An Efficient Data Location Protocol for Self-organizing Storage Clusters

Hong Tang and Tao Yang *
Department of Computer Science
University of California, Santa Barbara, CA 93106
{htang, tyang}@cs.ucsb.edu

ABSTRACT

Component additions and failures are common for large-scale storage clusters in production environments. To improve availability and manageability, we investigate and compare data location schemes for a large self-organizing storage cluster that can quickly adapt to the additions or departures of storage nodes. We further present an efficient location scheme that differentiates between small and large file blocks for reduced management overhead compared to uniform strategies. In our protocol, small blocks, which are typically in large quantities, are placed through consistent hashing. Large blocks, much fewer in practice, are placed through a usage-based policy, and their locations are tracked by Bloom filters. The proposed scheme results in improved storage utilization even with non-uniform cluster nodes. To achieve high scalability and fault resilience, this protocol is fully distributed, relies only on soft states, and supports data replication. We demonstrate the effectiveness and efficiency of this protocol through trace-driven simulation.

1. INTRODUCTION

High-performance cluster-based storage is a critical component for many large-scale data-intensive applications [2, 10, 12, 13, 14, 26, 37]. As scalable cluster-based storage becomes more widely used in applications, availability and manageability become important for a production system. Since application demand for storage capacity constantly grows as faster computers with larger memory at lower cost become available for more advanced computing, it is desirable to expand a storage cluster incrementally over time [34]. On the other hand, node departures in a production storage cluster occur periodically due to node failures or scheduled maintenance. For example, at Ask Jeeves [1], a storage cluster with hundreds of nodes is used to index billions of web pages continuously and disk additions or failures can happen as often as once every few days. It is desirable to have a storage cluster with sustained performance and availability during storage node additions and departures. We call a storage cluster that can adapt to the changes of cluster nodes as *self-organizing*.

*Also affiliated with Ask Jeeves/Teoma.

SC '03, November 15–21, 2003, Phoenix, Arizona, USA

Previous work on cluster-based storages mainly focuses on virtualizing distributed storage resources as a single disk image and delivering high I/O performance [10, 27, 37, 42]. They typically assume that the set of cluster nodes is relatively stable and the mapping from file data (or metadata) to physical nodes are either fixed or through modulus-based hashing. Thus a change in the set of cluster nodes would require a reconfiguration of the storage system and data need to be reassigned to new locations, which adversely affects the availability and manageability of the system.

Adaptive data placement has been extensively studied in the peer-to-peer (P2P) research community [11, 21, 35, 39]. In principle, these schemes could be applied for a cluster-based storage system for high availability. However, a local-area network (LAN) has many different characteristics from a wide-area network (WAN) and it is important to exploit the high-speed nature of a local-area network for developing high performance self-organizing storage clusters. For instance, in Chord, each node maintains a finger table of size $O(\log N)$ and a data location request requires $O(\log N)$ network communications, where N is the number of member nodes of the P2P network. In a LAN, N is typically small and it is affordable to maintain a global picture of the set of nodes, which then could reduce the number of network communications to one.

To accommodate the fact that storage resources and data locations are dynamically changing, this paper studies a flexible data location scheme such that the address mapping from data to storage nodes is adaptive to the change of storage nodes. Our protocol can be used in existing cluster-based parallel file systems such as PVFS [10] to extend their capability to handle dynamic cluster environments. Our protocol design, inspired by previous research projects on distributed data location protocols, specifically targets the cluster environment.

The primary contribution of this work is that it takes advantage of the fact that in a file system, most of the files are small, while most of the space is taken by a small fraction of large files to reduce the protocol management overhead. When using variable-sized blocks to organize file data, the size distribution of blocks also exhibits the same pattern, namely, most blocks are small while most space is taken by large blocks. Our protocol differentiates between small and large blocks as follows: The locations of small blocks are chosen by consistent hashing [23]. Large blocks are placed based on a usage-based mechanism or any method that exploits I/O parallelism or data locality, and Bloom filters [7] are used to keep track of their locations. The proposed scheme improves storage utilization even with non-uniform cluster nodes. This protocol is fully distributed, relies only on soft states, and supports data replication. Through experiments, we show that the differentiated protocol design is much more efficient than uniform strategies in three aspects: access efficiency, storage utilization, and management overhead.

