UCSB Technical Report

Sorrento: A Self-Organizing Storage Cluster for Parallel Data-Intensive Applications

Hong Tang, Aziz Gulbeden, Jingyu Zhou, Lingkun Chu, and Tao Yang
Department of Computer Science
University of California, Santa Barbara, CA 93106
{htang, gulbeden, jzhou, lkchu, tyang}@cs.ucsb.edu

Abstract

This paper describes the design and implementation of Sorrento – a *self-organizing* storage cluster built upon commodity components. Sorrento complements previous researches on distributed file/storage systems by focusing on incremental expandability and manageability of the system and on design choices for optimizing performance of parallel data-intensive applications with low write-sharing patterns. Sorrento virtualizes distributed storage devices as incrementally expandable volumes and automatically manages storage node additions and failures. Its consistency model chooses a version-based scheme for data updating and replica management, which is especially suitable for data-intensive applications where distributed processes access disjoint datasets most of the time. To further facilitate parallel I/O, Sorrento provides load-aware or locality-driven data placement and an adaptive migration strategy. This paper presents experimental results to demonstrate features and performance of Sorrento using both microbenchmarks and trace-replay of real applications from several domains, including scientific computing, data mining, and offline processing for web search.

1 Introduction

Large-scale storage systems that consist of a cluster of nodes with locally attached disks have become a cost-effective solution for a large class of data-intensive applications, ranging from high-performance computing to data mining and network services [13, 14, 38, 46].

There have been an extensive body of research on or relevant to storage clusters, such as Petal [30], xFS [8], AFS/-Coda [28], PVFS [13], and GoogleFS [18]. The important aspects being studied are: resource virtualization, availability, and scalability. *Resource virtualization* means that distributed disks in a storage cluster must be virtualized as a single disk (or a few of them). Additionally, files should be named and addressed in a location-transparent fashion. *Availability* requirement means that a storage cluster must mask out component failures and make data available all the time. *Scalability* means that the system is able to deliver scalable performance as the number of storage nodes grows.

The Sorrento project is based on the pioneering effort of previous work and complements their solutions by focusing on the issues of *incremental expandability* and *manageability*. As hardware cost decreases steadily, a cluster must be able to expand incrementally as the application demand for storage increases [34]. Second, human errors are a significant source of unmasked system failures [16]. To reduce the chance of operational errors, we must reduce the involvement of administrators and make the system easy to manage. Thus an important aspect in the design of Sorrento is that we try to make the storage cluster *self-organizing*, i.e. the system is able to automatically adapt to environment changes such as node additions or component failures. Specifically, the whole system is built upon many independent entities, which constantly monitor and dynamically adjust their behavior. These entities collaborate together and automatically perform tasks such as resource virtualization, data location and placement, replication, and recovery.

Additionally, the design of Sorrento is optimized toward a class of data-intensive applications that exhibit different workload from file systems used in interactive environments such as desktop PCs. Specifically, the applications we target are data-intensive applications involving a large number of parallel processes issuing concurrent I/O requests, and they operate on disjoint data sets most of the time. Examples of such applications include partitionable network services [17, 38], graphical rendering for movie scenes, satellite image processing, and offline processing of Web documents in search engines [1, 2]. As a result, Sorrento uses a version-based update and replica management scheme that allows applications to exploit I/O parallelism efficiently.

Other design features of Sorrento are summarized as follows: (1) Storage resources are organized in expandable virtual volumes. A logical file may be divided into several variable-length data segments. Segments are stored in their entirety on native file systems, and are addressed by location-independent SegIDs. Segments can be replicated on or migrated to any cluster nodes. The physical location information of segments are distributed among cluster nodes as soft states. (2) Our data location scheme can be regarded as a variation of the Chord [40] protocol, with various changes that make it more suitable for a LAN environment. (3) Data locations are chosen by a load-aware placement policy that can balance both storage utilization and I/O workload. The system constantly monitors node departures and joins, and recreates data segments or migrates them accordingly. We further provide a locality-driven placement policy for applications that exhibit good access locality.

We have implemented a Sorrento prototype with all these features incorporated. We have used a combination of microbenchmarks and application trace replay to demonstrate the scalable performance and the effectiveness of self-organizing features of Sorrento. The rest of the paper is organized as follows: Section 2 outlines our design assumptions and provides an overview of the system. Section 3 gives details on the system design. Section 4 presents the system implementation and evaluation results. Section 5 describes related work, and Section 6 summarizes the conclusions.

2 Design Assumptions and System Overview

2.1 Design Assumptions

Our targeted hardware architecture is PC or workstation clusters. Each cluster node may contribute storage resources to globally accessible Sorrento volumes. We also assume a storage cluster runs in a trusted environment, and security issues are not the focus of this research project. However, the system does need to cope with failures caused by various reasons including software bugs and human operation errors.

While our system works for general applications, our targeted workload typically involves intensive read and write requests through a large number of parallel processes and these processes operate on disjoint datasets. Thus our design consideration seeks to to reduce the overhead of data updates and replica management, and to exploit I/O concurrency among application processes as much as possible.

2.2 System Overview

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. Each volume is identified by an ASCII name. Data stored in each volume are organized in a hierarchical directory tree, which is maintained by namespace servers. In the current implementation, there will be one instance of namespace server per volume.

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

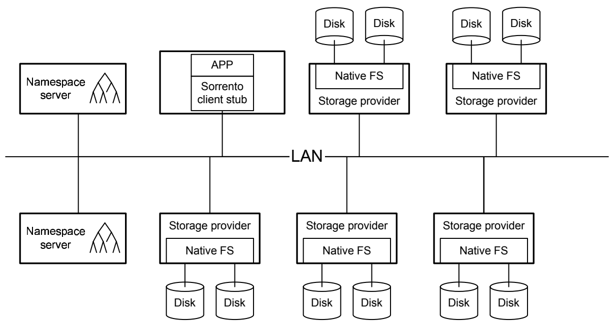

Figure 1: Sorrento system architecture.

A Sorrento deployment can be configured and maintained incrementally without interrupting the normal operation of the whole system. To add storage resources, we just need to attach more disks to a storage provider, configure it to join a designated volume, and connect it to the network. The maintenance is also simple. To repair a failed node or to recycle a node (maybe because it runs out of its useful life span), we can directly shut down the machine. When a machine is repaired, it can be directly connected to the network without the need to reformat the partitions, and the system will automatically determine what data are still current and what are outdated.

