A Self-Organizing Storage Cluster for Parallel Data-Intensive Applications*

Hong Tang§, Aziz Gulbeden†, Jingyu Zhou†, William Strathearn†, Tao Yang§†, and Lingkun Chu§
{htang, gulbeden, jzhou, bill, tyang, lkchu}@cs.ucsb.edu
† *University of California at Santa Barbara*
§ *Ask Jeeves*

ABSTRACT

Cluster-based storage systems are popular for data-intensive applications and it is desirable yet challenging to provide incremental expansion and high availability while achieving scalability and strong consistency. This paper presents the design and implementation of a self-organizing storage cluster called *Sorrento*, which targets data-intensive workload with highly parallel requests and low write-sharing patterns. Sorrento automatically adapts to storage node joins and departures, and the system can be configured and maintained incrementally without interrupting its normal operation. Data location information is distributed across storage nodes using consistent hashing and the location protocol differentiates small and large data objects for access efficiency. It adopts versioning to achieve single-file serializability and replication consistency. In this paper, we present experimental results to demonstrate features and performance of Sorrento using microbenchmarks, application benchmarks, and application trace replay.

1. INTRODUCTION

Large-scale cluster-based storage systems [3, 16, 33, 21] are widely used in high performance data-intensive applications [2, 6, 17, 42, 51]. When being deployed in a production environment with continuous workload, a storage cluster should not only deliver high I/O performance, but also be available in a 24×7 manner. Additionally, certain applications desire a consistency model stronger than common NFS session semantics.

Achieving high performance, incremental expansion, and high availability with strong consistency for general applications is challenging and typically requires compromise in at least one area [19]. Previous research on cluster-based storage systems relaxes either the constraint on system availability during failure (such as PVFS [16]), or the feature of incremental expandability (such as Petal [33] and GPFS [41]). Other systems exploit application-storage co-design by weakening the consistency requirement of application data [39, 53, 42]. In this research, our compromise is to target a specific set of parallel applications with low write-sharing patterns. We then take advantage of these application characteristics to provide a storage clustering solution that offers single-file serializability, incremental expandability, and high availability.

Our proposed solution self-organizes available storage resources in the following two aspects: (1) *Adaptivity to node addition and departure*. The system automatically detects node joins and departures and quickly adapts to these changes. (2) *Replication healing*. The system masks component failures through replication and conducts replica repairing to restore replication degree. We want to emphasize that the adaptation and failure recovery is carried out without interrupting the normal operation of the storage service, and do not require human intervention. The latter property is very important because human errors have been identified as a significant source of unmasked system failures [36]. The key design points of our system, called *Sorrento*, are summarized as follows: (1) Sorrento aggregates storage resources as a single object-based storage device [9]. (2) Logical files are divided into segments and can be placed on any storage node. The data location table is partitioned among storage nodes through consistent hashing [30]. Our scheme can be considered a variation of the Chord [44] protocol with improvements specific for a local area network. (3) Sorrento adopts version semantics to keep data consistent and manage concurrent accesses. The overhead of versioning is relatively small for targeted applications with a large amount of I/O parallelism.

The contribution of this work is to develop a storage clustering solution to achieve the aforementioned goals by exploiting the characteristics of targeted applications. The rest of the paper is organized as follows: Section 2 outlines our design assumptions and provides an overview of the system and Section 3 describes the details. Section 4 presents the implementation and evaluation results using microbenchmarks, trace-reply and applications. Section 5 describes related work. Section 6 summarizes the conclusions.

2. ASSUMPTIONS AND OVERVIEW

While our system works for general applications, our targeted applications have the following characteristics: (1) A large number of parallel processes issuing concurrent data-intensive I/O requests. (2) Distributed processes that typically work on disjoint data sets. Write conflicts occur infrequently and if any sharing occurs, it is primarily in the form of read-sharing. Our design exploits the above characteristics and seeks to reduce the overhead of data updates and replica management for single-file serializability, while exploiting I/O concurrency among application processes as much as possible. We have seen many such applications at Ask Jeeves for document processing, data mining, and information searching, and also at Google, Yahoo, MSN, and several other Internet service compa-

*0-7695-2153-3/04 $20.00 (c)2004 IEEE.

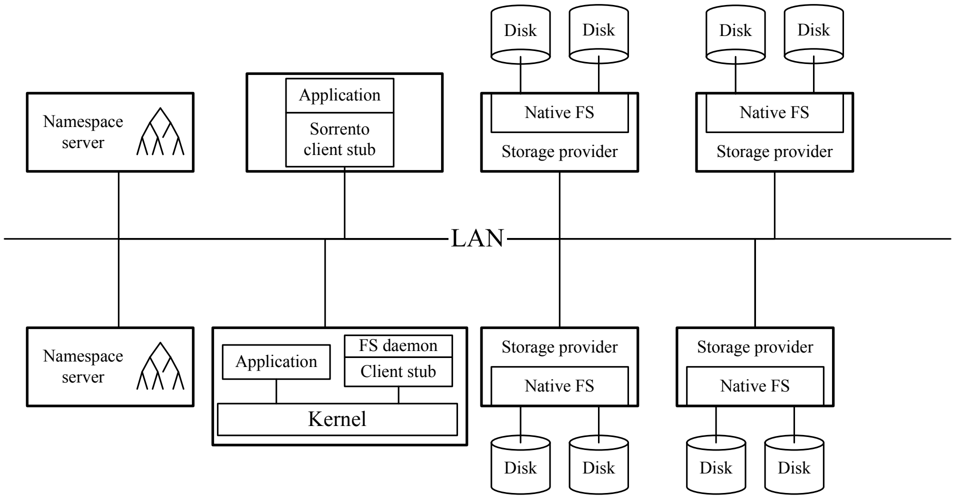


Figure 1: Sorrento system architecture.

nies that use hundreds of machines for fast processing of large-scale data. For example, we studied the twelve distributed applications that crawl and index web documents at Ask Jeeves. Among them, four only read; seven partition the data with no write sharing; and only one incurs infrequent write sharing. Other applications exhibiting these characteristics include: collaborative caching, movie scene rendering, satellite or biomedical image processing, and online e-commerce services.

Our targeted applications often demand a consistency model which allows multiple read and write operations on a single file to be grouped as an atomic transaction, and the outcome of a set of concurrent transactions must be equivalent to some sequential execution of these transactions on the same file. The updates of any uncommitted transaction (most likely due to process failure) can be rolled back. We refer to this type of consistency model as *single file serializability*. Examples of operations that need single file serializability include the atomic append operation proposed in GoogleFS [21], and the updates of files with embedded checksums. We discuss our design choice for data consistency in Section 3.5, which works for general applications and can deliver high I/O performance for applications with low write-sharing patterns.

Figure 1 shows the general system architecture of Sorrento. The basic building blocks of Sorrento are two types of cluster nodes: *storage providers* and *namespace servers*. Storage providers are responsible for managing locally attached disks through the native file system interface. They also collaborate on virtualizing the distributed storage into expandable *volumes* to users. Data stored in each volume are organized in a hierarchical directory tree, which is maintained by namespace servers. Namespace management will be discussed in Section 3.1.

Applications can access data stored in Sorrento from any cluster node, either through Sorrento's client stub library or through a kernel interface. Note that storage providers or namespace servers are software entities (daemons) and do not have to run on dedicated machines. In a typical deployment, they may co-locate with Sorrento client applications.

A Sorrento deployment can be configured and maintained incrementally without interrupting the normal operation of the whole system. To add storage resources, we simply attach more disks to a storage provider, and configure this provider to join a designated volume. To repair a failed node or to recycle rack space, we can directly take the machine offline. When a machine is repaired, it can be directly connected to the network without the need to reformat the partitions, and the system automatically determines the freshness of the locally stored data.

The core Sorrento API layer exports an NFS-style interface, in which operations are based on opaque file and directory handles [7]. Upon this layer, we have implemented a library interface similar to the UNIX file-system calls, and a kernel module based on FUSE [4] allowing applications to access a Sorrento volume transparently without recompilation once this volume is mounted to a local file system.

3. SYSTEM DESIGN

We break the core Sorrento system into seven interconnected components and this section describes these components in details. Their dependences are illustrated in Figure 2 and an arc $A \rightarrow B$ represents that B relies on the functionality of A.