The rest of the paper is organized as follows: Section 2 describes our targeted architecture and the design goals. Section 3 compares various data location schemes. Section 4 presents our differentiated data location protocol design. We evaluate the performance of our protocol design in Section 5 through trace-driven simulations. Section 6 concludes the paper.

2. ARCHITECTURE ASSUMPTIONS AND DESIGN OBJECTIVES

In this paper, we mainly target a generic storage cluster where a number of commodity PCs or workstations are connected by a high-bandwidth low-latency network. Each node in the cluster has storage devices attached to it. We follow three architecture features proposed in the previous work:

- **Share-nothing architecture.** Storage devices are only accessible from the hosts to which they are attached. This differs from the *shared-disk* architecture adopted by GPFS [37] or Snappy [27] where each storage device may be accessed from multiple nodes. The shared-disk architecture requires special hardware support while one of our design principles is to employ commodity hardware components as much as possible so that the cost-effectiveness of the cluster architecture will not be compromised[1]. Systems based on the share-nothing architecture include Petal/Frangipani [26, 40] and PVFS [10].

- **Functionally symmetric nodes.** There are no explicit functional distinctions among cluster nodes. All nodes can carry the tasks of computation and storage management. This choice is different from systems such as Slice [4] or Coda [24] where there is a dedicated set of nodes acting as servers or storage suppliers. The concept of a functionally symmetric architecture is not new and is similar to the *serverless* design of xFS [5] in spirit. A functionally symmetric design can simplify administration tasks, smooth system expansion, and is more resilient to failures. We want to emphasize that functional symmetry does not mean architectural uniformity. Our system design targets a typical cluster environment where it is common to have nodes with different processing power or storage capacity.

- **Resource virtualization.** To make a cluster-based file system easy to use and manage, it is important to have a uniform namespace and virtualize distributed storage resources as a single disk volume. This goal has been the main design objective of a number of parallel or cluster-based file systems such as GPFS [37] and PVFS [10]. On the other hand, distributed file systems such as NFS [3] and Coda [24] do not provide storage virtualization because a file cannot span multiple servers (however, they do export a uniform namespace through the UNIX mount mechanism).

The focus of our research in the above context is to deal with the changes of storage cluster configuration while providing high availability. There are always changes in a storage cluster over time: existing machines may fail or exhausts its usage lifespan, and new machines can join the cluster as the demand for storage capacity increases. We envision a storage system must have 24x7 availability to the applications; namely, the configuration change in the system should have little or no impact to the system availability. Previous research on distributed file systems or storage systems [4, 5, 10, 19, 20, 26, 37, 42] mainly assume that the set of nodes supplying storage resources is relatively stable, and the mapping from file data (or metadata) to physical nodes are either fixed or determined through modulus-based hashing (details of some of these data location schemes will be discussed in Section 3.). Typically, a change in the storage node membership would require system reconfiguration and data migration, which interrupts the availability of the system. The goal of *self-organizing* is to let the system automatically adapt to the changes of the cluster environment. When a new node joins the cluster, its storage resource will be automatically "merged" into the virtualized storage space; when a node leaves the cluster, its storage resource will automatically be excluded from the virtualized storage space.

The following optimization objectives are considered in our design:

- **Data access performance and reliability.** Performance is always the overriding concern for a cluster-based system supporting today's I/O-intensive applications. Data access performance can be further broken down into two parts: data location performance and data transfer performance. We focus on optimizing data location performance for fast lookup and placement in a large-scale cluster. Reliability is also an important aspect to be considered. A centralized server with fast lookup is not acceptable because it is a potential single-point of failure.

- **Storage utilization.** The system should be able to effectively utilize the available storage space. As node additions and departures occur, the distribution of file blocks in the cluster can become uneven and some space may not be usable due to the restriction of mapping method, as will be shown in later sections.

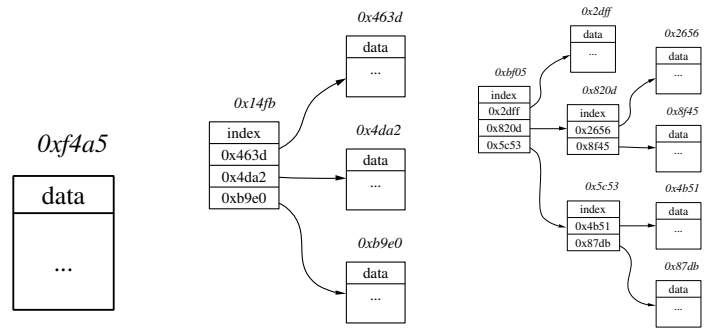
- **Maintenance overhead.** As the cluster configuration changes, the metadata for the location information may also need to change. Additionally, data migration may be necessary to maintain the data locations in the cluster in a consistent state. During this process, the system may consume various resources to maintain the consistency of state information and it is desirable to keep such maintenance cost low.