2.3 Programming Interface

Sorrento provides multiple flavors of client-side programming interfaces. The basic Sorrento API layer exports an NFS-style interface, in which operations are based on opaque file and directory handles [4]. Upon this layer, we have implemented another library interface that is similar to the UNIX file-system calls. We are also working on a kernel file system module to make a Sorrento volume mountable to the local file system.

Deciding on which interface to use is a tradeoff between functionality and transparency. Accessing a Sorrento volume through operating system calls would be convenient and would allow existing tools and applications to run without modification; however, to achieve the best efficiency or to utilize the functional extensions offered by Sorrento, the user-level library interface must be chosen. An application can fine-tune how the data of a certain file should be managed through these extensions, such as replication degree, data organization, and data placement policies.

3 System Design

We break the core Sorrento system into seven interconnected components, the dependences among which are illustrated in Figure 2. The remaining discussion generally follows the *bottom-up* order, with the exception of data placement and migration, which we will leave to the end.

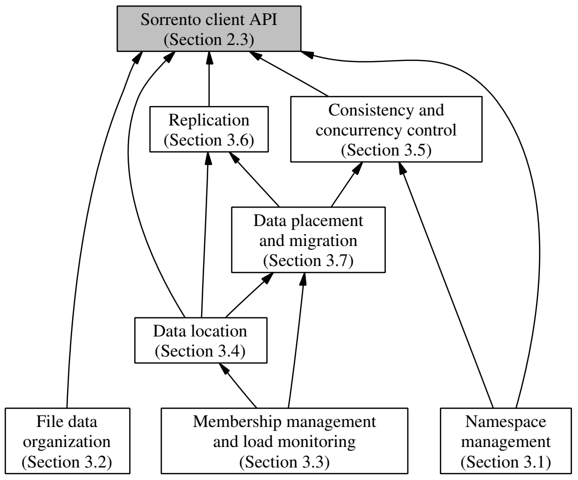
Sorrento client API (Section 2.3)
Replication (Section 3.6)
Consistency and concurrency control (Section 3.5)
Data placement and migration (Section 3.7)
Data location (Section 3.4)
File data organization (Section 3.2)
Membership management and load monitoring (Section 3.3)
Namespace management (Section 3.1)

Figure 2: Sorrento software components and their interdependences. Sorrento API (the gray box) was discussed previously in Section 2.3. An arc $A \rightarrow B$ means that B relies on the functionality of A.

3.1 Namespace Management

As mentioned previously, Sorrento separates the namespace management from storage management. Such a separation is not a new concept and has been adopted in other work such as Zebra, NASD architecture, PVFS and GoogleFS [13, 18, 19, 23]. 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 span multiple data objects. While for application I/O, it is more important to serve large I/O requests fast, and typically operations on different files are independent. Separating these two types of workloads allows us to make optimizations targeted to their specific workload.

The namespace servers are 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 FileID, the file's latest version, and the timestamp information, etc.

One notable difference of our namespace server design from other research such as PVFS [13] or GoogleFS [18] is that the namespace servers do not keep the physical locations of data blocks for files. FileIDs are persistent across the lifespan of files, and are location independent. Such a design is necessitated by the fact that in Sorrento, data blocks may constantly migrate among providers to balance storage usage or I/O workload, or to exploit data locality. Thus, requiring the namespace servers to track the locations of mobile data blocks would make them vulnerable to become a performance bottleneck.

In our current implementation, the directory tree is stored in a database using Berkeley DB [33]. We employ a combination of write-ahead logging and checkpointing to allow a namespace server to recover from disk failures. We can also adopt solutions proposed in several previous work that improves reliability and availability of a name server through replication or directory tree partitioning [7, 10].

3.2 File Data Organization

Sorrento adopts the conventional representation of a file as a linear array of bytes. The actual file data may be split into multiple variable-length *data segments* and stored on different storage providers. Each data segment is kept in its entirety on a storage provider. How to organize the data segments as a linear byte array 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 logical file's FileID is the same as the SegID of the index segment.

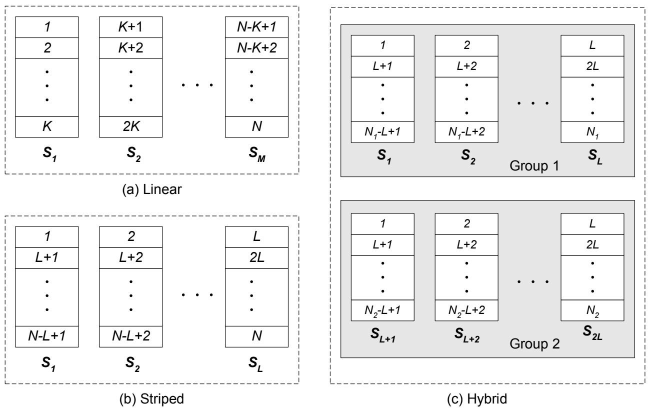

Figure 3: Illustration of data organization modes in Sorrento. Each cell in a segment is a fixed block, and is marked by its sequence number in the logical file (or a stripe group in the Hybrid mode). The number of segments in this illustration for the Linear, Striped, and Hybrid modes are M, L and $2L$ respectively.

Sorrento currently supports three data organization modes, as illustrated in Figure 3:

Linear: In the Linear mode, the index segment specifies an order of all the data segments, and the byte array is a linear concatenation of the data segments in that order (Figure 3 (a)). Linear mode is suitable for sequential access of file data.

Striped: In the Striped mode, file data are striped across a fixed number of data segments (Figure 3 (b)), which is similar to RAID-0. All the data segments are of equal size. Striped mode is suitable for applications that need to tap into the aggregated I/O bandwidth of multiple disks. One

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

restriction of the striped mode is that the file size (or its maximum size) and the number of data segments must be specified when the file is created.

Hybrid: The Hybrid mode can be considered as a combination of the Linear and Striped modes (Figure 3 (c)). The data segments are organized in several groups. Within each group, the data are organized in Striped mode. Different groups concatenate linearly. The Hybrid mode provides the parallel I/O performance of the Striped mode, and does not require applications to know the file size in advance.

