results in a 50% or 30% decrease in performance, respectively, because n is higher and each request involves more storage-nodes. This is a sizeable performance drop and it would likely be unacceptable if it would affect all data in a multi-purpose storage system. But, the resulting robustness benefits could be required for critical data.

4.4 Ursa Minor: On-line change

To illustrate the efficiency of re-encoding data to match workload access characteristics and the subsequent benefits, we constructed a synthetic workload that accesses a 1 GB object randomly (with a read-to-write ratio of 1:2) with an access block size of 64 KB. Originally, the block size for the data is 128 KB. Subsequently, it is re-encoded to use a 64 KB block size and simultaneously migrated to a second storage-node, to match the access patterns.

Figure 7 illustrates the efficiency of re-encoding the data stream as a function of distribution coordinator aggressiveness. The coordinator's aggressiveness is a tunable parameter that can be adjusted to be high

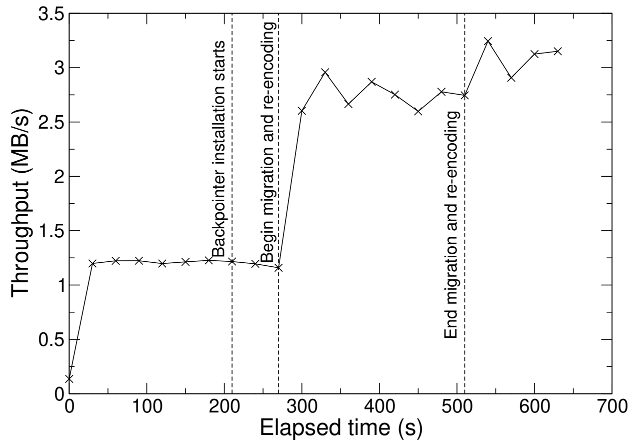

Figure 8: **Re-encoding of a live database system.** This graph shows the positive effect of re-encoding on the throughput that the TPC-C benchmark sees when accessing the underlying database. Re-encoding changes the default block size of 64 KB to match the client's request size of 8 KB.

if the workload's read-to-write ratio is high (e.g., greater than 1) and low if the ratio is low. The incremental process is contrasted to another way of re-encoding that blocks access to the object until re-encoding completes.

Ursa Minor's way of incrementally changing the distribution has minimal impact on the foreground workload and completes within a reasonable amount of time. This is true for both the back-pointer installation period and the coordinator copy period. As well, for a write-mostly workload, such as this, the role of the coordinator is less important since the workload's writes implicitly complete the re-encoding process.

Figure 8 illustrates the process of re-encoding for the TPC-C benchmark, from Section 4.2, running over the Ursa Minor NFS server. In this setup, 10 clients access one warehouse with a footprint of approximately 500 MB. The observed access size for TPC-C is 8 KB and the initial block size for the database stream is 64 KB. Hence, unnecessary cost is paid especially when writing to the database. Writing 8 KB pages incurs the cost of first reading a 64 KB block and then performing the write. The re-encoding procedure re-encodes the database file incrementally so that the new block size is 8 KB. The TPC-C workload is write-mostly, hence most of the benefit in re-encoding comes immediately after back-pointers have been installed. At that time, all writes are done with the new block size.

4.5 Case study: Diversity in the real world

To further illustrate that versatility is important in practice, we analyze three NFS traces from Harvard: EECS03 (February 2003), DEAS03 (February 2003), and Campus (October 2001) [11]. EECS03 is a typical engineering workload from Harvard's Electrical Engineering and Computer Science department. DEAS03 contains a mix of research and development, administrative, and email traffic from Harvard's Division of Engineering and Applied Sciences. Campus traces the home directories for the Harvard College and Harvard Graduate School of Arts and Sciences.

Figure 9 shows the file size distributions as well as the on-disk footprint due to files of different sizes. The results match common storage systems wisdom: most files are small, but most data is in large files. To illustrate the value of versatility, we consider the difference, in terms of storage efficiency, if specialized encodings are used for small files relative to large files. Specifically, we consider the use of three-fold replication for small files compared with a 4-of-6 erasure code for large files. Replication generally enables efficient random access, whereas 4-of-6 erasure coding generally enables efficient sequential access. Both