3. DESIGN CHOICES AND A COMPARISON OF DATA LOCATION SCHEMES

3.1 Problem Statement

We consider that the physical data of a user-perceived file are divided into blocks and are organized as a tree. Non-leaf blocks (called *index blocks*) contain pointers to other blocks, and the actual file data are stored in leaf blocks (called *data blocks*). Each block is stored in its entirety on a cluster node. We use a unified address space to address these blocks. The address space can be set to be large enough (128-bit integers) that block addresses (called *BlockIDs*) can be generated in a distributed fashion without worrying about collisions. To support files with a wide range of sizes, from several kilobytes to several terabytes, the sizes of blocks are typically not fixed [10, 15, 37, 43]. Figure 1 illustrates several examples of how file data are organized as a tree of blocks: Small files may be represented by a direct data block (Figure 1(a)). Large files may have one index block and a set of data blocks (Figure 1(b)). Very large files may require multiple levels of indexing

[1]This argument does not imply that shared-disk architecture is a bad design. In fact, shared-disk architecture can be the choice of design where very high performance and availability is required.



(a) A small file represented by a data block.

(b) A large file with an index block and a set of data blocks.

(c) A very large file with multiple levels of index blocks.

Figure 1: Tree-based data organization. The top slot of each block marks whether it is a *data* block or *index* block. In this example, BlockIDs are 16 bits integers.

(Figure 1(c)). Such a structure is quite general, for instance, the data organization schemes used by OceanStore [36], PVFS [10], or PIOFS [15] can all be considered as special cases of such a tree-based data organization scheme. An additional advantage of such a scheme is that each file can be uniquely identified by the BlockID of the root block (called *FileID*).

Note that file system namespace management is orthogonal to data organization. For example, we can have a dedicated namespace server that maps human understandable pathnames to the corresponding FileID. We can alternatively implement file system directories as special files, which further contain names and FileIDs of the files or sub-directories stored in it (then all we need to know is the FileID of the root directory). Scalable and reliable namespace management has been studied in various research projects [4, 8, 37], and is not the focus of this paper[2].

The problem of data location is to derive and maintain the mapping function $\mathcal{F}$ from the block address space $\mathcal{B}$ to the set of cluster hosts $\mathcal{H}$. Given a new data block, we need to find a host (or several hosts under a replication scheme) to host this block, and add the mapping between the BlockID to the host ID (or host IDs). When accessing a data block, we need to retrieve the host (or hosts) that maintains the block. There are a number of issues that need to be considered in designing a data location scheme:

- **How are data mapped to hosts?** There are a number of objectives driving different data mapping strategies. In the previous research, striping or related methods are studied for exploiting parallel I/O [10, 15, 37]. Copies of data can be mapped to multiple locations for better fault tolerance. Balancing space usage among nodes is another strategy so that data blocks are evenly distributed among machines, which allows a higher degree of I/O parallelism. Chang et al. [12] further proposed to cluster relevant data together on a same host to reduce disk seek time.

- **How is the mapping information maintained?** If data are placed on hosts without any restriction, an explicit table for block mapping needs to be maintained. Additionally, the mapping information needs to be made available to any node that needs to access blocks on behalf of an application.

  If the mapping is computable through some pre-defined function, an explicit table for block mapping is not necessary. In this case, there will be no cost for maintaining the consistency of the mapping information across nodes. However, such a function may increase the maintenance overhead in the event of node additions or departures, and lead to imbalance in data distribution.

- **What is the impact of node additions or departures on the mapping table maintenance?** When nodes join or leave the system, the mapping table may need to be updated. If the change of cluster size affects data mapping, data blocks need to be migrated among nodes in order to keep the mapping consistent. When the table is replicated, any changes to the mapping table need to be applied to all the replicas.

  When dealing with a large storage cluster hosting terabytes of data, the migration cost should be minimized. Otherwise, moving a large amount of data would reduce the availability of such a cluster, affect the system performance, and increase operation challenge if human intervention is required to fix some unexpected problems.