3.1 Namespace Management

Sorrento separates namespace and storage management. Such a separation is not a new concept and has been adopted in other work such as Zebra [25], NASD [22], PVFS [16] and GoogleFS [21]. The underlying motivation is that namespace operations are very different from application I/O operations. Namespace operations typically involve small read/write requests and may require atomicity for requests that access multiple data objects. While, for application I/O, it is more important to serve large I/O requests as quickly as possible. Typically operations on different files are independent. Separating these two types of workload allows making optimizations targeted to each specific type.

The namespace servers are only responsible for operations such as creating/removing a directory, creating/removing a file entry[1] under a directory, directory listing, and pathname lookup. Each file entry contains a 128-bit file identification number and we call it FileID

[1] A file entry on the namespace server can be considered as the inode equivalent in Sorrento.

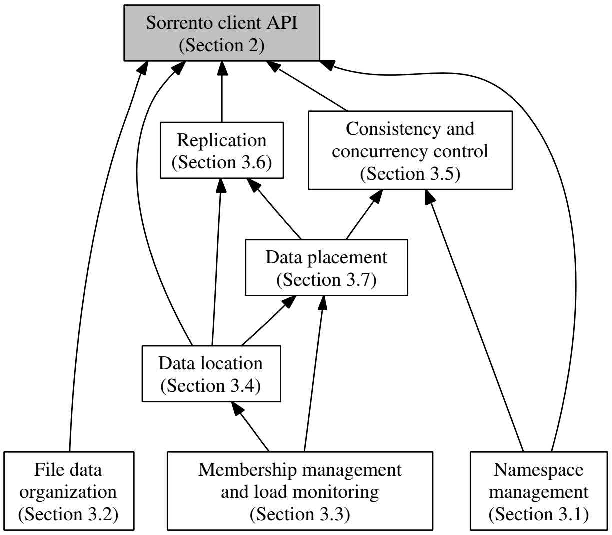


Figure 2: Sorrento components and their interdependencies.

in the rest of this paper. Each file entry also contains the file's latest version, time stamp, and other control information.

Similar to the metadata servers in Lustre [3], but different from most other research such as PVFS [16] or GoogleFS [21], the namespace servers do not keep the physical locations of data segments. FileIDs are persistent across the lifespan of files, and are location independent. Such a design is desired to allow data blocks to migrate among providers to balance storage usage or I/O workload, or to exploit data locality. Requiring the namespace servers to track the locations of mobile data blocks would make them vulnerable to become a performance bottleneck.

The design of the namespace server is independent of other system components, so we only present the sketches of our design here to avoid overwhelming readers with details: The directory tree is stored in a Berkeley DB [35] database. To tolerate process failures, we employ a master-slave replication scheme. Namely, we pair each namespace server (the *master*) with a hot-standby (the *slave*). All client requests are served by the master. The master also passes the database updates to the slave. The slave takes over as the master server when the master fails. Both the master and slave employ a combination of write-ahead logging and checkpointing to ensure the database integrity.

3.2 File Data Organization

Sorrento adopts the conventional representation of a file as a linear array of bytes. The actual file is divided into variable-length *data segments* and stored on different providers. Each data segment is kept on one storage provider in its entirety. The exact organization of data segments is specified in an *index segment*. All data segments and index segments are addressed through 128-bit SegIDs, which can be generated locally with little chance of collision by combining a machine's MAC address, its internal high-resolution timer, and random seeds. In our implementation, a FileID is the same as the SegID of the index segment.

Sorrento currently has three data organization schemes: linear, striped, and hybrid. The hybrid scheme is illustrated in Figure 3. The data segments are sequentially grouped into several stripe groups; while within each group, data are striped across multiple segments.

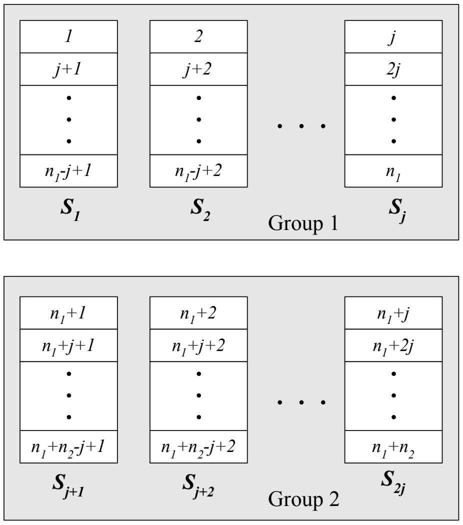


Figure 3: Illustration of the Hybrid data organization scheme in Sorrento. $S_1, S_2, ...$ are segments. Each segment is further divided into fixed-size blocks (32KB).

Sorrento automatically determines the segment sizes for different files so that small files are stored in small segments, while larger files are mostly stored in large segments. For small files, to avoid the inefficiency of two data transfers (first reading the index segment, then accessing the data segment), we *attach* the file data within the index segment [20].

3.3 Membership and Load Monitoring

Previous systems, such as xFS [12] or PVFS [16], keep a table of I/O servers as hard states, which are either updated synchronously among all servers or maintained at a central location. For a large-scale cluster, cluster nodes may join or leave the network frequently, updating the membership information synchronously is costly and may lead to scalability problems. On the other hand, the membership information is needed for most of the other components in Sorrento (see Figure 2), relying on a central server to supply the membership information would easily saturate that server and thus limit the system's scalability.

In Sorrento, the membership manager (which runs on all cluster nodes) maintains the set of live storage providers as soft states in a way similar to the one used in Neptune [42] and all storage providers periodically multicast heartbeat packets. In our current implementation, all storage nodes subscribe to the same multicast group. Our measurement showed that multicast consumes less than 1% of the CPU time (measurement done on dual 1.4GHz P3) and around 1MB/s bandwidth for a 1000-node cluster. For clusters whose scales go beyond that, we plan to adopt a hierarchical approach.

3.4 Data Location

One of the major design challenges of Sorrento is the data location scheme because a segment may be placed on any provider (or *providers* in the case of replication) in Sorrento. Given a SegID, the location scheme needs to quickly locate which provider (providers) stores the segment [46].

3.4.1 Partitioning using Consistent Hashing

Sorrento's data location scheme has been influenced by data location protocols proposed in peer-to-peer networking research, and the base scheme is similar to Chord [44] in various ways. The data location information is partitioned among all the storage nodes. For each SegID, we designate a *home* host that is responsible for tracking the hosts that store the segment, which are called the *owners* of the segment. Currently, we use *consistent hashing* [30] to determine the home host of a SegID. Unlike Chord, where each host maintains a finger table and performs lookup in $\log N$ (N is the number of providers) steps, a Sorrento client has the complete view of all the storage providers (Section 3.3) and can directly determine the home host of a certain SegID without involving other servers or clients in the process.

Each storage provider maintains a location table that maps the SegIDs it is responsible for (as the home to these SegIDs) to segment owners. The location tables are also managed as soft states, which is reconstructed every time a storage provider starts up and is refreshed periodically during its life span. There are four types of events that trigger the update of the location table:

(1) **Periodic content refreshing.** Each owner periodically updates the location tables on other providers by sending the SegIDs of locally stored segments to their corresponding home hosts. In our test, these tables are refreshed every 15 minutes. The complexity of calculating the list of SegIDs for a remote home host is proportional to the size of the list, which is asymptotically optimal. The network overhead of periodic refreshing is typically negligible. For instance, for a very large storage cluster with one billion segments and one thousand storage nodes, the update traffic would consume 18KB/s incoming and 18KB/s outgoing bandwidth on the link between a node and the network switch.

An entry in a provider's location table could become garbage when the provider is no longer the home host of a SegID. This may happen when a newly joined provider takes over the provider as the new home of that SegID. The garbage entries can be detected based on the last refreshing time because garbage entries will never be refreshed.

(2) **Node-join notification.** This event happens when the membership manager adds a new provider (provider A) to the set of live providers. Note that this event could imply two possible situations: (i) an existing provider (B) learns about a newly joined provider (A), or (ii) a newly joined provider (B) discovers an existing provider (A). In response to this event, provider B schedules a refreshing event for provider A. To avoid a newly joined provider from being overwhelmed by simultaneous refreshing requests from existing providers, the refreshing event is scheduled after a random delay (within 20 seconds in our test environment).