The three data organization modes being supported are not meant to be conclusive, and it is fairly easy to extend Sorrento to support other data organization schemes.

Choosing segment sizes. Sorrento automatically determines the segment sizes for the Linear or Hybrid mode. In the Linear mode, starting from offset zero, we gradually increase the segment size exponentially until it reaches a predefined maximum segment size. After that, all segments will be of the maximum segment size. In our current design, the size of the i-th segment (i starts from 0) is determined by the following formula (in MB): $\min\left\{512, 8^{\left\lfloor\frac{i}{8}\right\rfloor}\right\}$. The underlying motivation behind this sizing scheme is to use small segments for small files, and large segments for large files. In the Hybrid mode, a similar scheme is used: Suppose all segment groups are of the same size j, then the segments in the i-th segment group (i again starts from 0) is: $\min\left\{512, 8^{\left\lfloor\frac{i\times j}{8}\right\rfloor}\right\}$. Note that Sorrento does not preallocate space for a whole segment. Instead, it expands the last segment (or the last segment group in the Hybrid mode) when a file grows.

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. Currently, the maximum attachable file size is set to 60KB to fit in a UDP packet.

3.3 Membership Management and Load Monitoring

Previous systems, such as xFS or PVFS, keep a table of I/O servers (or storage providers in Sorrento's term) 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 fairly 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 [38]. Specifically, all storage providers periodically send out heartbeat packets through a multicast channel, and the membership manager can construct the set of live providers by monitoring the same multicast channel. If a process fails to receive heartbeat packets from a provider for a prolonged period (five times the heartbeat announcement interval), the membership manager will remove that provider from its membership set. The heartbeat packets also include the load information and storage resource availability of the storage providers.

3.4 Data Location

As mentioned previously, one design challenge 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 (or providers) stores the segment.

3.4.1 The Base Scheme

The 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 [40] in various ways.

The task of data location is distributed 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* [27] 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.

On each storage provider, we maintain a location table that maps SegIDs to segment owners. The location table is 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 will trigger the update of the location table:

(1) **Periodic content refreshing.** Each provider (as an *owner*) periodically refreshes the location tables on other providers (as *home hosts*) by sending the SegIDs of locally stored segments to their corresponding home hosts. Content refreshing is important since location tables are considered soft states in our system and a node may fail to keep its location table up-to-date due to various reasons (e.g. unexpected faults). In our test environment, we set the table refreshing cycle to 15 minutes, namely the location 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 (asymptotically optimal).

(2) **Node-join notification.** This event happens when the membership manager adds a new provider (provider A) to the live set of providers. Note that this event could imply two possible situations: (i) an existing provider (B) learns

about a newly joined provider (A), and (ii) a newly joined provider (B) discovers an existing provider (A). In response to this event, provider B will schedule 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 short 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 locations were previous managed by 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, or when a local segment's home host is changed. In response to these events, 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 fast updating and it happens between periodic content refreshing described above.

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. To reclaim resources taken by the garbage entries, every entry in the location table is marked with its last refreshing time. Since valid entries will be refreshed periodically while garbage entries will never be refreshed, the latter can be identified based on their ages and eventually be purged.

3.4.2 The Backup Scheme

The base scheme described above may occasionally fail either because a client asks 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, the client will fall to a backup scheme in which it will simply query all the providers by issuing a request through the multicast channel.

3.5 Data Consistency and Concurrency Control

Sorrento provides a version-based consistency model, which bears similarity to Amoeba [32]. Such a choice follows our objective of optimizing performance for application workloads with low data-sharing as discussed in Section 2. However, it should also work for general I/O workload. In this section, we will briefly describe the semantics of such a model, how it is implemented in Sorrento, and finally provide justifications of such a design.

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. Sorrento forbids version branching, which complicates the data consistency model and will likely offer programmers more confusion than flexibility. The version-based semantics can be implemented hidden behind conventional file system primitives: a commit operation is implicitly 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. We employ the standard copy-on-write technique to avoid the overhead of making complete shadow copies of segments. 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 structure to maintain the mapping from region ranges to physical segments where the valid data for the shadow copy can be located. The index structures are kept in memory, and will be flushed to disk when the shadow copy is committed and becomes immutable.

To cope with failures of client applications, which may leave uncommitted shadow segments as garbage, all shadow copies are given an expiration time. The application must either commit a shadow segment before its expiration, or reset the expiration timer before it expires.

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-lock 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 [29] or Bayou's merge procedures [41]), 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) [44] to ensure the atomicity of such an operation, whose details are omitted here due to space constraints.

To conserve storage resource, Sorrento consolidates earlier versions of a segment and only keeps one or a few latest stable versions. In the future, we plan to allow users to specify or automatically detect milestone versions that will never be consolidated, a feature similar to the Elephant file system [36].

Justification The choice of a version-based data consistency model may sound surprising for those who have been accustomed to the UNIX semantics. One may feel that there could be too many versions in the system and the efficiency could be a problem. There are four reasons behind our design:

(1) The decision is driven by the suitability for our targeted applications and the runtime efficiency. For parallel applications which access disjoint datasets most of the time, update conflicts seldom occur. Thus optimistic concurrency control is preferred. Additionally, for several applications that Sorrento intends to support, the correctness of an operation requires the atomicity of multiple reads and writes on different offsets of a file. Reads and writes from two different operations cannot interleave.

(2) Storage overhead is not an issue due to the use of copy-on-write and version consolidation. Only one or few versions are needed for each object. In the latter case, older versions serve as backups in the events when the latest version gets lost due to failures.

(3) Implementing UNIX semantics could incur unnecessary cost since a read/write request may involve multiple segments, and making it atomic will require proper coordination among the owners of these segments. On the other hand, in many cases, our applications do not need to see the immediate changes made by other processes, so UNIX semantics are unnecessary. If UNIX semantics are indeed desirable, it can be emulated by issuing a `sync` call after each write.