3.2 Data Location Schemes

Data location has been extensively studied in previous research on distributed and parallel file systems. Additionally, recent research on peer-to-peer network also proposed various interesting data location schemes. In the following sections, we present a comparison of these schemes which are used with some modification in our context. Throughout the following discussion, we let N be the number of physical blocks in the system, H be the number of hosts in the cluster ($N \gg H$), and S be the total volume of stored data.

We categorize data location schemes into two types based on whether the mapping information is *replicated* or *partitioned* among cluster nodes.

3.2.1 Replicated Mapping Table and Bloom Filter

A simple approach is to implement $\mathcal{F}$ as a globally replicated mapping table [27]. Each entry in the table specifies a ⟨BlockID, Host⟩ pair. The space requirement for this table would be $O(N)$,

[2]However, we did implement a namespace server for practical usage in this project.

and the lookup cost is $O(\log N)$ (suppose the table entries are sorted by BlockIDs). The main problem of this approach is that the size of the mapping table could be very large in supporting a large-scale storage cluster with billions of file blocks. A back-of-the-envelope calculation shows that it would take as much as 2GB memory to store such a table with one hundred million entries.

An improved approach is to "compress" the mapping table using Bloom filters [7]. Bloom filters have been used in Summary Cache [18] and OceanStore [25]. A Bloom filter is a compact data structure to encode a set of keys (in our case BlockIDs) in a bit array and can be used to check whether a given key is in that set with a small percentage of false positive.

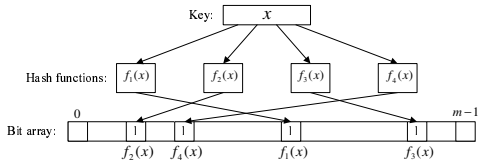


Figure 2: Adding a key to a Bloom filter using four hash functions.

To encode a set $B = \{b_1, b_2, ..., b_n\}$ of n keys into an m-bit Bloom filter, we first initialize all bits to 0. We then use a group of j independent hash functions $\{f_1, f_2, ..., f_j\}$ to set some bits in the bit array (f_i hashes a key to an integer between 0 and $m-1$). Specifically, for each key b_i, we set the bits $f_1(b_i)$, $f_2(b_i)$, ..., and $f_j(b_i)$. Figure 2 illustrates how we add a key into a Bloom filter using four hash functions. To check whether a key x is in B, we simply test whether the bits $f_1(x)$, $f_2(x)$, ..., and $f_j(x)$ are all set. If any of those bits are not set, we are sure that x is not in B. However, it does introduce a small percentage of false positive, i.e., when all those bits are set, there is a small chance that x may not be in B. Sacrificing a small amount of certainty, Bloom filters can reduce the space requirement by at least one order of magnitude. For example, it is possible to use only 150MB memory to encode one hundred million BlockIDs with a false positive rate less than 0.4% using six hash functions. In practice, j is typically statically determined to limit the false positive rate, and m is dynamically adjusted to maintain the percentage of set bits to be around 50%.

To manage data location information using Bloom filters, each host maintains a Bloom filter that encodes all the blocks stored locally. It also broadcasts updates of the Bloom filter to all other hosts on the cluster. So each host keeps $H-1$ additional Bloom filters, one for every other host. To lookup a BlockID b, we iterate through all the Bloom filters and perform the membership test. If a test of b on host h_i's Bloom filter is positive, then b can be accessed on h_i with high probability. False positives will be discovered eventually when we try to access the block from a host that does not have it. The space requirement for Bloom filters is still $O(N)$, but the constant factor is much smaller than the replicated mapping table case. The lookup cost is $O(H)$.

3.2.2 Partitioned Mapping Table

A Bloom filter table could still be large and in order to avoid the scalability problem, we can partition the mapping information so that each host only manages a portion of the mapping information. Basically, we designate a host for each BlockID which is responsible for resolving the location of that block. This host is called the *home host* of that block. The host that actually stores a block is called the *owner* of the block. A lookup operation first determines which host is the home host of a BlockID, and then finds the owner of the block through the home host. Thus each lookup involves both computation and communication.

