Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST '05).
San Francisco, CA. December 13-16, 2005.

Ursa Minor: versatile cluster-based storage

Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger,
James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad,
Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen,
John D. Strunk, Eno Thereska, Matthew Wachs, Jay J. Wylie
Carnegie Mellon University

Abstract

No single encoding scheme or fault model is optimal for all data. A versatile storage system allows them to be matched to access patterns, reliability requirements, and cost goals on a per-data item basis. 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 specialized choices, rather than suffering from "one size fits all." Experiments with Ursa Minor show performance benefits of 2–3× when using specialized choices as opposed to a single, more general, configuration. Experiments also show that a single cluster supporting multiple workloads simultaneously is much more efficient when the choices are specialized for each distribution rather than forced to use a "one size fits all" configuration. When using the specialized distributions, aggregate cluster throughput nearly doubled.

1 Introduction

Today's enterprise storage is dominated by large monolithic disk array systems, extensively engineered to provide high reliability and performance in a single system. However, this approach comes with significant expense and introduces scalability problems, because any given storage enclosure has an upper bound on how many disks it can support. To reduce costs and provide scalability, many are pursuing cluster-based storage solutions (e.g., [2, 8, 9, 10, 11, 12, 18, 23]). 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 [27] and arguments for cluster computing over monolithic supercomputing.

Cluster-based storage has scalability and cost advantages, but most designs lack the versatility commonly found in 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 stored within a single 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, data accessed with large, sequential I/Os often should be erasure coded to reduce the capacity and bandwidth costs of fault tolerance, while randomly-accessed data often should be replicated to minimize the number of disk I/Os per data access.

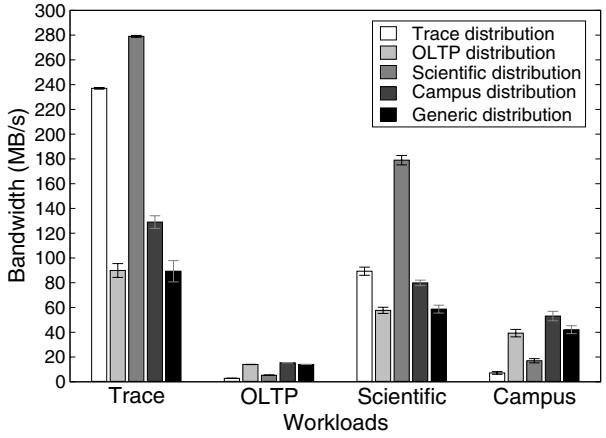

Figure 1: Matching data distribution to workload. This graph shows the performance of four workloads run on Ursa Minor as a function of the data distribution. For each workload, five distributions were evaluated: the best distribution for each of the four workloads and a generic "middle of the road" choice for the collection of workloads. Although the "Scientific" data distribution provided better performance for the "Trace" workload than the "Trace" distribution, and the "Campus" data distribution provided better performance for the "OLTP" workload than the "OLTP" distribution, these distributions failed to meet the respective workloads' reliability requirements. Section 4.3 details the workloads and data distributions. These numbers are the average of 10 trials, and the standard deviations are shown as error bars.

This paper describes Ursa Minor, a versatile, cluster-based storage system designed to allow the selection of, as well as on-line changes to, the data location, encoding, block size, and fault model on a per-"data object" basis. 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. Using a protocol family shifts the decision of which types of faults to mask from system implementation time to data creation time. This allows each object within a single infrastructure to be protected from the types and quantities of faults appropriate to that particular class of data. Ursa Minor's protocol family supports a per-object choice of *data distribution*. This includes the data encoding (replication or erasure coding), block size, storage-node fault type (crash or 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 the data distribution choice is specialized to access patterns and fault tolerance requirements. These benefits remain even when multiple workload types share a storage cluster. In addition to performance benefits, capacity benefits are also realized when erasure coding is used instead of replication. 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 10% performance penalty is paid for doing this, because the 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 as much as a factor of three over a reasonable "one size fits all" configuration.

Ursa Minor's ability to support on-line data distribution change is also demonstrated. The ability to reconfigure data distributions on-line enables tuning based on observed usage rather than expected usage. This simplifies tuning, since pre-deployment expertise about an application's access patterns becomes less important. Minimizing the amount of pre-deployment expertise and planning is important for reducing the excessive administration effort required with today's storage infrastructures. Additionally, the ability to make on-line distribution changes allows the system to be adapted as goals and workloads evolve.

This paper makes the following contributions. First, it makes a case for versatile cluster-based storage, demonstrating that versatility is needed to avoid significant performance, reliability, and/or cost penalties when storage is shared among different classes of data. Second, it describes the design and implementation of Ursa Minor, a versatile, cluster-based storage system. We are aware of no existing cluster-based 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 specializing according to access patterns and reliability requirements as well as the value of allowing on-line changes to data distributions.

2 Versatile cluster-based storage

Today's enterprise storage systems are typically monolithic and very expensive, based on special-purpose, high-availability 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 they do so at great monetary expense.

Cluster-based storage is a promising alternative to today's monolithic storage systems. The concept is that collections of smaller storage-nodes should be able to provide performance and reliability competitive with today's high-end solutions, but at much lower cost and with greater scalability. The cost reductions would come from using commodity components for each storage-node and exploiting economies of scale. Each storage-node 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 redundancy and workload distribution across storage-nodes. If successful, cluster-based storage should be much less expensive (per terabyte) than today's enterprise storage systems, while providing similar levels of reliability and availability [10].

Cluster-based storage also helps with the scaling challenges inherent in 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 systems. Avoiding this drastic step-function in effort and capital expenditure can push administrators to purchase oversized (but mostly empty) systems. Most cluster-based storage designs allow growth of capacity and performance through the in-

cremental addition of storage-nodes, with automated balancing of the data to utilize the new resources.