(4) 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 [18], which is illustrated in Figure 4.

```
// 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 4: Implementation of *atomic append*.

For applications (such as DBMS) that prefer to implement their own data consistency, Sorrento also provides an option to disable data versioning. In this case, all reads and writes will be directly applied to the data segments. However, since our replication scheme is dependent on versioning, enabling this option will also disable replication. Currently, this option is being used to support the parallel I/O byte-range sharing primitive [21].

3.6 Data Replication

Sorrento tolerates component failures through data replication. Sorrento lets applications customize the replication degree for each file to fit the nature of the file. For instance, in a typical search engine, the same set of pages are crawled and indexed periodically. If we lose a small fraction of the pages, we can borrow the ones crawled during the previous batch, and users will probably notice little or no difference. In this case, it is desirable not to replicate the page sources. On the other hand, inverted word indexes are time-consuming to generate, and are read-only afterward. So they are better to be replicated with a high replication degree.

Sorrento's version-based data consistency simplifies the management of replica consistency. We can identify whether two replicas of a segment are the same or which one is older by comparing their versions. This allows Sorrento to propagate updates to replicas lazily [20]. 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 send the updated location information to the home host of the segment, which also contains the new version number. On the home host side, whenever it inserts or refreshes an entry in the location table, it will check whether the segment replicas are of the same version. If version discrepancy is found on different owners, the home host will notify those with older versions to synchronize with the latest version owner. Replication degree is maintained similarly. When a home host finds a segment has fewer replicas than the specified degree, it will choose new replica sites and notify them to request a copy from existing owners.

Such a lazy propagation scheme allows writes to be executed as fast as if there were no replication. Update propagation is done in background (after a segment is committed) and is not in an application's execution path. Such an approach also brings one drawback. Namely, an application would not have any guarantee whether its latest changes have been successfully replicated when it commits a file. For certain applications that do require such a guarantee, we also offer synchronous commitment as an option when an application closes a file. In a synchronous commitment, the application will query the home host of the set of accessible replicas, detect version discrepancies among them, and push changes to older replicas before it returns the control back to the application.

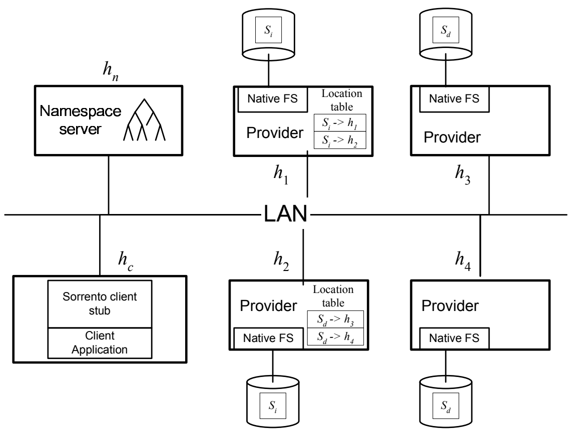

Figure 5: The initial setting of a simple example. File `/foo` is represented by an index segment S_i and data segment S_d. Both segments are replicated twice. The home hosts of S_i and S_d are h_1 and h_2 respectively. h_n is the namespace server, and h_c is the node where client application is launched.

Putting All Things Together We demonstrate how different components and entities in Sorrento work in sync through a simple example. This example involves four storage providers (h_1, h_2, h_3, h_4), and one file `/foo`, which is physically represented by an index segment (S_i) and a data segment (S_d). Both segments are replicated twice (S_i 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 (running on host h_c) opens the file `/foo`, performs a write, and closes the file. The initial arrangement of this example is shown in Figure 5, where h_n is the namespace server. The timeline of the execution is illustrated in Figure 6. The activities carried out in each step are explained in Figure 7.

Note that although the whole process might seem quite involving, the actual I/O transfer goes directly between the client and the storage provider (steps (3)-(5)) 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.

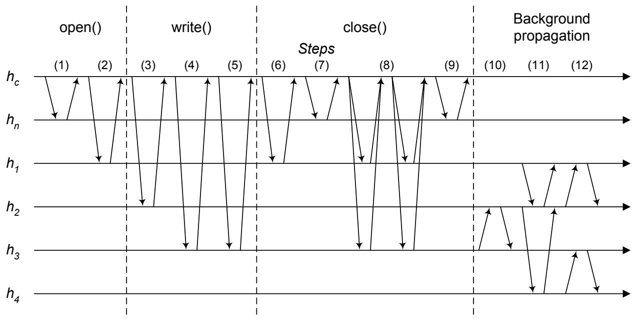

Figure 6: The timeline of activities for a simple example, in which an application opens a file, writes to it, and closes it. The file has two segments S_i and S_d, whose home hosts are h_1 and h_2 respectively. S_i is replicated on h_1 and h_2, and S_d is replicated on h_3 and h_4. Time flows from left to right. Arcs between different timelines stand for communications. We also illustrate the steps corresponding to `open()`, `write()`, `close()`, and the background propagation process.

Step	Activity
(1)	The client retrieves file `/foo`'s FileID from the namespace server.
(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, and succeeds.
(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 7: Activities carried out in the steps of Figure 6.

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 (1) to balance I/O load and (2) to balance storage usage among the providers. As a result, data placement in Sorrento is closely coupled with data migration. Note that our two balancing objectives are to some degree correlated. For example, studies have shown that a typical UNIX file system performance degrades when it is close to saturation [31], so balancing storage usage among providers could potentially improve I/O performance. On the other hand, these two objectives may also conflict with each other. For instance, when a new provider joins the network, we may want to fill it up quickly by placing more new segments on it according to the second objective (balancing storage utilization). However, since newly created segments are more likely to be accessed subsequently by their creators, it will cause a quick load increase on that provider at a later time.

3.7.1 The Base Scheme

We consider that each segment has its *temperature*, which may change over time. A *hot* segment is one that is being actively accessed recently; while a *cold* segment is one that has not been touched for quite a while. We use the last access time (LAT) of a segment to measure its temperature: a more recent LAT stands for a higher temperature. The distribution of segments in relation to temperature is typically bimodal – most of the segments are either hot or cold, with few *lukewarm* ones spread in between. To balance I/O load, we need to migrate hot segments from heavily loaded providers to lightly loaded providers.

The problem of selecting the location of a new segment appears in three contexts: (1) placing a newly created segment; (2) making a new replica of an existing segment; and (3) migrating an existing segment to a different location. Note that the third case is equivalent to making a new replica of the segment somewhere else and then erasing the local copy. We use the same provider selection algorithm in all three cases. The algorithm only relies on the load information, which is maintained by the local membership manager, and is randomized: First, for each candidate provider p_i, we calculate a weight w_{p_i}. Second, we randomly select a provider among the candidates, in which p_i has the probability of being chosen proportional to its weight. The weight of a provider considers both the provider's current workload (through the *load factor* f_l) and its available storage resources (through the *storage factor* f_s). Specifically, f_l and f_s are calculated as follows:

$$f_l = \min\left\{10, \left(\frac{1}{l} - 1\right)\right\}, \text{ and } f_s = \min\left\{10, \log_2 \frac{S}{s}\right\}.$$

In the above formulas, l ($0 \leq l \leq 1$) is the provider's CPU and I/O wait load, S the provider's available space, and s the segment size ($s \leq S$)[2]. These formulas are based on some empirical studies. The storage factor is calculated as the logarithm of the ratio between the available space and the segment size. The load factor is calculated as the inverse of the current workload. We add adjustments to keep both factors within range $[0, 10]$. Given f_l and f_s of a provider p_i ($0 \leq f_l, f_s \leq 10$), its weight can be calculated as $w_{p_i} = f_l^{\alpha} \times f_s^{1-\alpha}$, where α is within $[0, 1]$ and can be used to adjust the favoritism of the storage factor or load factor in the final weight. For instance, we can use a small value of α if we want to place a file on providers with more storage space. By default, we chose each file's α value to be 0.5.

In terms of data migration, we still need to decide when and what segments should we migrate. These questions are relatively straightforward to answer. Migration is desirable when there is significant imbalance of either I/O load or storage utilization. To this end, we measure a provider's I/O load using the EWMA[3] of the I/O wait percentage, and the storage utilization using the percentage of consumed space. *Significant imbalance*, translating into mathematics, means that some provider's I/O load or storage utilization is outside the $\pm 3\sigma$ range of the cluster-wide average, where σ is the standard deviation. The migration process on a provider will be activated if the provider's I/O load or storage utilization is among the highest 10% of all providers, and is higher than the system-wide average plus three times the standard deviation.

As to what segments to migrate, the answer depends on the providers. For providers with high I/O load, we prefer migrating hot segments because our goal is to lower their I/O load. Additionally, we choose the value of α to be 0.8, favoring providers with low load as the destination of migration. For providers with high storage utilization, we prefer migrating cold segments, and set the value of α to be 0.3, favoring providers with low storage utilization.

To avoid oscillation in which a segment may be migrating back and forth due to sudden bursts of traffic, the migration design is made once every minute. Additionally, to prevent the traffic generated from data migration to disturb the normal operation of the system, we only allow one active data migration process per node.

3.7.2 Optimizations and Customizations.

In the base data location scheme, each data access requires at least two network operations: first contacting the home host, and then interacting with the actual owner. For large segments, such an overhead would 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 modify our provider selection algorithm and increase the weight of a home host by a factor of $3N$ (N being the number of providers) to make a home host more favorable to be the owner for a small segment. We also adjust how a provider responds to location table refreshing requests: for small segments, we will send the actual data instead of the SegIDs to the remote provider.

To increase data survivability against failures, we also seek to store replicas of a segment on different providers. This is achieved by excluding the current replica holders from the candidate set when selecting a new replica site. We also plan to extend our replica placement policy to support more sophisticated requirement, such as placing replicas on different racks as GoogleFS does.

Another aspect of our data placement and migration scheme is that it is customizable by user applications. First, an application can specify the favoritism parameter α for

[2]If a segment's size is not know, as in the case when we create a new segment, we use the potential maximum size determined in our segment sizing scheme.

[3]EWMA stands for Exponentially Weighted Moving Average.

each file during their creation time. The customized α values will then be used in place of the system-wide defaults during the placement and migration of that file. For instance, if a file will be accessed frequently, the application may want to place it on a lightly loaded node; on the other hand, if the file is meant to be created and put away (such as a checkpoint file or crawled HTML pages), we do not have to worry about the load of its destination.

We also support a special *locality-driven* data placement policy. This is again based on our observation that for many applications, the application dataset is partitioned and different processes will access disjoint sets of data, exhibiting good locality. Thus, it is desirable to take advantage of such kind of 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 a significant percentage of the traffic it receives is from that provider. The threshold percentage is specified as a parameter to the locality-driven placement policy. To avoid instability, the threshold value must be greater than 50%. Traffic monitoring is achieved by maintaining an access history for each data segment managed under locality-driven policy. We also limit the memory consumption by only keeping the latest one thousand accesses for the most recently accessed one thousand segments. Through a segment's access history, we can derive the traffic volumes generated by remote nodes.

4 Implementation and Evaluation

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. For instance, we have not implemented any caching for either namespace server or file data.

The following evaluations seek to answer two questions: (1) What is the overhead and scalability of Sorrento in comparison with NFS and parallel file systems such as PVFS (Section 4.1 and 4.2); (2) How effective are the self-organizing features of Sorrento, such as online data recovery and data migration (Section 4.3, 4.4 and 4.5).

The evaluations are conducted on two clusters A and B. The hardware and software configuration of these two clusters are summarized in Figure 8. In both clusters, all nodes are connected to the network through Fast Ethernet links and none of the following experiments would saturate the switches. 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 lazy propagation for replica updates and load-aware data placement and migration. 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 and application trace replay as our evaluation methodology. The latter one is chosen for several reasons: (1) This enables us to quickly test how the system would perform under different kind of application workloads without either modifying the applications or extending the underlying I/O sub-system being used (such as ROMIO [3, 42] or the kernel file system module); (2) We have a more controllable testing environment and the results can be reliably reproduced and analyzed. (3) Practically, some applications cannot be run directly. For instance, we cannot run a crawler application on either clusters because internal nodes are isolated from the Internet. We do acknowledge that through trace replay, we may not accurately represent the exact workload imposed by the applications; however, we believe the results are adequate to provide us reasonable indications on how the system would perform under various types of workload so that we can make more informed design decisions during our development. 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 5% to 20% depending on the granularity and frequency of requests.