(3) **Node-departure notification.** This event happens when the membership manager removes a provider (provider A) from its membership set. In response to this event, a data segment owner (provider B) will cancel the pending refreshing events for home host A, and remove from the location table the SegIDs that are stored on provider A. Finally, provider B will compose a list of locally stored segments whose original home was A, and send those SegIDs to their new home hosts.

(4) **Segment creation and deletion.** Between periodic refreshing, a provider will also update other providers' location table in response to the creation or deletion of a local segment. Specifically, the provider will instruct the home host of the segment to add or remove an entry in its location table. Note that this event allows for fast updating and happens between periodic content refreshing described above.

The location table is kept in memory for access efficiency. The space overhead of the location table is around 20MB for a 1000-node cluster with one billion segments. It is insignificant considering that desktops nowadays typically are equipped with one GB memory.

3.4.2 Optimizing Small File Accesses

One problem of the aforementioned data location scheme is that each data access requires at least two network operations: first contacting the home host, and then interacting with the owner. For large segments, such an overhead will be amortized by the data transfer cost; however, it is very inefficient for small segments, whose cost is dominated by network latencies. (A particular case is accessing index segments.) We solve this problem by collaborating with the data placement design so that small segments will be placed directly on home hosts (unless the home hosts run out of space). In this way, a client will always request a segment from the segment's home host. If the home host has a copy of the segment, it will return data directly; otherwise, it will reply with a *redirection* response, and instruct the client to contact a different provider. The actual data location algorithm is integrated as part of the data access routines. Figure 4 illustrates the algorithm for the read operation.

As we can see in Figure 4, a read request starts with a location cache lookup (step 1), which allows read requests for the same segment to be served directly without invoking the whole data location protocol. If the location of the desired segment is not found in the cache, we contact the home host of the segment. The home host will either return the data immediately or return a redirection response (step 2). Typically, it is sufficient to retrieve the data after one indirection. However, in rare occasions when segments are migrating or when the set of storage providers is changing, we may need to follow redirections for multiple times. We limit the maximum number of redirections to avoid being trapped in transient redirection loops.

We may fail to locate the desired segment in the first two steps either because we contacted the wrong home host due to an inconsistent view of the live providers or because the location information has not been propagated to the home host yet. In those cases, we will fall back to a backup scheme which simply queries all the providers by issuing the request through a multicast channel. The probability of resorting to the multicast-based query depends on the actual application workload and how often nodes join or leave the system. Our experimental study shows that the backup scheme has been used very infrequently, less than 0.001%.

3.5 Consistency and Concurrency Control

Conventionally, there are two ways to achieve single file serializability: we can either *pessimistically* use locks to avoid conflicting operations before a transaction starts; or *optimistically* let a transaction proceed normally and rollback its effect if conflicts are detected during commitment. Following the characteristics of our targeted applications and workload, we find that it is more efficient and simpler to use a version-based consistency model. Conflicting transactions will result in conflicting version advances, and rolling back a transaction simply means discarding an uncommitted ver-

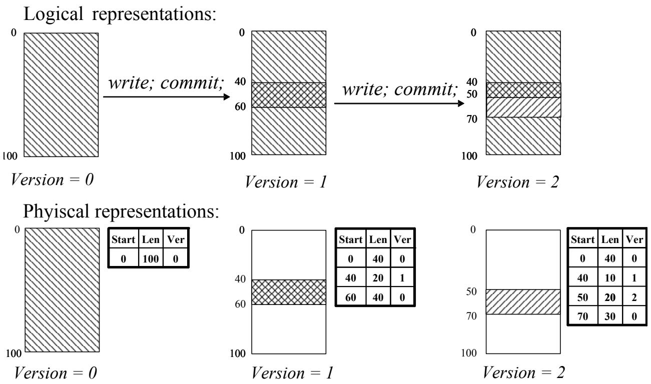
```
int read_segment(Segment b, ...)
{
  // Step 1: check the local location cache.
  Provider h = location_cache.lookup(b);

  if (h.valid) { // cache hit
    RetVal rval=remote_read(h, b, ...);
    if (rval.status==OKAY)
      return rval.length;
  }
  // Step 2: cache miss,
  // Contact home host and follow redirections.
  int max_redir=10; // up to ten redirections.
  h = CH_hash(providers, b.sid); // find home.

  while (max_redir--) {
    RetVal rval=remote_read(h, b, ...);
    switch (rval.status) {
    case OKAY:  location_cache.insert(b, h);
                return rval.length;
    case REDIR: h = rval.redir_host;
                break; // redirection
    default:    max_redir = 0;
                break; // failed.
    }
  }
  // Step 3: fall back to multicast lookup.
  h = mcast_lookup(b.sid);
  if (h.valid) {
    RetVal rval=remote_read(h, b, ...);
    if (rval.status == OKAY) {
      location_cache.insert(b, h);
      return rval.length;
    }
  }
  return -1; // read failure.
}
```

Figure 4: Illustration of the data location protocol with the `read_segment` operation. `read_segment` reads a portion of a segment starting from a certain offset. The parameters of data buffer, starting offset and read length are not shown for `read_segment` and `remote_read` functions.

sion. This section describes the semantics and implementation of such a model in Sorrento, and provides justifications of such a design choice in our context.

Semantics. From a user's point of view, a file evolves over a series of versions. Modifications to a file can only be applied to the latest version. To make changes to a file, an application first creates a shadow copy of the latest version, which is only visible to that application. Once the application makes some modifications that transform the shadow copy to a new consistent state, it can commit the shadow copy and make it the latest version of the file. A committed version is immutable and further modifications to the file will advance the version again. Versioning is implicitly tied with conventional file system primitives: a commit operation is invoked when we make a `close()` or `sync()` call. A shadow copy is created when we `open()` a file for write or after we finish a `sync()` call. The latest version of a file is maintained on the namespace server.

Implementation. Behind the scene, all index segments and data segments are also versioned. The versions of data segments for a specific file version are stored in the corresponding index segment. In fact, a file's version is just the version of the index segment. If part of a file is changed, only the modified segments and the index segment will have their version numbers advanced. To avoid the overhead of making complete copy of shadow segments, we employ an optimization based on the idea of copy-on-write. To create a shadow copy of a segment (the base segment), we simply create a blank segment and truncate it to the same size as the base segment. Unmodified regions of the shadow copy will be found in the base segment or its ancestor versions, and modified regions will always be found in the newly created segment. We use an index table to maintain the mapping from region ranges to physical segments where the valid data for the shadow copy can be located. The index tables are kept in memory, and will be flushed to disk when the shadow copy is committed and becomes immutable. Figure 5 shows an example of how a shadow segment is created and modified. In this example, a segment with 100 sectors is updated twice. First the region from sector 40 to 60 is updated, and then the region from sector 50 to 70 is updated.



Start	Len	Ver
0	100	0

Start	Len	Ver
0	40	0
40	20	1
60	40	0

Start	Len	Ver
0	40	0
40	10	1
50	20	2
70	30	0

Figure 5: An illustration of the copy-on-write process. Initially, the logical data layout and physical layout of version 1 is the same. When a shadow copy is created, it consists of a blank segment. When the shadow segment is updated, we update the corresponding index table. The index table allows us to directly determine which physical version of the segment we should resort to for a read request.

To cope with failures of client applications, which may leave uncommitted shadow segments as garbage, all shadow copies are given an expiration time. Applications must either commit a shadow segment before its expiration, or renew its expiration time.

When two processes attempt to modify the same file, each will work on its own shadow copy of the file. However, a conflict will occur when both of them attempt to commit their changes. Update conflicts can be avoided among cooperative processes by write-leases through the namespace servers. Otherwise, they will always be detected during the commit phase: when a process attempts to commit a new version of a file to the namespace server, it will also specify the base version of the file. If the version stored on the namespace server is higher than the base version, then it means that another process has made some changes to the same file and successfully committed the changes. The former process may attempt to resolve the conflict by reapplying the changes to the new version and recommit (as OceanStore's predicate-based update primitives [32] or Bayou's merge procedures [47]), or it may just notify end-users about the conflict.

Committing a new version of a file may require the commitment of multiple segments on distributed providers. We use the standard two-phase commitment (2PC) [49] to ensure the atomicity , whose details are left out for simplicity.