2.1 Versatility in cluster-based storage

To replace monolithic storage effectively, cluster-based storage must provide similar versatility. It must be possible to specialize the data distribution for different classes of data and their respective workloads.

This section describes several choices that should be specialized to individual data based on application requirements (e.g., fault tolerance and performance goals), access patterns, and cost restrictions. Almost all modern disk array systems allow the encoding 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 a greater impact than in the centralized controller architecture of monolithic storage systems.

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

There are two common encoding 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.[1] 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 disk-bound throughput for non-sequential accesses. On the other hand, erasure coded data can tolerate failures (especially multiple failures) with less network bandwidth and storage space [40, 43]. 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 components than traditional 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 the importance of tolerating multiple disk failures [6]. 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 tradeoffs, however, the number of failures tolerated must be configurable.

Fault model choices: In traditional systems, a centralized controller provides a serialization point, single restart location, and an unambiguous storage-node (i.e., disk) failure indication. In contrast, most cluster-based storage designs are decentralized systems, enjoying none of these luxuries. 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, and 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 otherwise not cooperating). 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 make a system more robust at the expense of additional data redundancy and communication.

It is tempting to assume that tolerating storage-node crashes is sufficient and that good engineering can prevent Byzantine (i.e., non-crash) 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 significant 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 [20] that disks occasionally write data sectors to the wrong location.[2] Such a fault corrupts two pieces of data: the old version of the 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

[1] RAID 5 is an $(n-1)$-of-n scheme.

[2] 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.

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 the data distribution choices could be adjusted based on the actual access pattern. Even the number and type of faults tolerated could be changed based on the problems observed in practice.[3]

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 a storage infrastructure. Instead, a trial-and-error approach could be used to arrive at an acceptable system configuration. Additionally, on-line change can be invaluable as access patterns and goals change over the course of the data's lifecycle.

2.2 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 work for specific mechanisms are discussed with those mechanisms.

Many scalable cluster-based storage systems have been developed over the years. Petal [23], xFS [2], and NASD [13] are early systems that laid the groundwork for today's cluster-based storage designs, including Ursa Minor's. More recent examples include FARSITE [1], FAB [34], EMC's Centera [8], EqualLogic's PS series product [9], Lustre [24], Panasas' ActiveScale Storage Cluster [26], and the Google file system [12]. All of these systems provide the incremental scalability benefits of cluster-based storage, as well as some provisions for fault tolerance and load balancing. However, each of them hard-codes most data distribution choices for all data stored in the system. For example, Petal replicates data for fault tolerance, tolerates only server crashes (i.e., fail-stop storage-nodes), and uses chained declustering to spread data and load across nodes in the cluster; these choices apply to all data. xFS also uses one choice for the entire system: parity-based fault tolerance for server crashes and data striping for load spreading. Ursa Minor's design builds on previous cluster-based storage systems to provide versatility. Its single design and implementation supports a wide variety of data distribution choices, including encoding scheme, fault model, and timing model. All are selectable and changeable on-line on a per-object basis.

FAB [34] and RepStore [45] offer two encoding scheme choices (replication or erasure coding) 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, uses AutoRAID-like [41] 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 that handle non-crash failures. Compared to FAB, Ursa Minor also supports re-encoding of data, allowing configuration choices to be modified on-line.

Pond [30] 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 is to provide scalability by interposing a proxy [35], such as Mirage [3], Cuckoo [21], or Anypoint [44]. Proxies can spread data and requests across servers like a disk array controller does with its disks. This approach to building a storage infrastructure represents a middle-ground between traditional and cluster-based storage.

AutoRAID [41] automates 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 many of the benefits from 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] 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 likely encounter more failures than one in a more hospitable environment.

3 Ursa Minor

Ursa Minor is a versatile cluster-based storage system. Its design and implementation grew from the desire to provide a high level of versatility in a cost-effective, cluster-based storage system.

3.1 Architecture

Ursa Minor provides storage of *objects* in the style of NASD [13] 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, etc. The primary difference from file systems is that there are no ASCII names or directories.

The main 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 metadata 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 [13]. 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 the object manager, which provides them with metadata and authorization. Afterward, they can interact directly with the storage-nodes for data operations. Metadata operations, such as object creation and deletion, are done through the object manager.

Much of Ursa Minor's versatility is enabled by the read/write protocol family it uses for data access [15]. 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. The protocol family allows each block 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 a sequence of blocks with a common block size, encoding scheme, data location, fault model, and timing model. Different slices within the same object can have different values for any of these choices. Slices allow large objects to be partitioned across multiple sets of storage-nodes. Although slices are integral to the Ursa Minor design, to simplify discussion, most of this paper refers to the data distribution of objects rather than of slices of objects.

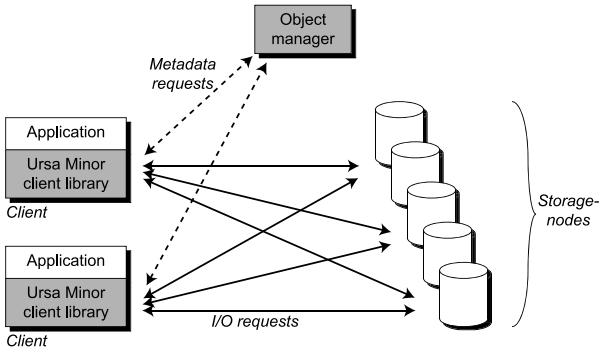

Figure 2: Ursa Minor high-level architecture. Clients use the storage system via the Ursa Minor client library. The metadata needed to access objects is retrieved from the object manager. Requests for data are then sent directly to storage-nodes.