4.1 Small File I/O Operations

Although interactive I/O workload is not the focus of Sorrento, we do want to verify whether Sorrento performs reasonably under those kind of workload, which primarily consists of small file I/O operations [9, 43].

4.1.1 Interactive Responses

In this experiment, a single client process 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`. 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 NFS caching. For

	Cluster A	Cluster B
CPU	30 dual P-II 400MHz nodes (10 with exported storage)	46 nodes (38 with exported storage): 8 dual P-III 1.3GHz, 30 dual P-III 1.4GHz, 4 quad Xeon 1.8GHz and 4 quad Xeon 2.4GHz.
Memory	512MB per node	4GB per node
OS	RedHat Linux (kernel ver 2.4.18)	RedHat Linux (kernel ver 2.4.20)
Disk drives	10 disks: 2 Seagate Cheetah ST373405LW (10K rpm, 5.1ms s.t.) and 8 Seagate Barracuda ST336737LW (7.2K rpm, 8.5ms s.t.)	114 disks: 3 Seagate Cheetah ST336704LC (10K rpm, 5.1ms s.t.), 99 Hitachi Ultrastar DK32EJ-72NC (10K rpm, 4.9ms s.t.), 6 Seagate Cheetah ST373405LC (10K rpm, 5.1ms s.t.), and 6 Fujitsu Enterprise MAN3735MC (10K rpm, 5.0ms s.t.).
Total capacity	210 GB	6.55 TB
Connectivity	All nodes connected to a Lucent Cajun 550 through Fast Ethernet links	Nodes are connected to two HP 5308XL switches through Fast Ethernet links. The two HP switches connect to a Cisco catalyst 6509 through Gigabit Ethernet links.

Figure 8: Hardware and OS settings of two clusters. In Cluster A, 10 of the 30 nodes export storage. In Cluster B, 38 nodes export storage. Each of those nodes exports a software RAID-0 partition consisting of three SCSI partitions.

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 9.

	`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 9: 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, NFS is highly optimized for small I/O operations and is tightly integrated with OS kernel, while for both PVFS and Sorrento, the 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. Thirdly, in Sorrento, it 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. However as we will show later, Sorrento outperforms NFS for large I/O requests and in terms of system scalability when the number of concurrent clients increases.

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 BerkeleyDB 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 lazy propagation scheme. However, Sorrento eagerly removes all replicas when a file is unlinked, so the response time of `unlink` increases when replication is employed.

4.1.2 Sustained Small File I/O Throughput

We also evaluate the sustained throughput of small file operations. We launch multiple client processes simultaneously, each of which repeatedly creates a file, writes 12KB into it, and closes it. We calculate the aggregated system throughput in terms of the number of completed file sessions (one create/write/close is counted as one session). We compare *Sorrento-(8,2)* with *PVFS-8* and NFS. The experiment is conducted on Cluster A and the results are shown in Figure 10. The x-axis is the number of clients, and the y-axis is the measured throughput.

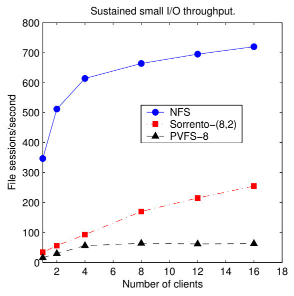

Figure 10: Small file I/O throughput. Throughput is reported as the number of file sessions (open/write/close) completed per second.

As we can see, a single NFS server can deliver a higher small I/O throughput than both Sorrento and PVFS (it saturates at about 700 sessions/second). This is again due to the optimizations of NFS server implementation over years of experiences.

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. On the other hand, PVFS saturates at a low throughput (64 sessions/second). This is largely due to the bottleneck caused by their metadata server. In Sorrento, the services provided by namespace servers is very 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 is able to handle 1300 namespace operations per second, which would provide a theoretical upper bound of 400-500 sessions/second.

4.2 Large File I/O Operations

We further evaluate the large file I/O performance using microbenchmarks and application trace replay.

4.2.1 Microbenchmarks

In this experiment, benchmark `bulkread` repeatedly reads 4MB data at random offsets (aligned at 4KB boundary) from a set of 512MB-large files. Similarly, benchmark `bulkwrite` repeatedly writes 4MB data at random offsets to a set of 512MB-large files. 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*, and *Sorrento-(8,2)*. Different client processes access disjoint sets of files. For Sorrento and PVFS, a total of 160 files (80GB) are pre-populated. For NFS, a total of 30 files (15GB) are pre-populated. In each run, a client reads or writes 256MB data. Note that since the total dataset cannot be fit in memory, and the data being accessed are at random offsets, the impact of file system caching is negligible. For `bulkwrite`, we also show the performance of Sorrento using eager propagation (*Sorrento-(8,2), eager*). The results are shown in Figure 11.

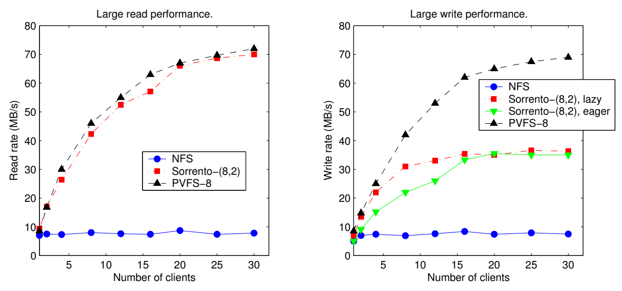

Figure 11: Large file read/write performance.

As we can see, NFS performs the worst and the read or write rates saturate at 8MB/s. On the other hand, both PVFS and Sorrento are able to scale up the transfer rates with the number of clients. For read rates, Sorrento and PVFS perform similarly. For write rates, PVFS outperforms Sorrento by a factor of two. This is because Sorrento 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, the peak transfer rates under eager propagation and lazy propagation are very close. However, lazy propagation is able to deliver a higher transfer rate when the number of clients is small and the system is underloaded. This is because under lazy propagation, a client does not have to wait until all replicas are updated before it can issue the next request.

4.2.2 Application Benchmarks

We further access the performance of Sorrento through trace-replay of two real applications. The first is the BTIO benchmark from NAS Parallel Benchmark Suite (NPB) [45]. 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 [5] (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 B. For BTIO, four trace replayers wrote 2.7GB data and read 1.7GB data[4]. For PSM, eight trace replayers read a total of 3.1GB data (there is no write operations for PSM). BTIO uses PVFS's `list-write` primitive, which is emulated in Sorrento through asynchronous I/O calls, and we disabled version-based data management to support concurrent writes to different byte ranges. In both settings, the trace replayers are launched simultaneously, and they issue requests sequentially as fast as they can. For both applications, we compare *Sorrento-(8,1)* with *PVFS-8* and NFS. We show the results (Figure 12) 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 12: Performance comparison using NPB BTIO and parallel Protein Sequence Match.

As we can see, for both applications, the workload is distributed to clients in a very 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 does have an 11% edge over Sorrento for BTIO. This is because after all, PVFS has been specifically tailored for this type of applications, while our prototype system is far from optimal.

4.3 Handling Node Failures and Additions