Over time, a segment may have many versions, which could lead to space waste and data fragmentation. Sorrento consolidates versions after they have not been accessed for a certain amount of time. Version consolidation frequency can also be controlled by system-wide parameters. By default, we start consolidation before the version chain length grows beyond three. In the future, we plan to allow users to specify or the system to automatically detect milestone versions that will never be consolidated, a feature similar to the Elephant file system [40].

Justification The choice of a version-based data consistency model is motivated by the following three reasons: (1) Since in our targeted workload, update conflicts seldom occur, an optimistic concurrency control scheme such as versioning is preferred comparing to distributed locking. (2) Storage overhead is not an issue by using the optimizations of copy-on-write and version consolidation[2]. (3) Finally, versioning greatly simplifies the management of replica consistency (Section 3.6).

We also want to emphasize that in our design, update conflict resolution mechanisms are not included in the core system. Instead, applications can choose to implement their own conflict resolution protocol based on the application semantics. For instance, we have written an atomic append operation [21], which is illustrated in Figure 6.

```
// append a record r to file f.
void atomic_append(string &f, Record &r)
{
   while (1) {
      // Open a shadow copy of f.
      FileHandle *fh = open(f, "w");
      fh->append(r); // Append r to f.
      // Try to commit the file.
      if (fh->commit() == true) {
         // Succeed: return.
         return;
      }
      else {
         // Conflict: delete the shadow copy and retry.
         fh->drop();
      }
   }
}
```

Figure 6: Implementation of *atomic append*.

For applications (such as DBMS) that prefer to manage their own data consistency and concurrency control, Sorrento also provides an option to disable data versioning completely. Since concurrency control and replica consistency management are tightly coupled, disabling versioning also disables replication.

3.6 Fault Tolerance and Data Replication

Sorrento tolerates two types of component failures through replication: (1) **Node failure.** This implies the total lose of a cluster node, detected by other nodes based on heartbeat packets; (2) **Disk failure.** This implies a locally attached disk fails while the node is still alive. Disk failures can be detected through periodic probing performed by storage providers. Upon the detection of a node failure, other nodes (acting as home hosts) will assume that all segments stored in the failed node are lost and update their location table accordingly. Upon the detection of a disk failure, the storage provider will react as if the segments stored on the failed disk are deleted and notify the home hosts correspondingly.

The replication degree can be customized on the file basis to fit the reliability requirement of different applications. Sorrento's version-based data consistency also simplifies the management of replica consistency. We can identify whether two replicas of a segment are the same by comparing their versions. This allows Sorrento to adopt the idea of asynchronous replication [23] (and maintain single file serializability): Updates to a segment will first be applied to the shadow copy of one replica. When the shadow copy is successfully committed, the provider will notify the home host of the version upgrade. This event would cause the home host to discover that there is a version discrepancy among different replicas, so the home host will notify those with older versions to synchronize with the latest version owner. Note that we still guarantee that an `open()` call always finds the latest committed copy of a file even when asynchronous replication is adopted by keeping the latest version information at the namespace server.

Asynchronous replication is enabled by default, and it allows writes to be executed as fast as if there were no replication. Updates are propagated in background (after a segment is committed) and is not in an application's execution path. However, for applications that do expect updates to be synchronously applied to all replicas, they can override the default policy when committing the file. In a synchronous replication, the application will be blocked until the host determines that all replicas receive the update and notifies the application. Typically, an application may allow minor changes to be propagated lazily, and use synchronous commitment to commit a milestone version.

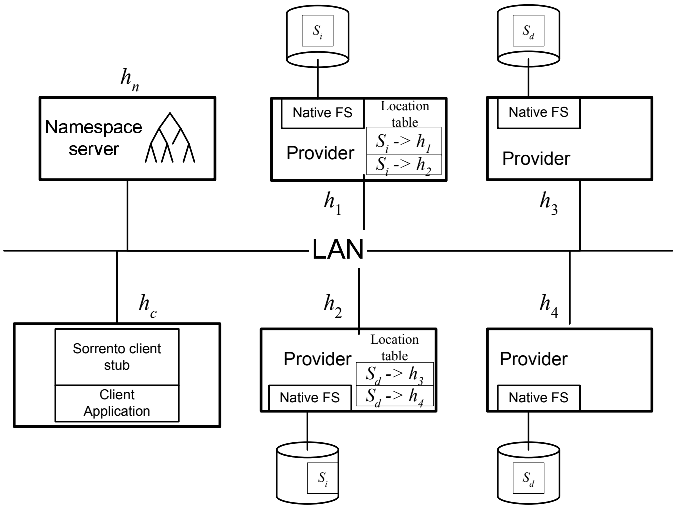


Figure 7: The initial setting of an example that demonstrates the interaction of Sorrento entities during a file session. File `/foo` is represented by an index segment S_i and a data segment S_d. Both segments are replicated twice. The home hosts of S_i and S_d are h_1 and h_2 respectively. h_n is the namespace server, and h_c is the node where client application is launched.

Replication degree is also maintained by home hosts: A home host maintains a list of locations for each segment it is responsible for. The length of the list represents the replication degree of the segment. When a home host finds an under-replicated segment, it will choose new replica sites (different from existing owners) and notify them to retrieve a copy from existing owners. It is also possible

[2] In fact, one cannot consider that a lock-based scheme has no space overhead because a lock-based scheme still requires detailed logging to support failure rollback.

that a segment has more replicas than specified, for instance, when a failed node rejoins the system carrying data stored on it before its failure. In that case, the home host will choose one replica site and inform it to remove the excess replica.

Putting It All Together. We demonstrate how different entities in Sorrento work together through an artificial example. This example involves four storage providers (h_1, h_2, h_3, h_4), and one file `/foo`. File `/foo` is physically represented by an index segment S_i and a data segment S_d. Both segments are replicated twice: S_i is replicated on h_1 and h_2, and S_d on h_3 and h_4. The home hosts for S_i and S_d are h_1 and h_2 respectively. A client application, which runs on host h_c, opens the file `/foo`, performs a write operation, and closes the file.

The initial arrangement of this example is shown in Figure 7, where h_n is the namespace server. The timeline of the execution is illustrated in Figure 8. The activities carried out in each step are explained in Figure 9.

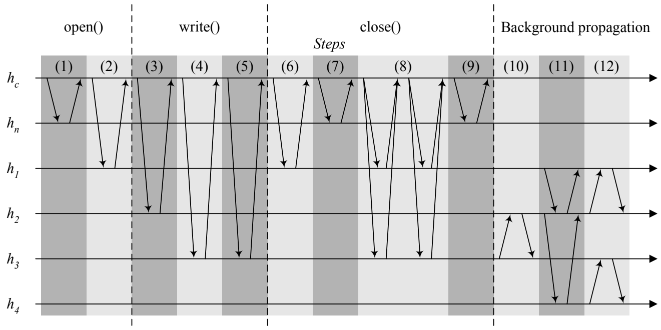


Figure 8: The timeline of a file session, in which an application opens a file, writes to it, and closes it. Time flows from left to right. Arcs between different timelines stand for communications. We also illustrate the steps corresponding to `open()`, `write()`, `close()`, and the background propagation process.

Although the whole process seems lengthy, the actual I/O transfer goes directly between the client and the storage providers (steps (4)-(6)) with little overhead. Typically, an application will issue many read write requests during a file session, and the cost of the `open` and `close` operations will be amortized. Additionally, update propagation are performed in the background and will not slow down the client application.

3.7 Data Placement and Migration

Our aforementioned data location scheme offers the flexibility of placing a segment on any provider without restriction. In Sorrento, a segment is not confined to the location where it is initially created, and the system dynamically adjusts its location to balance I/O load and storage usage, or to exploit data access locality by placing data locally to processes that access them. Due to space constraint, we only briefly describe the data placement and migration policies below. Interested readers are referred to the extended version of this paper for more details [45].

The default placement policy is to evenly distribute actively accessed segments (*hot* segments) across storage nodes to balance I/O workload, and to use infrequently accessed segments (*cold* segments) as *fillers* to balance storage usage. The initial placement of a segment is mainly based on I/O workload, assuming a newly created segment will be accessed soon. Sorrento then automatically classifies whether a segment is hot or cold based on its recent access history. Migration decisions are triggered when there is a significant I/O load or space usage imbalance. The migration destinations are chosen by individual storage providers without contacting other providers. To avoid a newly added provider, which is idle with high free space, being suddenly swamped by other providers, the algorithm selects migration destination probabilistically, and employs randomized back-off. Finally, to prevent the data movement traffic from interfering with normal operation, we control the number of active data transfers per storage provider.