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 single block, and clients are not prevented from accessing the object's data during the distribution change. Such a data distribution change can alter the storage locations, encoding, fault model, timing model, or block size. Section 3.4 will describe this process in detail.

3.2 Protocol family for versatile access

Data access in Ursa Minor builds on a protocol family 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 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. Pseudo-code and proofs of correctness are available in separate technical reports [16, 42].

3.2.1 Protocol family versatility

The fault tolerance provided by 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. Each of these parameters provides tradeoffs for the performance, availability, and reliability of data.

Timing model: Protocol family members are either asynchronous or synchronous. Asynchronous members rely on no timeliness assumptions. There are no assumptions about message transmission delays or execution rates of storage-nodes. In contrast, synchronous members assume known bounds on message transmission delays between correct clients and storage-nodes as well as request processing times.

By assuming a synchronous system, storage-node crashes are detectable via timeouts. This allows clients to contact fewer storage-nodes for each operation and trust that they will get an answer from all non-faulty storage-nodes. On the other hand, if a client incorrectly "detects" that a live storage-node timed out (e.g., due to overload or a network partition), it may read inconsistent data. The asynchronous timing model is able to protect against this scenario but at the cost of additional storage-nodes and an additional round trip during writes to generate a logical timestamp.

Storage-node failure model: Each family member supports a hybrid failure model [38] for storage-nodes. Up to t storage-nodes may fail. A subset of the t failures, b, may be Byzantine faults [22], and the remaining $t - b$ must only be crash failures. Such a model can be configured across the spectrum from wholly crash (i.e., $b = 0$) to wholly Byzantine (i.e., $b = t$).

The number or storage-nodes that must be contacted for each operation increases with the number of failures that the protocol is configured to tolerate. Tolerating Byzantine failures increases the number of storage-nodes still farther. By choosing a configuration that can withstand Byzantine storage-node failures, data is protected from data corruption by storage-nodes, disk firmware errors, and buggy software.

Client failure model: Every member of the protocol family tolerates an arbitrary number of 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 client (Byzantine or otherwise) can write arbitrary values to storage. Protecting against Byzantine clients ensures only that the values written by a client are consistent (i.e., all clients reading a given version will observe the same value), not that the data itself is non-malicious. Although the implementation supports them, we do not employ the Byzantine client mechanisms in our evaluation of Ursa Minor.

3.2.2 Protocol guarantees and constraints

All members of the protocol family guarantee linearizability [17] of all correct operations. To accomplish this, the number of storage-nodes (and thus the number of fragments, n) must conform to constraints with regard to b and t (the number of storage-node failures) as well as m (a data encoding parameter). For asynchronous members, the constraint is $2t + b + \max(m, b + 1) \leq n$. For synchronous members, the constraint is $t + \max(m, b + 1) \leq n$. Full development and proof sketches for these and other relevant constraints (e.g., read classification rules) are presented by Goodson, et al. [16].

3.2.3 Protocol operation and implementation

Each protocol family member supports read and write operations on arbitrarily-sized blocks. 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 [29] or replication) could be used. Logical timestamps associated with each block totally order all write operations and 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). Clients form this timestamp either by issuing GET_LOGICAL_TIME requests to storage-nodes (for asynchronous members) or reading the local clock (for synchronous members). Each of the n fragments is sent to its storage-node, tagged with the logical timestamp. Storage-nodes provide fine-grained versioning, retaining a fragment version (indexed by logical timestamp) for each write request they execute.

To read a block, a client issues read requests to a subset of the n 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—by far the most common case. Only in certain cases of failures or concurrency are incomplete or repairable candidates observed. If the candidate is classified as incomplete, it 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 fragments back to storage-nodes that do not have them (with the logical timestamp shared by the existing fragments). Then, the data is returned.

Because Byzantine storage-nodes can corrupt their data-fragments, it must be possible to detect and mask up to b storage-node integrity faults. *Cross checksums* [14] are used to detect corrupted data-fragments: a cryptographic

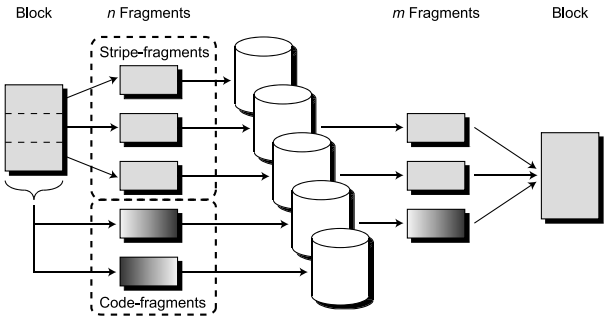

Figure 3: Erasure coding example. This example shows a 3-of-5 erasure encoding WRITE and then READ of a block. On WRITE, the original block is broken into three stripe-fragments and two code-fragments, then stored on five separate storage-nodes. On READ, any three fragments (stripe or code) can be retrieved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

hash of each data-fragment is computed, and the set of n hashes are concatenated to form the cross checksum of the data-item.[4] 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. Our implementation uses MD5 [32] for all hashes, but any collision-resistant hash could be substituted.

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 complete in the background. To make read operations more network efficient, only m read requests fetch actual fragment contents, while all fetch version histories. 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 used. 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 produces the original block. Because "decoding" with the m stripe-fragments is computationally less expensive, the implementation preferentially tries to read them. The stripe-fragments are used to generate *code-fragments* that provide the necessary redundancy (i.e., the remaining $n - m$ fragments) via Rabin's information dispersal algorithm [29]. Figure 3 illustrates how stripe- and code-fragments are stored.

3.3 Ursa Minor components

In addition to the protocol family used for read and write operations, Ursa Minor is composed of several key components: the storage-nodes store all data in the system; the object manager tracks system metadata and arbitrates access to objects; the client library encapsulates system interactions for applications; the NFS server allows unmodified clients to use the system.