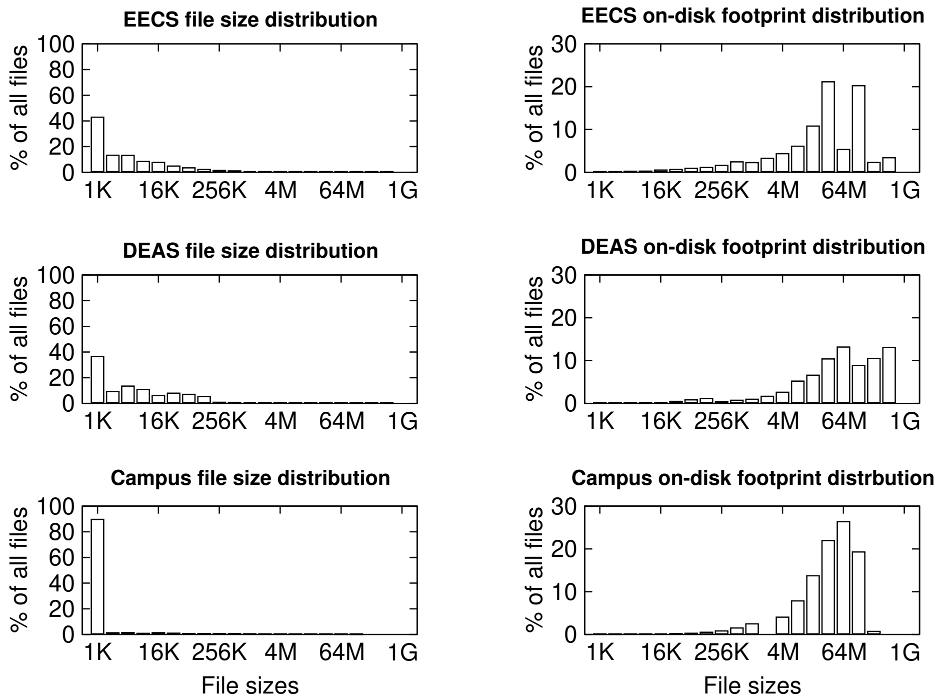

Figure 9: **File size and on-disk footprint distributions.** The file size distribution graphs show the percent of all files seen in each trace, binned by file size. These graphs show that most files are small. The on-disk footprint distribution graphs show the contribution to the total on-disk footprint by files of the given sizes. These graphs show that most storage space is consumed by large files.

distributions tolerate two storage-node failures, however the replication distribution requires a $3\times$ storage blowup to do so, whereas the erasure coding distribution requires only a $1.5\times$ storage blowup. Table 3 lists the overall blowup if a cutoff of 64 KB file size is chosen for small files to be replicated and large files to be erasure-coded. From the perspective of storage capacity, it is essentially free to use replication just for small files in these workloads.

5 Conclusions

Versatility is an important feature for storage systems. Ursa Minor enables versatility in cluster-based storage, complementing cluster scalability properties with the ability to specialize the data distribution for each piece of data. Experiments show that specializing these choices to access patterns and requirements can improve performance by a factor of two or more for multiple workloads. Further, the ability to change these choices on-line allows them to be adapted to observed access patterns and changes in workloads or requirements.

An important complement to versatility is exploiting it properly. The resulting tuning effort currently placed on administrators needs to be shifted to automated tools. For example, even for our experiments, selecting the best choices shown in graphs like Figure 1 required significant exploration and expertise. Automated tuning tools, such as Hippodrome [4] and Polus [43], will be crucial in the fight against administrative overload. We believe that inherent versatility and ability to change dynamically will make these tools easier to exploit.

Trace	Storage blowup
EECS	1.54×
DEAS	1.53×
Campus	1.51×

Table 3: **Storage blowup for 64 KB file size cutoff.** This table lists the storage blowup for three traces, given a file size cutoff of 64 KB. If a file is smaller than 64 KB, it is encoded via three-fold replication, yielding a 3× storage blowup. Otherwise, it is encoded via 4-of-6 erasure coding, yielding a 1.5× storage blowup.

Acknowledgements

We thank Dan Ellard and Margo Seltzer for sharing the NFS traces.

References

[1] *RSA Data Security, Inc. MD5 Message-Digest Algorithm*. http://www.ietf.org/rfc/rfc1321.txt.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–11 December 2002), pages 1–15. USENIX Association, 2002.

[3] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 22–25 October 2000), 2000.

[4] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 175–188. USENIX Association, 2002.

[5] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, **14**(1):41–79. ACM Press, February 1996.

[6] Bruno Becker, Stephan Gschwind, Thomas Ohler, Peter Widmayer, and Bernhard Seeger. An asymptotically optimal multiversion b-tree. *VLDB Journal*, **5**(4):264–275, 1996.