Step	Activity
(1)	The client retrieves file `/foo`'s FileID from the namespace server h_n.
(2)	The client contacts h_1, the home host of the index segment S_i, to retrieve the data of S_i. Since h_1 happens to have S_i, it sends back the data immediately.
(3)	The client contacts h_2, the home host of S_d, to retrieve the data of S_d. h_2 sends back a redirection response to h_d with the two owners of S_d (h_3 and h_4).
(4)	The client contacts one owner h_3 and creates a shadow copy of S_d (called S'_d).
(5)	The client issues the write request to h_3.
(6)	The client closes the file, so it creates a shadow copy of the index segment on h_1 (called S'_i).
(7)	The client contacts the namespace server for the approval of commit. The namespace server finds that the latest version of the file matches the base version specified in the client request, so it approves the commit request.
(8)	The client performs the two-phase commit to make sure the commitment of S'_i and D'_i are carried out atomically.
(9)	The client contacts the namespace server and completes the version commit operation (After (7) and before (9), other processes will be blocked from committing changes to `/foo`.)
(10)	h_1 updates the local location table (not shown), and h_3 updates the location table on h_2 to reflect the version advances of S_i and S_d.
(11)	h_1 and h_2 instruct h_2 and h_4 to sync with h_1 and h_3 for S'_i and S'_d respectively.
(12)	h_2 and h_4 retrieve the updates from h_1 and h_3 respectively.

Figure 9: Activities carried out in the steps of Figure 8.

We also support a special *locality-driven* data placement policy. This is again based on our observation that for many data-intensive applications, the data set is partitioned and different processes access disjoint datasets most of the time, exhibiting good locality. Thus, it is desirable to take advantage of this locality by co-locating segments with the process that accesses them, so that data transfers do not need to go through network. A segment will migrate to a remote provider if the traffic it receives is from that provider exceeds a pre-defined threshold.

4. IMPLEMENTATION AND EVALUATION

4.1 Prototype Implementation

A prototype of Sorrento has been developed, which implements all the components discussed in Section 3. The whole system is written mostly in C/C++ plus a few hundred lines of assembly code.

Although various components in Sorrento are based on ideas from previous research, we choose to implement the whole system completely from scratch due to either source code unavailability or efficiency considerations. The core components, such as client libraries and server daemons, consist of 50K lines. System monitoring, diagnosis and maintenance utilities took another 10K lines. As of this writing, our effort mainly focuses on the system's functionality and reliability. Much room is available for further optimization and feature addition.

The Sorrento file system kernel module is based on FUSE [4], a user-space file system module. We made two changes to FUSE to improve its performance: First, we modified the kernel module to use a bigger buffer size than 4KB size of demanding paging, thus reducing the number of up-calls and corresponding Sorrento calls. Second, writes to a file are first cached in memory, and are issued to storage providers asynchronously. The kernel module also intercepts the `sync()` or `close()` calls and flushes all pending writes upon these calls.

4.2 Evaluation Objectives and Settings

Our experimental study seeks to answer the following four questions: (1) How effective are the self-organizing features of Sorrento in handling storage node joins and departures with replication healing (Section 4.3 and 4.4). (2) What is the data-intensive I/O performance of Sorrento in comparison with NFS and parallel file systems such as PVFS (Section 4.5). (3) What are the effectiveness and overhead of design choices such as versioning, replication and asynchronous propagation (Section 4.6 and 4.7). (4) How does Sorrento handle small file I/O operations (Section 4.8).

The evaluations are conducted on three clusters A, B, and C. The hardware and software configuration of these clusters are summarized in Figure 10. It should be noted that each experiment may not use all the available storage nodes of a cluster. In the remaining sections, we use the notation *Sorrento-(n, r)* to denote a Sorrento deployment with n storage providers and all files are replicated r times. Similarly, we use *PVFS-n* to specify a PVFS deployment with n I/O nodes. By default, Sorrento uses asynchronous propagation for replica updates. Additionally, unless otherwise specified, the benchmark programs or application processes run on a separate set of machines from the storage nodes. To realize the best possible performance of PVFS experiments, we modify the programs to directly use PVFS library functions instead of calling UNIX file system calls.

We use a combination of microbenchmarks, real applications, and application trace replay as our evaluation methodology. For the purpose of trace replay, we have implemented two trace collection utilities: one intercepts file system calls through glibc modification and the other intercepts PVFS calls by changing the PVFS library. The traces being collected all have accurate timing information for the starting and ending time of each I/O request. The overhead of trace collection ranges from 3% to 14% depending on the granularity and frequency of requests.

4.3 Handling Node Failures and Additions

We first evaluate how the system handles node failures and additions using a microbenchmark: We employ 10 nodes from Cluster A as storage providers, and populate the system with 200 512MB files, each has three replicas (totally 300GB data). The workload is driven by five client processes. Two processes repeatedly issue large write requests of 4MB each; three of them issue large read requests of 4MB each. We report the aggregated data transfer rates. We deliberately kill one storage provider at time 30 (seconds); and then launch a new storage provider on a different machine at time 45 to compensate the reduced total storage capacity and system processing power.

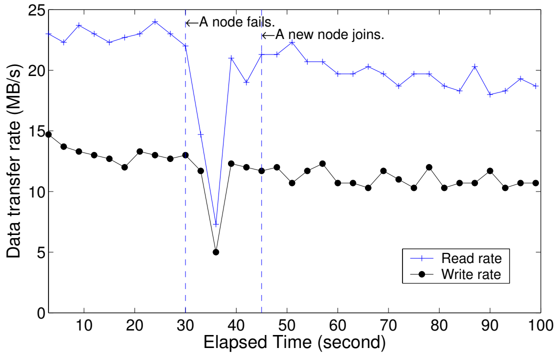


Figure 11: Handling node failures and additions. A storage provider fails at time 30, and a new node joins at time 45.

As shown in Figure 11, transfer rate drops sharply right after the node failure, because the requests issued to the failed node will be blocked until they are timed out. After that, the system quickly adjusts the location table and the transfer rates recover to about 94% of the initial setting. However, even after we add a new node to the system, the transfer rate is only 85% of the level before the failure because at time 60, the system starts to make new copies of under-replicated data, which generates an extra workload. Eventually, all lost replicas are restored after 20 minutes (not shown in Figure 11). The aggregate bandwidth used by the recovery process is 33MB/s. Because a typical MTTR for a single node takes about one day for hard faults, Sorrento's automatic replica regeneration effectively shortens the MTTR by a factor of 72. Given a replication degree of three, this translates to an improvement of overall system availability by five 9's. This experiment confirms that Sorrento is able to handle node failure gracefully, and only exhibit minor performance degradation during recovery. The idea of distributed volume rebuild for improved reliability was also used in [52].

4.4 Replica Healing

We further demonstrate how quickly and effectively the replicas are repaired or healed using a microbenchmark: We employ five nodes from Cluster A as storage providers to form a storage cluster with replication degree 3 and without any initial data. The workload is driven by two clients that continuously issue random writes of 4MB each. The metric used to evaluate the effectiveness of replica healing is the average replication degree of all segments. The replication degree statistics are kept in individual home hosts' logs, and the system-wide average replication degree are then calculated offline. We kill one storage provider at time 200 to observe the effectiveness of the replica healing process.

The results are shown in Figure 12, where x-axis is the elapsed time and y-axis the average replication degree. Since initially the file system is empty, the line starts with an average replication degree of one for newly created segments. After that, the replication degree grows steadily to around 2.1. The average replication degree does not climb to three because new segments are generated continuously during the experiment. At time 200, one storage provider is killed, which causes the replication degree to drop to 1.7 because

	Cluster A	Cluster B	Cluster C
CPU	30 dual P-II 400MHz nodes	30 dual P-III 1.4GHz nodes	8 dual P-III 1.4GHz nodes
Memory	512MB per node	4GB per node	4GB per node
OS	RedHat Linux (kernel 2.4.18)	RedHat Linux (kernel 2.4.25)	RedHat Linux (kernel 2.4.25)
Disks	10 disks, 210 GB	90 disks, 3.6 TB	24 disks, 9.6 TB
Connectivity	Fast Ethernet	Fast Ethernet	Gigabit Ethernet

Figure 10: Hardware and OS settings of three test clusters.