Storage-node: Storage-nodes expose the same interface, regardless of the protocol family member being employed—read and write requests for all protocol family members are serviced identically. Clients communicate with storage-nodes via a TCP-based RPC interface. For write requests, storage-nodes provide an interface to write a fragment at a specified logical timestamp. For read requests, clients may request the version of a fragment with the greatest logical timestamp or a previous version by specifying a specific timestamp. Several other operations are supported, including retrieving the greatest logical timestamp of a fragment and retrieving a fragment's version history.

Requests to storage-nodes address data fragments by block number because the fragment size is not fixed. Fragment sizes vary for three reasons. First, the data 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 about in-progress distribution changes instead of data. On a write, the storage-node accepts whatever number of bytes the client sends and records them, indexed by the specified object ID, block number, and timestamp. 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. A log-structured organization [33] is used to reduce the disk I/O cost of data versioning. Multi-version b-trees [4, 36] are used by the storage-nodes to store fragments. Fragment versions are kept in a per-object b-tree indexed by a 2-tuple $\langle blocknumber, timestamp \rangle$. Like previous research [28, 37], 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 used to prevent capacity exhaustion of the storage-nodes. Because write completeness is a property of a set of storage-nodes, a storage-node in isolation cannot determine which local fragment versions are safe to garbage-collect. A fragment version can be garbage-collected only if there exists a later complete write for the corresponding block. Storage-nodes classify writes by executing the read pro-

[4] In the special case of replication, a single hash is sufficient.

tocol in the same manner as a client, excluding the actual data fetches. This garbage collection typically completes in the background, before writes are flushed to disk, and it can be batched across a number of data blocks.

The storage-node implementation is based on the S4 object store [36, 37]. It uses a write-back cache for fragment versions, emulating non-volatile RAM.[5] 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 queried just to ensure consistency (rather than to retrieve one of m fragments).

Object manager: The object manager maintains Ursa Minor metadata about each object, including data distribution. Clients send RPCs to the object manager to create and delete objects, access attributes, and retrieve 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. The 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, timing 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 [25] b-trees, stored in objects, to organize and index the Ursa Minor metadata. To enable crash recovery of the object manager, Berkeley 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. The "revocation" of capabilities is handled with callbacks to clients rather than communication with storage-nodes. Although not acceptable for deployment, this should not affect performance experiments.

Client library: The client library provides a byte-addressed object interface to application code, hiding the details of Ursa Minor. It includes a protocol library that, given the data distribution, handles the data encoding and protocol execution on behalf of the caller. The client library also hides other Ursa Minor details, such as interactions with the object manager. The client library is just a convenience for programmers, and it is not trusted by storage-nodes or object managers any more than application code.

[5] Our storage-nodes are battery-backed, but our implementation does not yet retain the cache contents across reboots.

NFS server: Access to data stored in Ursa Minor clearly involves non-standard protocols. To support unmodified clients, we have implemented a user-level NFS server that exports files and directories stored as objects in Ursa Minor. It supports UDP-based NFS version 3, and it uses the Ursa Minor client library to read and write data in the system. 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.

Such an NFS server is not intended as the primary method of access to a cluster-based storage system like Ursa Minor—a better choice being a parallel-access file system. However, our NFS server is convenient for incremental deployment.

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, risks, and goals 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 initial version of blocks in a new data distribution. This provides a link between the new distribution and the old, obviating the need to halt client access during the data re-encode step. 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 installs back-pointers to the old distribution by writing them to the storage-nodes that will store the new distribution. One back-pointer is written for each new block.[6] Second, the object manager revokes client access to the affected range of blocks. Third, the object manager updates its metadata with the new distribution and resumes issuing capabilities. 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 being 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

[6] This operation could be batched to improve the efficiency of installing back-pointers, but back-pointer installation is not a critical-path operation.

to access the identified old distribution. Once it encounters data, it proceeds normally. 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 the new (step four) is called a *distribution coordinator*. It copies data 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. The required gap in timestamps is created either by pausing after installing the back-pointers (in the synchronous case) or by reserving a fixed logical timestamp (in the asynchronous case).

One of the trickier aspects of data distribution change arises when the data block size is changed. Changes in the block size (used to break up the byte stream into blocks on which the protocol operates) will alter the number of blocks needed for a given byte range. This can cause conflicts between block numbers 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. A new range of block numbers within the object is used for the new distribution, eliminating any conflict and enabling the use of the fixed logical timestamp (mentioned above) for the asynchronous timing model.

Ursa Minor's approach to on-line distribution change minimizes blocking of client accesses and allows incremental application of change. Client access is only interrupted during the actual metadata update at the object manager. Further, the notion of slices allows a distribution change for a large object to be performed piecemeal rather than all at once. In addition, the coordinator can move data to the new distribution at whatever rate is appropriate. Since migration is tracked by the object manager and the distribution coordinator's actions are idempotent, coordinators that fail can be easily restarted.

4 Evaluation

This section evaluates Ursa Minor and its versatility in three specific areas. First, it verifies that the baseline performance of NFS with Ursa Minor is reasonable. Second, it shows that Ursa Minor's versatility provides significant benefits for different synthetic workloads. Third, it confirms that the Ursa Minor prototype can efficiently perform on-line changes of an object's data distribution.

4.1 Experimental setup

All experiments were run using Dell PowerEdge 650 machines equipped with a single 2.66 GHz Pentium 4 processor, 1 GB of RAM, and two Seagate ST33607LW, 36 GB, 10K rpm SCSI disks. The network configuration consisted of a single Intel 82546 gigabit Ethernet adapter in each machine, connected via a Dell PowerConnect 5224 switch. The machines ran the Debian "testing" distribution and used Linux kernel version 2.4.22. The same machine type was used both as clients and storage-nodes. The storage-nodes used one of the two local disks for data; the other contained the operating system.