In this section, we evaluate how the system handles node failures and additions. We employ 10 nodes from Cluster B as storage providers, and populate the system with 200 512MB files, each has three replicas (totally 300GB data). A constant workload of three `bulkread` and two `bulkwrite` processes is present during the experiment, which is at about 50% of the system capacity. Each client writes to its log the amount of data being read/written every three seconds, and we calculate the aggregated transfer rates offline.

[4]BTIO has five different classes, and this experiment uses the class B setting.

We knock down one storage provider at time 30 (seconds); and then add a new storage provider at time 45.

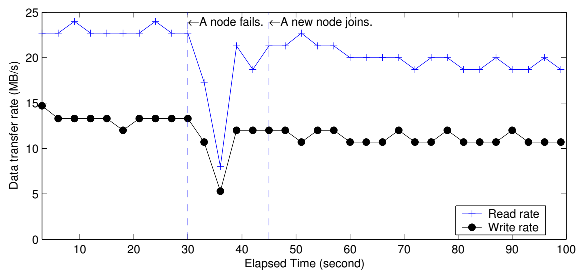

Figure 13: Handling node failures and additions. A storage provider fails at time 30, and a new node joins at time 45.

The results are shown in Figure 13. As we can see, right after the failure of a storage provider, the transfer rate drops because the requests issued to the failed node are all timed out. After that, the system quickly adjusts the content of location table and the transfer rates recover to about 94% of the inital setting, this is due to the fact that now each remaining storage providers receives an increased share of load from clients. About 30 seconds later, the transfer rate decreases to the 85% of the initial setting. This is caused by the data recovery process that seeks to restore the replication degree of under-replicated segments. Eventually, all lost replicas are restored after 20 minutes (not shown in Figure 13). This experiment confirms that Sorrento is able to tolerate node failures and deliver sustained I/O throughput during the data recovery process.

4.4 Load-Aware Data Placement and Migration

We evaluate the effectiveness of our load-aware data placement and migration policy through a search engine crawler application from Ask Jeeves [1]. In this application, a number of crawlers are assigned disjoint sets of seed URLs, and each one crawls within a confined set of URL domains and stores the pages to the storage system. Pages from one domain are stored in a single file. Statistical data from Ask Jeeves shows that the number of pages from a single domain can range from hundreds to millions. And there is typically a speed discrepancy of more than ten folds among crawlers. The high skewness of the file size distribution and I/O workload distribution makes it a good candidate to study the effectiveness of Sorrento's load-aware data placement and migration.

In this experiment, we use a 10-node Sorrento deployment at Cluster B, and launch 50 crawlers on the same set of storage providers (five crawlers per node). The trace replayers are multi-threaded, and they replay the traces collected from search engine crawlers deployed in Ask Jeeves's production environment. The trace replayers emulate the effect of Internet latency when fetching a page by blocking themselves for the same amount of time. The page files are not replicated. We compare three variations of Sorrento:

	Node with lowest storage usage	Node with highest storage usage	Unevenness ratio
Sorrento-random	7.1%	35.3%	4.97
Sorrento-space	9.1%	26.2%	2.88
Sorrento-migration	10.2%	18.5%	1.81

Figure 14: Comparison of storage usages of the crawler application under three data placement and migration schemes. *Sorrento-random* places data uniform randomly on all servers; *Sorrento-space* places data based on storage usages of the nodes. *Sorrento-migration* is *Sorrento-space* with online migration enabled.

(1) uniform random placement policy and no data migration (*Sorrento-random*); (2) space-usage based placement (setting α to 0) and no data migration (*Sorrento-space*); (3) space-usage based placement with data migration enabled (*Sorrento-migration*). We choose α to be 0 because the I/O workload generated by the crawlers is fairly light (totally less than 10MB per second). For each setting, we run the experiment for 12 hours, and report the nodes with the lowest and highest storage usage percentage at the end of the experiment. We use the ratio between the highest storage usage and the lowest usage to measure the unevenness of the final data placement. Each run starts with an empty file system and totally 243GB data are written. The results are shown in Figure 14. As we can see, because of the heavily skewed file size distribution, a uniform random placement performs poorly, with an unevenness ratio of 4.97. Sorrento-space performs better by placing data using realtime storage utilization information, however, since a file's size is not known when it is initially created, the unevenness ratio is still quite high (2.88). Through online data migration, Sorrento-migration is able to reduce the unevenness ratio to 1.81. This confirms the effectiveness of Sorrento's load-aware data placement and migration policy.

4.5 Locality-Driven Data Placement and Migration

Finally, we demonstrate the effectiveness of the locality-driven data placement and migration scheme through the parallel Protein Sequence Matching service (PSM) (as described in Section 4.2.2). In this experiment, the total dataset consists of 24 partitions, each of which is between 1GB and 1.5GB. Each PSM service process is statically assigned a disjoint set of three partitions. To serve a request, a PSM service process performs a local search on its assigned partitions, then sends the results to a dedicated aggregation node. Because of the data access locality exhibited by this service, it is desirable to place the partitions and their designated PSM service processes on the same machine. We collect the traces in an environment where partitions are manually placed with their designated PSM service processes. The traces also contain boundary marks of individual queries.

When playing back the traces in Sorrento, we import the partitions to an eight-node volume without the knowledge of which partitions will be accessed by which PSM service process, so as to verify whether Sorrento can automatically detect the access locality from the workload and migrate partitions accordingly. Each trace replayer issues I/O requests belonging to the same query as fast as it can, and then blocks itself for the same amount of time as indicated by the gap between the query-end mark and the next query-start mark. We show the time-varying behavior of the system by reporting the I/O portion of the service time.

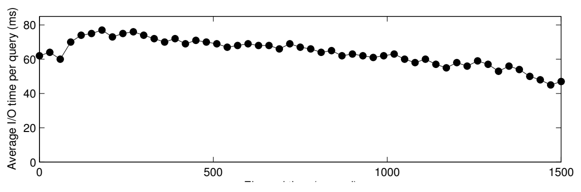

Figure 15: Locality-driven data placement and migration.

The results are shown in Figure 15. The x-axis is the elapsed time, and the y-axis is the I/O portion of the service time per query, which is termed as *I/O time* for simplicity. Each data point in Figure 15 is the average I/O time for queries served in the past 30 seconds. Initially, only four partitions are placed locally with their designated PSM service processes, and the I/O time is 62 ms/query. The data migration starts at time 60, and the background migration traffic increases the I/O time to around 75 ms/query. The I/O time gradually decreases when more partitions are migrated to co-locate with their designated PSM service processes. The migration process finally completes at around 1410, after which the I/O time is reduced 46 ms/query. As we can see, the system is able to dynamically migrate data to their processors without any interruption of services. Additionally, the results also demonstrate the importance of exploiting data locality in the parallel Protein Sequence Matching service, which can reduce the I/O portion of the execution time by as much as 26%.