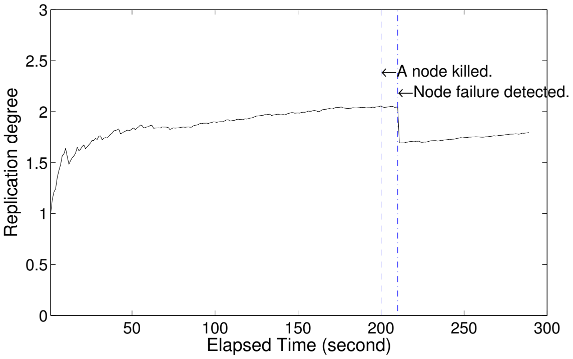


Figure 12: Replication healing after a provider is killed.

around 20% of the replicas suddenly become inaccessible[3]. After this time, the system starts to make new replicas for the under-replicated segments with the remaining four servers. Due to the drop in the number of servers, the load on the remaining servers increases which decreases the rate at which new replicas are created.

4.5 Data-intensive I/O Performance

Microbenchmarks. In this experiment, benchmark `bulkread` repeatedly opens a file, reads 4MB data at random offsets (aligned at 4KB boundary), and closes it. Similarly, benchmark `bulkwrite` repeatedly opens a file, writes 4MB data at random offsets, and closes it. Each file is 512MB large. We run the experiment on cluster B and vary the number of client processes. We compare the aggregated data transfer rate for NFS, *PVFS-8*, *Sorrento-(8,1)*, and *Sorrento-(8,2)*. In this experiment, different client processes access disjoint data sets. For Sorrento and PVFS, a total of 160 files (80GB) are pre-populated. For NFS, a total of 30 files (15GB) are pre-populated. For `bulkwrite`, we also show the performance of Sorrento using eager propagation (*Sorrento-(8,2), eager*). The results are shown in Figure 13.

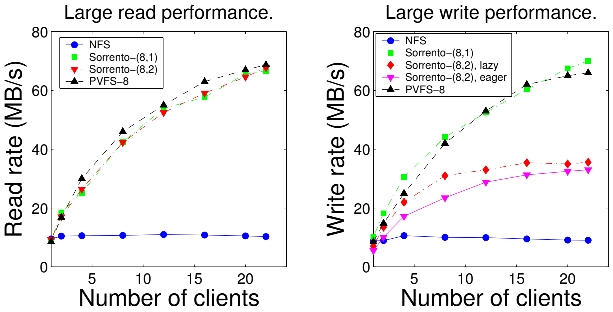


Figure 13: I/O performance comparison.

NFS performs the worst and the read or write rates saturate at 10MB/s, likely due to the saturation of the network connection. Both PVFS and Sorrento are able to scale up the transfer rates with the number of clients. For read rates, both versions of Sorrento and PVFS perform similarly. For write rates, PVFS performs similarly with *Sorrento-(8,1)*, but outperforms *Sorrento-(8,2)* by a factor of two. This is because *Sorrento-(8,2)* needs to commit each write to two replicas. For both systems, the transfer rates are saturated when the network links connecting to the storage nodes are saturated. Finally, for *Sorrento-(8,2)*, the peak transfer rates under synchronous replication (eager) and asynchronous replication (lazy) are close. However, asynchronous replication is able to deliver a higher transfer rate when the number of clients is small and the system is underloaded. This is because under asynchronous replication, a client does not have to wait until all replicas are updated before it can issue the next request.

Parallel Application Trace Replay. Next, we assess the performance of Sorrento through trace-replay of two real applications. The first is the BTIO benchmark from NAS Parallel Benchmark Suite (NPB) [50]. BTIO solves the Block-Tridiagonal problem and issues read/write requests through MPI-IO interface. The second application is a parallel Protein Sequence Matching service based on NCBI's Blast package [8] (PSM). In PSM, a set of backend service processes access a partitioned protein database to serve user submitted search queries.

We conduct our experiments on Cluster A. For BTIO, four trace replayers wrote 2.7GB data and read 1.7GB data. BTIO has five different classes, and we use class B setting. For PSM, eight trace replayers read a total of 3.1GB data (there is no write operation for PSM). BTIO uses PVFS's `list-write` primitive, which is emulated in Sorrento through asynchronous I/O calls. Additionally, in BTIO, distributed processes access a shared file, thus, we disable replication and version-based concurrency control. We plan to investigate using byte-range locking to control concurrency for such applications in the future. In both settings, the trace replayers are launched simultaneously, and they issue requests sequentially with zero thinking time. For both applications, we compare *Sorrento-(8,1)* with *PVFS-8* and NFS. We show the results (Figure 14) in terms of the maximum, minimum and average execution time of client processes, and the aggregated data transfer rates.

		Execution time (sec)			Aggregated transfer rates (MB/s)	
		Min	Max	Average	Read	Write
BTIO	NFS	1426.1	1509.7	1472.8	1.84	1.15
	PVFS-8	140.2	141.5	140.9	19.3	12.0
	Sorrento-(8,1)	156.3	158.1	157.2	17.3	10.7
PSM	NFS	1196.0	1274.7	1235.7	2.51	(N/A)
	PVFS-8	213.8	233.4	226.3	13.7	(N/A)
	Sorrento-(8,1)	200.7	222.5	214.8	14.5	(N/A)

Figure 14: Performance of BTIO and protein benchmarks.

[3] Precisely speaking, the dropping happens at time 210 in Figure 12 because there is a 10-second delay before other nodes discover the node failure.

As we can see, for both applications, the workload is distributed to clients in a balanced way, and thus the execution time for all the client processes is very close for all three types of file systems. Second, NFS again performs the worst while Sorrento and PVFS perform comparably. PVFS holds an 11% edge over Sorrento for BTIO. This is mainly because the prototype version of Sorrento used in this experiment has not been carefully optimized yet. We are currently working on a number of optimizations and some preliminary results indicate that these optimizations can improve Sorrento performance by 10% or more.

Document Repository Performance. We now further validate the performance of Sorrento through a document repository service, which is based on an application-level service from Ask Jeeves [1]. This service maintains a repository of crawled documents from the Internet. Given a URL, it will return the corresponding page source. The data for the service are updated in batch mode: first a new batch of pages are added to the document repository servers during off-peak time, then the servers simultaneously switch to the new dataset. Thus, the I/O workload of this service consists of mostly reads for normal query operations, and periodic intensive writes before data switches. The page data are compressed and distributed in a number of archive partitions based on URL hash. Each compressed page in the document repository is on average 4KB.

The service employs Sorrento as the storage back-end and runs the service front-end on a separate set of nodes. Each front-end node is responsible for a disjoint set of data partitions. In this experiment, five storage providers and five application front-end nodes from Cluster B are employed while 20 machines are used to launch client requests concurrently. There are totally 30 partitions, each of which is around 2.4GB, so each front-end node owns six partitions. The workload is generated by a varying number of clients, each of which sequentially generates read or write requests with zero thinking time, and sends requests to the proper front-end node based on URL hashing.

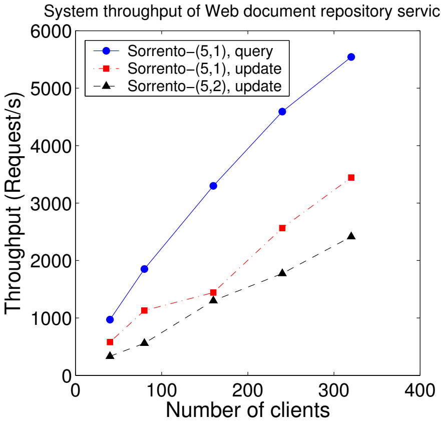


Figure 15: Document repository using Sorrento.

We report the system throughput in terms of the number of query or update requests being served per second. We vary the number of clients from 40 to 320, and the results are shown in Figure 15. As we can see, the service can deliver scalable throughput for both query and update requests. Since the amount of data transfer in serving each client request is relatively small, neither the network or storage bandwidth is saturated.

4.6 Effectiveness of Asynchronous Update

In this section, we evaluate the effectiveness of our asynchronous replication policy using a microbenchmark in which a client process issues 800 random writes of 32KB in 100 file sessions, each session consists of eight writes. We compare among three schemes: (1) **Sync-write.** Each write operation is synchronously applied to all replicas; (2) **Sync-session.** The process is blocked at the session close time until all replicas are updated; (3) **Async-session.** The updates are carried out in the background.