4.2 Baseline NFS performance

This section uses application-level benchmarks to show that Ursa Minor achieves reasonable performance. The Ursa Minor NFS server's performance was compared to that of the Linux kernel-level NFSv3 server. Both NFS servers were configured to communicate with clients using UDP, and in both cases, they ran on dedicated machines. The Linux NFS server exported an ext3 partition that resided on a dedicated local disk. The Ursa Minor NFS server exported data stored on a single storage-node and was configured to use 384 MB of data cache and 32 MB of attribute cache. The storage-node had 640 MB of data cache and 64 MB of metadata cache.

The performance of the two systems was compared using the TPC-C and Postmark benchmarks as well as a simple source-tree compile benchmark. The TPC-C benchmark [39] 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. TPC-C was run on the Shore database storage manager [5] and configured to use 8 kB pages, 10 warehouses and 10 clients, giving it a 5 GB footprint. The Shore volume was a file stored on either the Linux NFS server or the Ursa Minor NFS server.

Postmark [19] is a user-level file system benchmarking tool designed to measure performance for small file workloads such as e-mail and netnews. It measures the number of transactions per second that the system is capable of supporting. A transaction is either a file create or file delete, paired with either a read or an append. The configuration parameters used were 50000 files, 20000 transactions, and 100 subdirectories. All other parameters were left as default.

We constructed the "um-build" benchmark to measure the amount of time to clean and build the Ursa Minor source tree. The benchmark copies the source tree onto a target system, then cleans and builds the Ursa Minor pro-

	Linux NFS	Ursa Minor
TPC-C	447 tpmC (2.3)	993 tpmC (13)
Postmark	17.9 tps (.01)	15.0 tps (0.0)
um-build	1069 s (5.3)	874 s (2.0)

Table 1: Macro-benchmark performance. This table shows several macro-benchmarks used to compare the performance of the Ursa Minor NFS prototype against the Linux NFS server. Standard deviations based on ten trials are listed in parentheses.

totype. The results provide an indication of storage system performance for a programming and development workload. The source tree contained 2144 files and grew from 24 MB to 212 MB when built.

Table 1 shows performance for random I/O (TPC-C and Postmark) and system development (um-build) workloads. Overall, the two systems performed comparably in these tests. The Ursa Minor storage-node's log-structured layout allowed it to perform better than the Linux NFS server for TPC-C and um-build. However, the extra network hop between the NFS server and storage-node added latency to I/O requests, hurting Ursa Minor's performance for Postmark. These results show the prototype implementation is suitable for an investigation into the value of versatility.

4.3 Ursa Minor: Versatility

This section reports the results of several experiments that demonstrate the value of Ursa Minor's versatility. These experiments access Ursa Minor directly via the client library, not through the Ursa Minor NFS server. The first three experiments explore matching distributions to workloads, and the fourth experiment shows the costs of different storage-node fault models.

For these experiments, the working set was larger than the combined client and storage-node caches. The storage-nodes used a 32 MB data cache and a 64 MB metadata cache, ensuring that most data accesses were served from disk and metadata (e.g., version history information) remained cached.

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 data distributions to be matched to the requirements of each dataset. Without this versatility, datasets are forced to use a single distribution that is expected to perform adequately 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, four synthetic workloads were chosen to represent environments with different access patterns and different concerns about reliability, capacity, and performance.

Trace: This simulates trace analysis, common in research environments. It was modeled as streaming reads with a request size of 96 kB. We assumed that this data must tolerate two storage-node crash failures, since trace data can be difficult to re-acquire.

OLTP: This simulates an OLTP database workload. It was modeled as random 8 kB reads and writes in a 1:1 ratio. We assumed that this data must tolerate two storage-node crash failures, since such information is costly to lose.

Scientific: This simulates the temporary data generated during large scientific calculations. It was modeled as sequential reads and writes with a 1:1 ratio, using 96 kB requests. Because this data is generally easy to reconstruct, it did not need to tolerate any failures.

Campus: This simulates general academic computing. It was based on an analysis of the Harvard CAMPUS NFS trace [7], a mainly email workload. It was modeled as a 90% sequential and 10% random access pattern, using 8 kB requests. Fifty-five percent of accesses were reads. We assumed that this data must tolerate one storage-node crash failure.

We ran an experiment for each ⟨workload, distribution⟩ pair. In each experiment, six storage-nodes were used, and twelve clients ran the given workload with the specified distribution. Each client accessed a single 150 MB object.

For each workload, we determined a specialized distribution that provides it with the highest performance given the twelve client and six storage-node system configuration. We warmed the cache, then measured the throughput of the system. After trying the workload on the subset of the possible distributions where the failure requirements and block size match the workload, we chose the encoding that was most space efficient but still had throughput within 10% of optimal.

We also determined a "generic" distribution that provided good all-around performance for the four workloads. In order to determine this encoding, we ran each of the workloads on the encodings that met the failure requirements of the most stringent workload. For each encoding, we tried an 8 kB block size and all block sizes that are multiples of 16 kB up to a maximum size of 96 kB. The "generic" distribution was chosen to minimize the sum of squares degradation across the workloads. The degradation of a workload was calculated

Workload	Encoding	*m*	*t*	*n*	Block size
Trace	Erasure coding	2	2	4	96 kB
OLTP	Replication	1	2	3	8 kB
Scientific	Replication	1	0	1	96 kB
Campus	Replication	1	1	2	8 kB
Generic	Replication	1	2	3	8 kB

Table 2: Distributions. This table describes the data encodings for the experimental results in Figures 1 and 4. The choice for each workload was the best-performing option that met the reliability requirements and used six or fewer storage-nodes. The "generic" distribution met all workloads' fault tolerance requirements and performed well across the set of workloads.

as the percentage difference in bandwidth between using the specialized distribution and the "generic" distribution. This penalized encodings that disproportionately hurt a specific workload. The distributions chosen for each workload, and the "generic" distribution are identified in Table 2.