An extension of the naive approach described previously is to use a global redirection table to determine the home host of a block. The size of the redirection table is in the order of H, and each entry contains a host ID. Given a BlockID b, we take a modulus operation with the table size as the base and use the result to index into the redirection table. The host stored in the corresponding entry will be the home host of block b. Variations of this scheme has actually been adopted by Petal [26], xFS [5] and Slice [4]. The space requirement on each host is now $O\left(\frac{N}{H}+H\right)$, and the lookup cost is $O(\log N - \log H) \approx O(\log N)$ plus one network operation. One drawback of this scheme is that whenever a node joins or leaves the network, the global redirection table needs to be modified consistently to reflect the change. In previous research, this typically requires the system to enter a recovery mode and could result in a disruption of the service [5, 26].

Various distributed hash table (DHT) schemes for P2P systems such as Chord [39], Tapestry [21], and Pastry [11] can also be regarded as data location schemes where mapping information is partitioned among hosts. Because those schemes must work over a WAN, it is not possible to maintain the replication consistency of a global redirection table. Instead, they use some *distance metrics* to establish the distance between a key (or BlockID in our case) and a host, and designate the host that is the closest to a certain BlockID to be the home host of that block. Each node keeps a routing table with a small number of entries that allows it to forward a lookup request to a host that is closer to the BlockID than itself. The routing process guarantees it takes at most $O(\log H)$ network hops to locate the home host of a certain BlockID. At the home host, it again takes $O(\log N - \log H) \approx O(\log N)$ CPU cycles to locate a block's owner. The total space requirement is $O\left(\frac{N}{H}\right)$. An important advantage of these DHT schemes over the partitioned mapping table scheme is that they can automatically adapt to node additions or departures without any special reconfiguration.

Given the fact that it is relatively easy to maintain a consistent view of the set of hosts on a LAN, we can improve the lookup efficiency by reducing the number of network operations. For instance, we can modify the Chord scheme by changing the routing table (called the *finger table* in the original paper [39]) such that it maintains all the hosts in the cluster, then we can determine a BlockID's home host directly and thus reduce the total number of network operations to just one.

When the set of hosts changes, the partition of BlockIDs among hosts could also change. When a BlockID's home host changes, its original home host needs to forward the BlockID's owner address to the new home host. Suppose the mapping information is evenly partitioned among hosts, the bandwidth requirement would be $O\left(\frac{N}{H}\right)$ when a host joins or leaves the cluster.

One complication in the partitioned approach is that we still need to use replication to improve the availability of each mapping table partition in addition to the fault tolerance issue for data blocks.

3.2.3 Controllable versus Uncontrollable Data Placement

The above schemes all assume that physical blocks are placed in a controllable fashion. For example, a block is placed following the parallel I/O principle using striping or to balance the space usage. The storage system designer or application users can decide data placement policies based on their needs. As a result, the mapping information has to be maintained in a table.

If the requirement of controlled placement is relaxed, and if we can compute the location of a block directly using a function, then the mapping table is not needed, which eliminates the need to store mapping information explicitly and avoids consistency issues in

Scheme	Space	CPU-cost	Net-ops	Mig-overhead	Weakness
GMAP	$O(N)$	$O(\log N)$	0	0	High space overhead.
BLOOM	$O(N)$	$O(H)$	0	0	Space requirement may still be too high.
PMAP	$O(N/H)$	$O(\log N)$	1	$O(N/H)$	Cannot automatically adapt to cluster member changes.
P2P-DHT	$O(N/H)$	$O(\log N)$	$O(\log H)$	$O(N/H)$	High network communication overhead.
Chord-LAN	$O(N/H)$	$O(\log N)$	1	$O(N/H)$	Still requires network communication.
DCH	$O(H)$	$O(\log H)$	0	$O(S/H)$	Uncontrollable placement.
LH*	$O(1)$	$O(1)$	0	$O(S/H)$	Uncontrollable placement; centralized coordinator.

Figure 3: A comparison of data location schemes. A list of notations: N – number of blocks; H – number of hosts (we assume $\frac{N}{H} \gg H$); S – total volume of stored data; GMAP – replicated mapping table; BLOOM – Bloom filters; PMAP – partitioned mapping table; P2P-DHT – distributed hash table schemes proposed by P2P community; Chord-LAN – modified Chord with complete host table; DCH – direct placement using consistent hashing; LH* – distributed linear hashing. Note the column "Mig-overhead" means the additional bandwidth requirement to migrate data when a host joins or leaves the cluster.

maintaining such a table across multiple nodes. To achieve this computable mapping, we can use consistent hashing [23] (referred to as *DCH* in this paper) or distributed linear hashing LH* [28, 29, 30]. The basic idea is to let the home host of a certain BlockID store the actual physical block instead of a pointer to the owner of the block. The space requirement is much smaller, ranging from a constant for LH* to $O(H)$ for DCH, and data lookup only involves computation which varies from a constant for LH* to $O(\log H)$ for DCH. Although LH* seems more efficient than DCH, it actually requires a centralized coordinator that controls the update of two integer counters. The coordinator does not become a threat toward the system scalability because the work involved is fairly light-weight (incrementing the counters or reporting the current values of the counters); however, it is a potential single-point-of-failure.