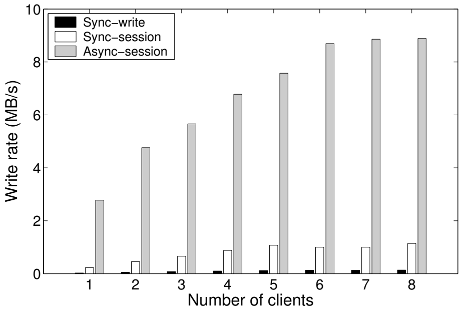


Figure 16: Effectiveness of asynchronous update propagation.

We run the experiment on Cluster A with five storage providers and the replication degree of each file is set to two. We vary the number of client processes from one to eight, and report the aggregated write rate. The results are shown in Figure 16. Async-session is one order of magnitude better than Sync-session, which is in turn much better than than Sync-write. Note that the reported results represent the performance of random small writes with zero thinking time. A write throughput of 8.8MB/s (as realized by the asynchronous scheme) using five servers with a replication degree of two is fairly reasonable. We have conducted offline measurement using the same benchmark on local disks, and a single disk can deliver a 4.3MB/s write throughput. This leads us to an ideal-case upper bound of 10.8MB/s for five providers and a replication degree of two.

4.7 Overhead of Data Replication

We also evaluate the overhead of data replication using a microbenchmark in which a client process repeatedly opens a 512MB file, writes 4MB data with different write sizes, and commits the change. We run the experiment with six storage providers and two clients on Cluster C. We report the aggregate throughput for different replication degrees in Figure 17. Two trends are evident: first, when write block size increases, the overall throughput also increases; second, increasing the replication degree from one (no replication) to two would lead to 10-23% drop of write rate, and further increasing the replication degree from two to three would cause 2-19% drop of write rate. This experiment indicates that asynchronous update with copy-on-write allows us to efficiently implement versioning and replication.

We further study overhead of replication on cluster B for *Sorrento-(12,2)* with 4MB block size. The overhead of replication in terms of write throughput difference between *Sorrento-(12,1)* and *Sorrento-(12,2)* is shown in Figure 18 using benchmark `bulkwrite`. The replication overhead increases as the number of concurrent client writes increases. That is because as more clients involved in parallel writes, these writes trigger more asynchronous updates across machines and create more contention. The overhead is less than 14% when the number of clients is less than 12 and 29% for 16 client machines. This number shows that asynchronous update effectively hides the overhead of replication for a reasonable size of

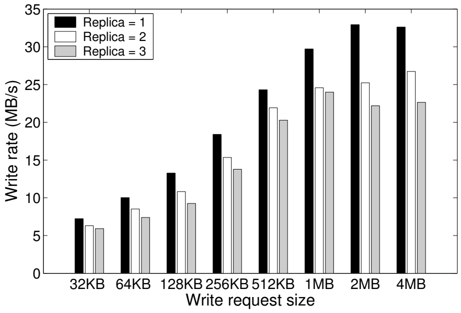


Figure 17: Replication overhead with varying block size.

client machines. If overall I/O bandwidth of the storage servers is saturated, the overhead ratio could be 50% with replication degree two.

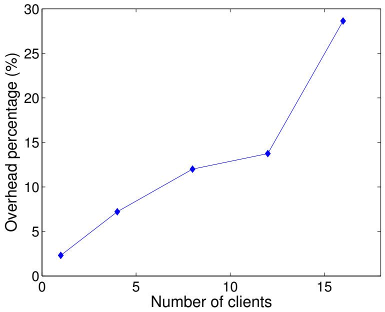


Figure 18: Replication overhead for Sorrento-(12,2) with varying client numbers.

4.8 Small File I/O Operations

Although interactive I/O workload is not the focus of Sorrento, we still wish to verify whether Sorrento performs reasonably under those kind of workload, which primarily consists of small file I/O operations [13, 48].

Interactive Responses. In this experiment, one client issues requests sequentially to an otherwise idle file system and measures the response time. Four types of workload are used: `create` repeatedly creates a new file then closes it immediately. `write` repeatedly opens the files created by `create`, writes 12KB data into it, then closes it. `read` repeatedly opens the files written by `write`, reads 12KB data from it, then closes it. `unlink` unlinks all the files created by `create`. Note that all the measurements below include the cost of `open` and `close` calls. We run these benchmarks on Cluster A and compare Sorrento with NFS and PVFS. For NFS, after each run, we unmount the file system and remount it to get rid of caching. For PVFS, two variations (*PVFS-4* and *PVFS-8*) are used. For Sorrento, four variations are used (*Sorrento-(4 or 8, 1 or 2)*). The results are shown in Figure 19.

	`create`	`write`	`read`	`unlink`
NFS	0.67	2.42	2.93	0.71
PVFS-4	50.3	60.1	60.1	19.4
PVFS-8	60.1	60.3	70.2	22.9
Sorrento-(4, 1)	31.4	43.5	33.5	32.4
Sorrento-(4, 2)	31.3	44.0	33.7	44.3
Sorrento-(8, 1)	32.6	45.4	34.4	32.2
Sorrento-(8, 2)	33.2	46.7	34.8	42.2

Figure 19: Small file I/O request response time (in ms).

As we can see, the overhead of Sorrento and PVFS is significant compared to NFS for small I/O requests, because NFS does not need to provide cluster management and self-organizing features as Sorrento, nor does it provide parallel file system semantics as PVFS. Additionally, Sorrento takes two TCP roundtrips to open a file and three TCP roundtrips to close the file. PVFS would also require multiple TCP roundtrips because metadata and data are stored on metadata server and I/O nodes respectively. Thirdly, NFS is highly optimized for small I/O operations and is tightly integrated with the OS kernel, while both PVFS and Sorrento's storage servers are running at user-level and the file data and meta-data must traverse through kernel-user boundary a few times before they are written to the underlying file system.

We can also see that Sorrento outperforms PVFS by 25-53% for file creation and read/write requests but is slower than PVFS for unlink operations. We explain the reasons as follows. Sorrento's namespace server is more efficient than PVFS's metadata server by storing the whole directory tree in a Berkeley-DB instead of representing each inode using a small file. We can also see that a higher replication degree in Sorrento does not have much impact on the response time even for writes because of our asynchronous update propagation scheme. However, Sorrento eagerly removes all replicas when a file is unlinked, so the response time of `unlink` increases when replication is employed.

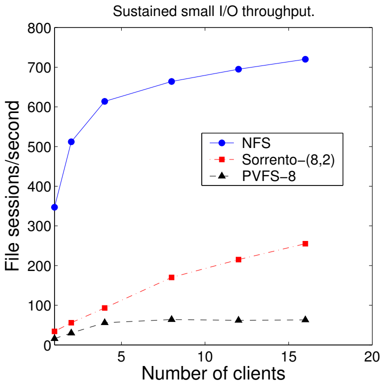


Figure 20: Small file I/O throughput.

Sustained Small File I/O Throughput. Figure 20 shows the sustained throughput of small file operations on Cluster A comparing *Sorrento-(8,2)* with *PVFS-8* and NFS. We launch multiple client processes simultaneously, each of which repeatedly creates a file, writes 12KB into it, then closes it. We calculate the aggregated system throughput in terms of the number of completed file sessions (open/write/close) completed per second. A single NFS server can deliver higher small I/O throughput than both Sorrento and PVFS (it saturates at about 700 sessions/second). This is again attributed to the optimizations incorporated into the NFS server implementation as a result of many years of feedback driven development. Comparing Sorrento with PVFS, we can see that the throughput of Sorrento scales up almost linearly with the number of clients, and we are not able to saturate the system with 16 clients. PVFS saturates at a low throughput (64 sessions/second) and this is largely due to the bottleneck caused by their metadata server. In Sorrento, the services provided by namespace servers is simple (mapping pathnames to FileIDs, and maintaining the versioning information) and thus can be implemented efficiently. Our offline performance tests show that a single namespace server in our current setting can handle 1300 namespace operations per second, which would provide a theoretical upper bound of 400-500 sessions/second. This namespace throughput number still may not be very fast compared to other name servers which can reach 10,000 [5], and we will study if we can use others' work to enhance performance.