Figure 1 on page 1 shows each workload using each of the five distributions in Table 2. As expected, specializing the distribution to the workload yields increased performance. The performance of a workload on a distribution specialized to another workload was poor, resulting in up to a factor of seven drop in performance. The generic distribution led to more than a factor of two drop in performance for many of the workloads. The one exception was OLTP, which performed the same with the generic encoding, since this encoding is the same as the best encoding for OLTP.

Each of the four workloads performed best when using a different data distribution. For example, the best encoding for the Trace workload was 2-of-4 erasure coding because it provided good space-efficiency as well as good performance. A 1-of-3 scheme (3-way replication) provided similar performance, but required 50% more storage space—a costly "feature" for large datasets like traces. A replicated encoding was best for OLTP because it used just one storage-node per read request (for data access). The smallest allowable amount of redundancy (i.e., the smallest t) was best, both to minimize the capacity overheads and to minimize the cost of writes.

The Scientific workload performed best with a 1-of-1 encoding because this incurred the lowest cost for writes. The best encoding for the Campus workload was a 1-of-2 scheme, which incurred the lowest number of I/Os 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 performed experiments to determine the impact of sharing a cluster among workloads while matching the distributions to those workloads. In the previous experiment, the specialized versus generic distributions were compared in isolation. For this experiment, all workloads are run simultaneously. Figure 4 shows the performance of each workload when all four were run concurrently on the same set of storage-nodes—first with the generic distribution, then with the specialized distributions. Specializing the distribution to the workload gave improvements to all of the workloads, ranging from 32% for the Trace workload to 171% for the Scientific workload.

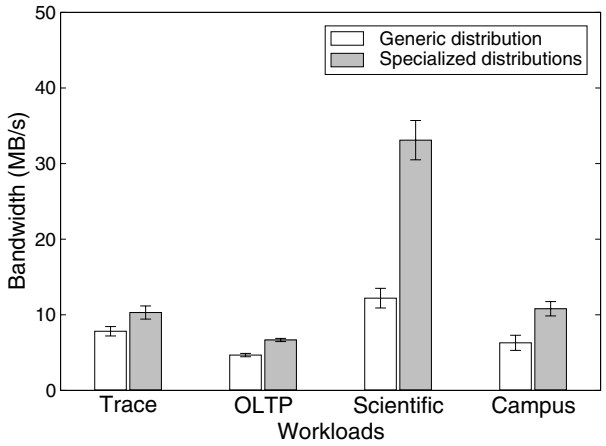

Figure 4: Matching distribution to workload on a shared cluster. This experiment shows the performance of the four workloads when they are run concurrently on a shared set of storage-nodes. The results show that, by specializing the distribution for each workload, the performance in aggregate as well as the performance for the individual workloads improves significantly. These numbers are the average of 10 trials, and the standard deviations are shown as error bars.

This shows that the cost of using a one-size-fits-all distribution is high. Moving from the generic distribution for each workload to the specialized distribution for each workload caused the aggregate throughput of the storage-nodes to increase over 96%, from 31 MB/s to 61 MB/s.

Based on Ellard's study of Harvard's NFS systems [7], it is apparent that real-world workloads are mixes. The studied NFS volumes showed random and sequential accesses, varied read/write ratios, and temporary as well as long-lived data. Our results show that such varied workloads could benefit greatly from the per-object versatility that Ursa Minor provides.

4.3.3 Specializing the block size

The data block size is an important factor in performance. Figure 5 shows the effect of block size on performance for two workloads in Ursa Minor. It shows

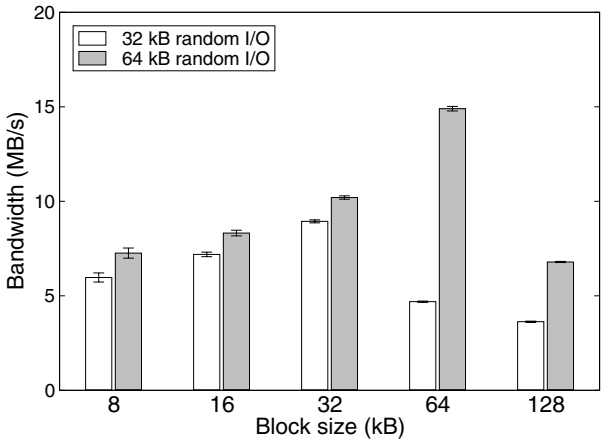

Figure 5: Matching block size to request size. This graph illustrates the importance of matching Ursa Minor's block size to the application's block size. In this experiment, a client performed random I/O with a 1:1 read/write ratio. The client I/O sizes were either 32 kB or 64 kB, aligned on I/O size boundaries. The block size of the object was varied between 8 kB and 128 kB. This experiment used a single client and a single storage-node. These numbers are the average of 10 trials, and the standard deviations are shown as error bars.

the bandwidth with a single client that issued an equal number of read and write requests to a single storage-node. The storage block size was varied between 8 kB and 128 kB, while the client request size remained constant. The first workload used a 32 kB request size, and the other used a 64 kB request size. Performance was best when Ursa Minor used a block size that matched the client's requests. 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 operations.

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.