The price we pay for computable mapping is that data placement becomes *uncontrollable*, because a hashing function does not consider if there is space available for hosting a block or not. Thus a certain amount of free space must always be reserved to keep such a scheme working, which leads to wasted space. Several techniques are proposed to improve storage utilization [9, 16, 22, 23, 29]. However, it remains a hard problem to maintain high storage utilization, especially in a heterogeneous environment where cluster nodes are added with different capacities.

It should be noted that for uncontrollable data placement, when the set of hosts changes, we need to migrate the physical blocks instead of pointers. The bandwidth consumption for migrating the blocks is $O\left(\frac{S}{H}\right)$ for DCH when a host joins or leaves the cluster [23]. If a modulus-based hashing method is used, almost all data blocks need to be migrated when the number of hosts changes.

3.2.4 A Summary of the Comparison

Figure 3 gives a comparative summary of the data location schemes described above. For each scheme, we also show its weakness compared to other schemes. The schemes are listed in the descending order of their space requirement, which reflects the memory consumption. For all these schemes, the CPU requirement is quite low. In essence, these schemes can be divided into three different classes: (1) Unscalable memory requirement but fast access speed (GMAP and BLOOM); (2) Scalable memory requirement but low access speed (PMAP, P2P-DHT and Chord-LAN); (3) Very low memory requirement and high access speed, but uncontrollable data placement (DCH and LH*).

4. DIFFLOC: A DIFFERENTIATED DATA LOCATION PROTOCOL

Each scheme described in Section 3.2 has strengths and weaknesses. Our solution is to combine multiple schemes together and dynamically decide which scheme should be used to locate a certain block. Specifically, we place and locate small file blocks through DCH, and we place large file blocks based on storage availability and locate them through Bloom filters. In this section, we first present the motivation behind such a design, followed by a description of the main modules of the protocol. We conclude this section with a discussion of various subtle details of the protocol design.

4.1 Motivation

The idea of differentiated treatment of small and large file blocks is motivated by previous studies by Baker et al. [6] and Vogels [41], in which they found that large files and small files exhibit different characteristics in terms of user activities, access patterns, and their life time. Particularly, in terms of file size distribution, it turns out that most operations are on small files while most bytes transferred are for large files. This could be translated into the statement that the majority of files in a file system are small files, but the majority of disk space is consumed by large files. Additionally, small files are created and deleted more frequently than large files. Both studies are based on typical *interactive* usage of file systems. However, we believe the conclusions still hold for *non-interactive* workload. To confirm our belief, we studied the file size distributions for three systems under non-interactive usage: (1) Storage for offline processing of a commercial search engine at Ask Jeeves/Teoma (such as page crawling and indexing) (**Service-offline**); (2) Storage for online database generation and data aggregation of the same company (**Service-online**). (3) A backup archive server for a research lab at UCSB (**Group-archive**). Service-offline and Service-online consist of disks scattered on multiple hosts, and Group-archive consists of three disks attached to the same server.

	Service-offline	Service-online	Group-archive
Total files	489,601	34,235	301,448
Total size (MB)	7,167,382	1,978,640	152,833
%-files using 90%-space	1.3	6.9	7.1
%-files using 95%-space	4.3	8.1	16.1
%-files using 98%-space	9.1	8.8	23.3
Knee-point(%-files,%-space)	(3.1,94.0)	(9.5,99.7)	(5.5,88.8)

Figure 4: The statistics and characteristics of three file systems under non-interactive workload.

The sizes and numbers of files in these three systems are shown in Figure 4. The file size distributions for these systems are shown in Figure 5. The x-axis in Figure 5 is the percentage of total files. The y-axis is the cumulative size of the largest x percent files, expressed as a percentage of the total data volume. To make the re-

sults easier to comprehend, Figure 4 further shows the *knee points*[3] on the file size distribution curves, as well as the percentages of files accounting for 90%, 95% and 98% of the total data volume. As we can see, the file size distribution is heavily skewed in all three systems. Additionally, for Service-online and Service-offline, which are used by data-intensive components in an Internet search engine, more than 90% of the files take up only 2% of the total space.