[7] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. *ACM SIGMOD International Conference on Management of Data* (Minneapolis, MN, 24–27 May 1994). Published as *SIGMOD Record*, **23**(2):383–394, 1994.

[8] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 1–14. USENIX Association, 2004.

[9] Transactional Processing Performance Council. TPC Benchmark C. Number Revision 5.1.0, 2002.

[10] Wei Dai. *Crypto++ reference manual*. http://cryptopp.sourceforge.net/docs/ref/.

[11] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 203–216. USENIX Association, 2003.

[12] EMC Corp. EMC Centera: content addressed storage system, August, 2003. http://www.emc.com/-products/systems/centera.jsp?openfolder=platform.

[13] EqualLogic Inc. PeerStorage Overview, 2003. http://www.equallogic.com/pages/products_technology.htm.

[14] Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. FAB: enterprise storage systems on a shoestring. *Hot Topics in Operating Systems* (Lihue, HI, 18–21 May 2003), pages 133–138. USENIX Association, 2003.

[15] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. *Self-* Storage: Brick-based storage with automated administration*. Technical Report CMU–CS–03–178. Carnegie Mellon University, August 2003.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM Symposium on Operating System Principles* (Lake George, NY, 10–22 October 2003), pages 29–43. ACM, 2003.

[17] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[18] Li Gong. Securely replicating authentication services. *International Conference on Distributed Computing Systems* (Newport Beach, CA), pages 85–91. IEEE Computer Society Press, 1989.

[19] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. *The safety and liveness properties of a protocol family for versatile survivable storage infrastructures*. Technical report CMU–PDL–03–105. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[20] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, **13**(1):124–149. ACM Press, 1991.

[21] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12**(3):463–492. ACM, July 1990.

[22] IBM Almaden Research Center. Collective Intelligent Bricks, August, 2003. http://www.almaden.ibm.com/StorageSystems/autonomic_storage/CIB/index.shtml.

[23] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, **45**(3):451–500. ACM Press, May 1998.

[24] Steve Kleiman. Personal communication, October 2002.

[25] Andrew J. Klosterman and Gregory Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[26] Hugo Krawczyk. Secret sharing made short. *Advances in Cryptology - CRYPTO* (Santa Barbara, CA, 22–26 August 1993), pages 136–146. Springer-Verlag, 1994.

[27] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.

[28] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[29] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine storage. *International Symposium on Distributed Computing* (Toulouse, France, 28–30 October 2002), 2002.

[30] W. Norcott. IOzone Benchmark, http://www.iozone.org/, 2001.

[31] *Object based storage devices: a command set proposal*. T10/99-315R0. October 1999. http://www.T10.org/.

[32] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. *Summer USENIX Technical Conference* (Monterey, CA, 06–11 June 1999). USENIX Association, 1999.

[33] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD International Conference on Management of Data* (Chicago, IL), pages 109–116, 1–3 June 1988.

[34] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 89–101. USENIX Association, 2002.

[35] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, **36**(2):335–348. ACM, April 1989.

[36] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003). USENIX Association, 2003.

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

[38] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. *Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 09–13 October 2004), pages 48–58. ACM, 2004.

[39] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. *International Conference on Distributed Computing Systems* (Cambridge, Mass), pages 198–204. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.

[40] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 43–58. USENIX Association, 2003.

[41] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

[42] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. *Symposium on Reliable Distributed Systems* (10–12 October 1988), pages 93–100. IEEE, 1988.

[43] Sandeep Uttamchandani, Kaladhar Voruganti, Sudarshan Srinivasan, John Palmer, and David Pease. Polus: growing storage QoS management beyond a "4-year old kid". *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 31–44. USENIX Association, 2004.

[44] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. *First International Workshop on Peer-to-Peer Systems (IPTPS 2002)* (Cambridge, MA, 07–08 March 2002), 2002.

[45] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

[46] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. *IEEE Computer*, **33**(8):61–68. IEEE, August 2000.

[47] Kenneth G. Yocum, Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Anypoint: extensible transport switching on the edge. *USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, 26–28 March 2003), 2003.

[48] Zheng Zhang, Shiding Lin, Qiao Lian, and Chao Jin. RepStore: a self-managing and self-tuning storage backend with smart bricks. *International Conference on Autonomic Computing* (New York, NY, 17–18 May 2004), pages 122–129. IEEE, 2004.