Table 3 shows the performance of the OLTP workload when tolerating different types of faults. This experiment used 12 clients and 6 storage-nodes. The table illustrates how, in general, making the data more robust (e.g., an asynchronous timing model instead of synchronous or withstanding more failures) impacts a workload's performance. These performance impacts would likely be unacceptable if they affected all data, but the resulting robustness benefits could be necessary for critical data.

Faults (total, byz)	Synchronous	Asynchronous
$1/0$	15.3 MB/s (.13)	15.8 MB/s (.15)
$1/1$	15.3 MB/s (.10)	11.3 MB/s (.15)
$2/2$	6.9 MB/s (.10)	N/A

Table 3: Fault model performance comparison. This table lists the aggregate bandwidth for the OLTP workload, using distributions that can withstand different types and numbers of storage-node failures. It shows the bandwidth as a function of the number and type of faults and the synchrony model. In all cases, replication ($m = 1$) was used. The number of storage-nodes, n, that each object was spread across, ranged from two (crash/synchronous) to five (two Byzantine/synchronous). Performance for the configuration tolerating two Byzantine failures with an asynchronous timing model is not shown since it required more than the available, six, storage-nodes. All numbers shown are the average of 10 trials, with the standard deviations shown in parentheses.

4.4 Ursa Minor: On-line change

This section describes three experiments that demonstrate Ursa Minor's support for on-line data distribution change. To illustrate the effect of re-encoding data to match workload access characteristics and the subsequent benefits, we constructed a synthetic workload in which a single client accessed a 2 GB object randomly, using an access block size of 64 kB. In the original encoding, the object resided on a single storage-node and the block size for the data was 128 kB. During the experiment, it was re-encoded to use a 64 kB block size as well as migrated to a different storage-node.

Figure 6 illustrates the effect of re-encoding data as a function of the workload's read:write ratio. Ursa Minor's incremental re-encoding process is contrasted to another way of re-encoding: blocking access to the object until re-encoding completes.

Ursa Minor's method of changing the distribution incrementally (using back-pointers) has minimal impact on the client's requests and completes within a reasonable amount of time. This is true for both the back-pointer installation period and the coordinator copy period. Additionally, for a write-mostly workload, the role of the coordinator is less important because the workload's writes assist the re-encoding process (back-pointers are overwritten with data as clients perform writes). A write-mostly workload also benefits quickly from the re-encoding process, because all writes are done with the new, efficient encoding.

Figure 7 illustrates the process of re-encoding for the TPC-C benchmark running over Ursa Minor's NFS server. In this setup, the benchmark spawns 10 client threads on a single machine that accessed one warehouse with a footprint of approximately 500 MB. The database is originally encoded to use two-way replication and the block size for the database object was 64 kB. The

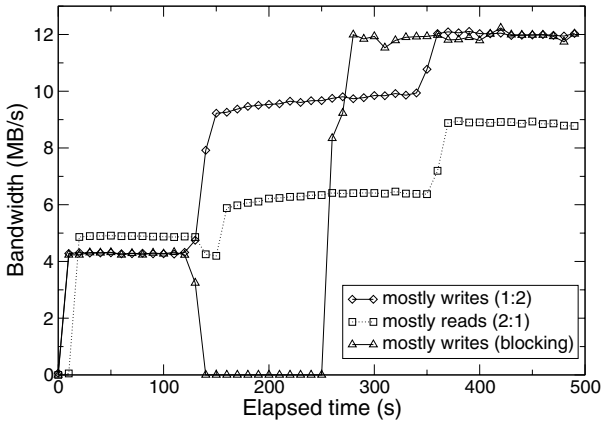

Figure 6: Distribution change. This graph shows the effect of migration and re-encoding as a function of the workload's read:write ratio. Each point is an average of ten trials. The standard deviation for each point was less than 0.5 MB/s. The back-pointer installation began at time 130, and the migration and re-encode began at time 140. The "blocking" case completed quickly but denied access to clients during the distribution change.

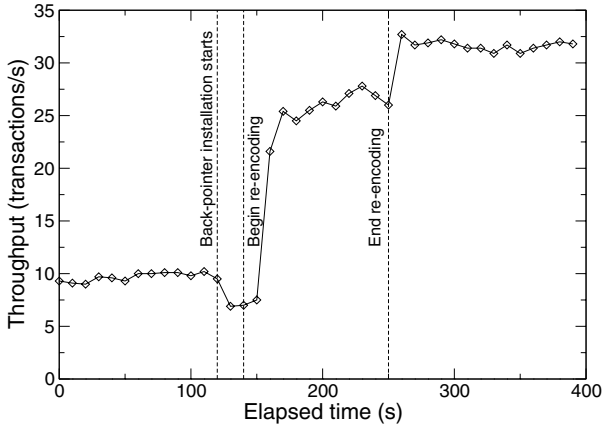

Figure 7: In-place re-encoding of a live database system. This graph shows the positive effect of re-encoding on the throughput that the TPC-C benchmark sees when accessing the underlying database. Ten trials are averaged and the standard deviation is less than 4 tps for each data point. Re-encoding changed the default block size of 64 kB to match the client's request size of 8 kB. The database was replicated on two storage-nodes and the re-encoding happened in-place.

database access size for TPC-C was 8 kB, causing inefficient access, especially when writing to the database. Writing an 8 kB page incurred the cost of first reading a 64 kB block and then performing a 64 kB write.

In Figure 7, the coordinator performed a re-encode in-place (using the same storage-nodes) to match the data block size to the access size. Because the re-encode used the same set of storage-nodes, there was contention between the coordinator and the client, which caused a performance drop during the back-pointer installation phase. The re-encode process took less than three minutes and upon completion, the client achieved approximately three times higher throughput from the storage-nodes.