Summary of findings: Our evaluation demonstrates the following points. (1) Sorrento is able to gracefully handle node failures through replication and automatically recovering lost data segments. (2) Asynchronous update and copy-on-write improve system performance for replication and versioning. Note that the main reason for Sorrento to employ asynchronous updating and support strong replica consistency at the same time is because of the version-based concurrency control model. (3) For data-intensive benchmarks, Sorrento delivers good parallel I/O performance and is competitive to PVFS while Sorrento provides extra features for self-organizing. (4) Sorrento is also competitive to PVFS for interactive I/O performance. It incurs overhead higher than NFS, which has been optimized for such kind of workload and does not have the overhead of distributed storage management.

5. RELATED WORK

Our work is motivated by previous work on parallel/distributed file systems and cluster-based storage systems such as AFS [29], GPFS [41], Petal [33], PVFS [16], Slice [10], Swift [15], xFS [12], and others [25, 31, 34, 24]. Our work complements those systems with a specific focus on self-organization. Petal and xFS organize storage in RAID volumes, which are difficult to expand incrementally. PVFS focuses on parallel I/O and but not on incremental expansion and fault tolerance. Objectives of GoogleFS [21] are the closest to ours. However, GoogleFS has a narrow focus on applications arising from Web search, and some of its design choices may not be suitable for other applications. For instance, using the centralized server to manage data locations works fine when most of the data objects are large, but further evaluation is needed using applications with a mixture of small and large blocks. Additionally, the design of atomic appending in GoogleFS may not be generalized to support other types of atomic operations.

Peer-to-peer systems such as CFS [18] and OceanStore [32] incorporate some kind of self-organizing design. These systems target the wide area network environment which has a different set of assumptions and constraints. Particularly, they do not export the traditional file system semantics as Sorrento does (CFS is read-only, and OceanStore uses predicated-updates to modify data). Sorrento can be considered as a middle ground between traditional tightly coupled storage systems and loosely coupled peer-to-peer systems. Dynamic distributed data location has been studied in CAN [37], Chord [44], and Tapestry [26]. Because our system only operates in a LAN environment, our solution is simplified and uses consistent hashing [30] to map SegIDs to home hosts.

A version-based data consistency model and immutable files were first proposed in Swallow [38]. Other version-based standalone file systems include Amoeba [34], Elephant [40], and CVFS [43]. Their goals are to protect data loss or to provide information for intrusion analysis, and differ from our base objectives. In Sorrento, we only need to maintain the most recent several versions, and the granularity of versioning is controllable by applications through standard UNIX file system calls. Asynchronous replication has been extensively studied in distributed database research such as the analysis by Gray et. al [23], a simulation study by Anderson et. al [11], and the Bayou project [47]. Sorrento incorporates asynchronous update propagation in the design of a distributed file system where a transaction is implicitly tied with file system operations. Asynchronous replication has also been used in Neptune [42] and Conit [53] to improve system efficiency with relaxed consistency requirement. On the contrary, Sorrento maintains a strong single file serializability.

Online data placement to balance storage usage and system workload have also been studied in CFS [18], in RUSH [27, 28], and by Brinkmann et. al [14]. Unlike these systems, which seek to place data proportional to statically determined *weights*, Sorrento places data based on both workload and storage usage.

Finally, Lustre [3] also uses Object-based Storage Device (OSD) Model [9]. Sorrento differs from Lustre in that all segments are addressed in a single SegID address space, and segments can be freely migrated among physical storage devices without restriction.

6. CONCLUDING REMARKS

The design of Sorrento exploits the low write-sharing characteristics of targeted parallel applications. We choose the version-based consistency model to provide single file serializability, which offers better concurrency compared to a lock-based approach, and significantly simplifies replica consistency management and failure rollback. The self-organizing features of Sorrento improve availability and also enhance parallel performance by integrating an adaptive data placement and location scheme. Our experiments show that Sorrento can achieve high availability while delivering good parallel I/O performance.

Acknowledgment. This work was supported in part by NSF ACIR-0086061/0082666, CCF-0234346, and EIA-0080134, and by Ask Jeeves. We would like to thank our shepherd Ethan Miller and the anonymous referees for their valuable comments and help.

7. REFERENCES

[1] Ask Jeeves, Inc. `http://www.ask.com/`.

[2] CERN (Conseil Européen pour la Recherche Nucléaire) . `http://www.cern.ch/`.

[3] Lustre. `http://www.lustre.org/`.

[4] Project: AVFS: A Virtual Filesystem. `http://sourceforge.net/projects/avf/`.

[5] SPEC SFS97_R1 Results. `http://www.spec.org/osg/sfs97r1/results/sfs97r1.html`.

[6] The Wellcome Trust Sanger Institute. `http://www.sanger.ac.uk/`.

[7] NFS: Network File System version 3 protocol specification. Technical Report SUN Microsystems, 1994.

[8] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215, 1990.

[9] D. Anderson. Object based storage devices: A command set proposal. Technical report, National Storage Industry Consortium, October 1999.

[10] D. Anderson, J. Chase, and A. Vahdat. Interposed request routing for scalable network storage. In *OSDI*, 2000.

[11] T. Anderson, Y. Breitbart, H. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *SIGMOD*, 1998.

[12] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *SOSP*, 1995.

[13] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *SOSP*, 1991.

[14] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform requirements. In *SPAA*, 2002.

[15] L.-F. Cabrera and D. Long. Swift: Using distributed disk striping to provide high I/O data rates. Technical Report UCSC-CRL-91-46, 1991.

[16] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proc. of the 4th Annual Linux Showcase and Conf.*, 2000.

[17] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: a high-performance remote-sensing database. In *ICDE*, 1997.

[18] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[19] A. Fox and E. A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *HotOS-VII*, 1999.

[20] G. R. Ganger and M. F. Kaashoek. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. In *USENIX Annual Technical Conference*, 1997.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[22] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of ACM*, 43(11), 2000.

[23] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[24] J. Hartman, I. Murdock, and T. Spalink. The Swarm scalable storage system. In *Proc. of Intl. Conf. on Distributed Computing Systems*, 1999.

[25] J. Hartman and J. Ousterhout. The Zebra striped network file system. *TOCS*, 13(3), 1995.

[26] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *SPAA*, 2002.

[27] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *IEEE International Parallel and Distributed Processing Symposium*, 2003.

[28] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *IEEE International Parallel and Distributed Processing Symposium*, 2004.

[29] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *TOCS*, 6(1), 1988.

[30] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Levin, and R. Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, 1997.

[31] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. 10(1), 1992.

[32] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[33] E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, 1996.

[34] S. Mullender and A. Tanenbaum. A distributed file service based on optimistic concurrency control. In *SOSP*, 1985.

[35] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. of USENIX Tech. Conf., FREENIX Track*, 1999.

[36] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USITS*, 2003.

[37] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[38] D. P. Reed and L. Svobodova. SWALLOW: a distributed data storage system for a local network. In *Local Networks for Computer Communications*, pages 355–373, North-Holland, Amsterdam, 1981.

[39] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service. In *Proc. of the 17th SOSP*, pages 1–15, 1999.

[40] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *SOSP*, 1999.

[41] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[42] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable replication management and programming support for cluster-based network services. In *USITS*, 2001.

[43] C. A. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *FAST*, 2003.

[44] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[45] H. Tang, A. Gulbeden, J. Zhou, L. Chu, and T. Yang. Sorrento: a self-organizing storage cluster for parallel data-intensive applications. Technical Report 2003-30, UCSB, September 2003.

[46] H. Tang and T. Yang. An efficient data location protocol for self-organizing storage clusters. In *Proc. of ACM/IEEE SC'03*, 2003.

[47] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[48] W. Vogels. File system usage in Windows NT 4.0. In *SOSP*, 1999.

[49] G. Weikum and G. Vossen. *Transactional information systems - theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann, 2002.

[50] R. F. V. D. Wijngaart and P. Wong. NAS Parallel Benchmarks I/O Version 2.4. Technical report, NASA Advanced Supercomputing (NAS), 2003.

[51] J. Wilkes. Data services – from data to containers. FAST Keynote Speech, 2003.

[52] Q. Xin, E. L. Miller, T. Schwarz, S. A. Brandt, and D. D. E. Long. Reliability mechanisms for very large storage systems. In *MSST*, 2003.

[53] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of the 4th OSDI*, Oct. 2000.