Summary of findings: (1) Sorrento outperforms PVFS for small file I/O performance, however, it still incurs higher overhead than NFS, which has been optimized for such kind of workload and does not have the overhead of distributed storage management. (2) For large file I/O performance, Sorrento delivers scalable performance and is comparable with PVFS using both microbenchmarks and real application traces. (3) Sorrento is able to automatically handle failures and recover lost data segments. (4) Through load-aware or locality-driven data placement and migration, Sorrento is able to balance storage usage, and improve data access performance without explicit interventions from applications or administrators, and does not interrupt the normal operation of the system.

5 Related Work

Our work is in large part motivated by previous work on parallel/distributed file systems and cluster-based storage systems such as AFS [26], GPFS [37], Petal [30], PVFS [13], Slice [7], Swift [12], xFS [8], and others [22, 23, 28, 32].

Our work complements those work with a specific focus on *incremental expandability* and *manageability* of the system. For instance, Petal and xFS organizes storage in RAID volumes, which is not easy to expand incrementally. PVFS focuses on parallel I/O support and but is notably weak in supporting system expansion and automatically handling component failures. Additionally, Sorrento targets the typical workload of data-intensive applications, and exploits their unique characteristics to optimize their performance.

The design objectives of GoogleFS [18] is the closest to ours. However, GoogleFS focuses on building a proprietary system for their own search engine, and makes several assumptions that may not be valid for other types of applications, such as the lack of directory listing, and the centralized metadata server.

Dynamic distributed data location has been studied peer-to-peer network research, such as CAN [35], Chord in CFS [15, 40], and Tapestry in OceanStore [24, 29]. Our solution is simplified by the fact that we only need to deal with the LAN environment and we use consistent hashing [27] to map SegIDs to home hosts.

Version-based data consistency model and immutable files are first proposed in Amoeba [32]. Other version-based standalone file systems include Elephant [36] and CVFS [39]. Their goals are to protect data loss or to provide information for intrusion analysis, and therefore are different from ours. 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.

Online data placement to balance system workload have also been studied in CFS [15], by Honicky et. al [25] and by Brinkmann et. al [11]. These work all seek to place data proportional to statically determined *weights* of the storage nodes. They can not balance both storage usage and I/O workload. On the contrary, Sorrento uses realtime information of storage usage and I/O workload to make placement decisions.

Dynamic data migration has also been proposed in GoogleFS [18]. Different from their work, the data migration decision in Sorrento is made in a distributed fashion instead of by a centralized server. Additionally, their work mainly focuses on balancing storage usage, while we seek to balance both I/O and storage load.

Finally, our design fits well to the Object-based Storage Device (OSD) Model [6], in which each physical device exports a set of variable-length objects addressed by location-independent OIDs.

6 Concluding Remarks

This paper presents the design and implementation of a storage cluster built upon commodity components called Sorrento. The design of Sorrento targets a class of data-intensive applications that have low write-sharing patterns, and seek to provide incremental expandability and easy manageability by making the system self-organizing. The design of Sorrento employs a version-based data consistency model, lazy propagation for replica updates, and a flexible data placement and migration scheme that is able to balance storage usage and I/O workload, or to exploit data access locality. We validated our design through a combination of microbenchmarks and trace-replay of real applications on a Sorrento prototype. In terms of system performance, we found that Sorrento is able to deliver reasonable performance for small file operations and scalable I/O for large file operations which is comparable with PVFS. Additionally, the self-organizing features of Sorrento simplify system failure handling, and delivers improved storage usage and I/O performance without any human intervention or interrupting the normal operation of the system.

References

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

[2] Google. URL `http://www.google.com/`.

[3] ROMIO: A high-performance, portable MPI-IO implementation. URL `http://www.mcs.anl.gov/romio`.

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

[5] 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.

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

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

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

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

[10] S. A. Brandt, L. Xue, E. L. Miller, and D. D. E. Long. Efficient metadata management in large distributed file systems. In *Proc. of the 20th IEEE / 11th NASA Goddard Conf. on Mass Storage Systems and Technologies*, April 2003.

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

[12] 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.

[13] 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.

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

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

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

[17] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP*, 1997.

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

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

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

[21] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.

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

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

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

[25] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *IPDPS*, 2003.

[26] 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.

[27] 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.

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

[29] 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.

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

[31] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *TOCS*, 2(3), 1984.

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

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

[34] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick. Intelligent RAM (IRAM): The industrial setting, applications, and architectures. In *ISCA*, 1997.

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

[36] 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.

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

[38] 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.

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

[40] 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.

[41] 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.

[42] R. Thakur, W. Gropp, and E. Lusk. An abstrat-device interface for implementing portable parallel-I/O interfaces. In *Proc. of the 6th Symp. on the Frontiers of Massively Parallel Computation*, 1996.

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

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

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

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