An additional experiment was conducted that changed the distribution of the TPC-C database from a 1-of-2 (mirroring) encoding to a 4-of-5 encoding scheme. This distribution change completed in under three minutes and impacted the foreground workload by less than 5%. Such a distribution change is valuable when storage space is at a premium, because it reduces the capacity overhead from 100% to just 25%.

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 data item. 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.

Acknowledgements

We thank the members and companies of the PDL Consortium (including APC, 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, by the Army Research Office, under agreement number DAAD19–02–1–0389, and by a National Science Foundation Graduate Research Fellowship. James Hendricks and Matthew Wachs are supported in part by NDSEG Fellowships, which are sponsored by the Department of Defense.

References

[1] A. Adya, et al. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. Symposium on Operating Systems Design and Implementation. USENIX Association, 2002.

[2] T. E. Anderson, et al. Serverless network file systems. *ACM Transactions on Computer Systems*, **14**(1):41–79. ACM, February 1996.

[3] S. Baker and J. H. Hartman. *The Mirage NFS router*. Technical Report TR02–04. Department of Computer Science, The University of Arizona, November 2002.

[4] B. Becker, et al. An asymptotically optimal multiversion b-tree. *VLDB Journal*, **5**(4):264–275, 1996.

[5] M. J. Carey, et al. Shoring up persistent applications. ACM SIGMOD International Conference on Management of Data. Published as *SIGMOD Record*, **23**(2):383–394. ACM Press, 1994.

[6] P. Corbett, et al. Row-diagonal parity for double disk failure correction. Conference on File and Storage Technologies. USENIX Association, 2004.

[7] D. Ellard, et al. Passive NFS tracing of email and research workloads. Conference on File and Storage Technologies. USENIX Association, 2003.

[8] EMC Corp. EMC Centera: content addressed storage system, October 2005. http://www.emc.com/products/systems/centera.jsp?-openfolder=platform.

[9] EqualLogic Inc. PeerStorage Overview, October 2005. http://www.equallogic.com/pages/products_technology.htm.

[10] S. Frølund, et al. FAB: enterprise storage systems on a shoestring. Hot Topics in Operating Systems. USENIX Association, 2003.

[11] G. R. Ganger, et al. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU–CS–03–178. Carnegie Mellon University, August 2003.

[12] S. Ghemawat, et al. The Google file system. ACM Symposium on Operating System Principles. ACM, 2003.

[13] G. A. Gibson, et al. A cost-effective, high-bandwidth storage architecture. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[14] L. Gong. Securely replicating authentication services. International Conference on Distributed Computing Systems. IEEE Computer Society Press, 1989.

[15] G. R. Goodson, et al. Efficient Byzantine-tolerant erasure-coded storage. International Conference on Dependable Systems and Networks, 2004.

[16] G. R. Goodson, et al. *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.

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

[18] IBM Almaden Research Center. Collective Intelligent Bricks, October, 2005. http://www.almaden.ibm.com/StorageSystems/-autonomic_storage/CIB/index.shtml.

[19] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[20] S. Kleiman. Personal communication, October 2002. Network Appliance, Inc.

[21] A. J. Klosterman and G. R. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[22] L. Lamport, et al. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.

[23] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[24] Lustre, October 2005. http://www.lustre.org/.

[25] M. A. Olson, et al. Berkeley DB. Summer USENIX Technical Conference. USENIX Association, 1999.

[26] Panasas, Inc. Panasas ActiveScale Storage Cluster, October 2005. http://www.panasas.com/products_overview.html.

[27] D. A. Patterson, et al. A case for redundant arrays of inexpensive disks (RAID). ACM SIGMOD International Conference on Management of Data, 1988.

[28] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. Conference on File and Storage Technologies. USENIX Association, 2002.

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

[30] S. Rhea, et al. Pond: the OceanStore prototype. Conference on File and Storage Technologies. USENIX Association, 2003.

[31] E. Riedel and J. Satran. OSD Technical Work Group, October 2005. http://www.snia.org/tech_activities/workgroups/osd/.

[32] R. L. Rivest. *The MD5 message-digest algorithm*, RFC–1321. Network Working Group, IETF, April 1992.

[33] M. Rosenblum and J. 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.

[34] Y. Saito, et al. FAB: building distributed enterprise disk arrays from commodity components. Architectural Support for Programming Languages and Operating Systems. ACM, 2004.

[35] M. Shapiro. Structure and encapsulation in distributed systems: the proxy principle. International Conference on Distributed Computing Systems. IEEE, 1986.

[36] C. A. N. Soules, et al. Metadata efficiency in versioning file systems. Conference on File and Storage Technologies. USENIX Association, 2003.

[37] J. D. Strunk, et al. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation. USENIX Association, 2000.

[38] P. Thambidurai and Y. Park. Interactive consistency with multiple failure modes. Symposium on Reliable Distributed Systems. IEEE, 1988.

[39] Transaction Processing Performance Council. TPC Benchmark C, December 2002. http://www.tpc.org/tpcc/ Revision 5.1.0.

[40] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. International Workshop on Peer-to-Peer Systems. Springer-Verlag, 2002.

[41] J. Wilkes, et al. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

[42] J. J. Wylie. *A read/write protocol family for versatile storage infrastructures*. PhD thesis. Technical report CMU–PDL–05–108, Parallel Data Laboratory, Carnegie Mellon University, October 2005.

[43] J. J. Wylie, et al. Survivable information storage systems. *IEEE Computer*, **33**(8):61–68. IEEE, August 2000.

[44] K. G. Yocum, et al. Anypoint: extensible transport switching on the edge. USENIX Symposium on Internet Technologies and Systems. USENIX Association, 2003.

[45] Z. Zhang, et al. RepStore: a self-managing and self-tuning storage backend with smart bricks. International Conference on Autonomic Computing. IEEE, 2004.