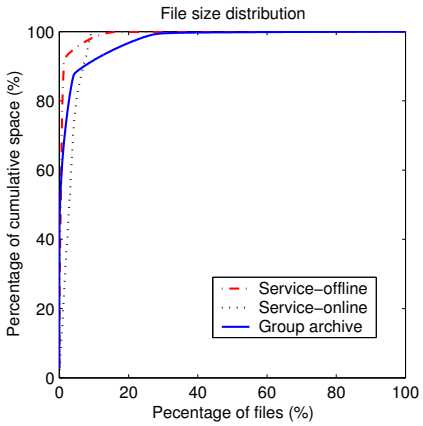


Figure 5: Cumulative file size distribution for three file systems under non-interactive workload. We sort files based on their sizes in descending order. x-axis is the percentage of total files; y-axis is the cumulative file size normalized as a percentage of the total data volume. A point (x, y) on a curve means that the largest $x\%$ files account for $y\%$ of the total space taken.

It should be noted that we use the number of files and their sizes as units to illustrate that storage space is dominated by a small number of large files even though there are a large number of small files. However, since we typically choose large block sizes for large files and small block sizes for small files when organizing file data in a storage cluster, it is reasonable to infer that the capacity of a large storage system is usually dominated by a relatively small number of large data blocks even though there are a large number of small data blocks.

Based on this observation, we seek to use different data location schemes to manage blocks with different sizes. First, we can see that if we use DCH to manage only small blocks, then the amount of wasted space that must be reserved to accommodate the uncontrolled placement can be significantly reduced because small blocks only take a small percentage of total storage space. For the same reason, the amount of data that needs to be migrated in the event of host additions or departures is also reduced. Second, if we only use BLOOM to manage large blocks, the amount of memory required by the Bloom filters can also be significantly reduced because the number of large blocks is only a fraction of total blocks. In addition, all lookup requests can still be served without communicating with other hosts. Figure 6 compares this differentiated data location scheme (which we call *Diffloc*) with DCH and BLOOM in terms of the optimization objectives specified in Section 2. As we can see, Diffloc enjoys the efficiency of both BLOOM and DCH while avoiding either scheme's weaknesses.

4.2 Protocol Design

The Diffloc protocol does not require centralized control. On each participating cluster node there is a Diffloc daemon that contains three cooperative modules: (1) protocol state management; (2) data placement and location request handling; (3) local storage management.

[3]A knee-point is defined as the point on a curve where the gradient is 1.0.

	DCH	BLOOM	Diffloc
Storage utilization	−	+	+
Migration overhead	−	+	+
Memory consumption	+	−	+
Access efficiency	+	+	+

Figure 6: A qualitative comparison of Diffloc with DCH and BLOOM. We define *goodness* as high storage utilization, low management overhead, or high access efficiency. We rank the schemes using marks of "+" (good), "−" (bad), or "." (fair, which happens not to be used).

4.2.1 Protocol State Management

Each host maintains two data structures: a set of live hosts (for DCH) and an array of Bloom filters (for BLOOM). All hosts subscribe to the same multicast channel, through which they periodically send their local states and receive state updates from remote hosts. The state information includes the host's current load condition, the amount of available storage space, and the Bloom filter that encodes its locally stored large blocks. A host learns about the set of live hosts simply by monitoring the state update packets (called *heartbeat* packets). The absence of heartbeat packets from a certain host for a prolonged period (e.g., ten times the average update interval) will prompt other hosts to remove that host from the live set.

It is not feasible to send out the whole Bloom filter every time a host announces its states. The Diffloc daemon keeps track of what bits have been set or cleared during the past announcement interval and only sends out an incremental update of the Bloom filter. This approach has also been suggested by Summary Cache [18].

4.2.2 Requests Handling for Data Placement and Location

The Diffloc daemon is responsible for handling all data placement or location requests originated from applications running locally. To adaptively switch between the DCH and BLOOM protocols, an important question is how to determine whether a block is a small block or a large block. As can be seen in Figure 4, the average block sizes for different applications may be different and there is no universally applicable threshold size to distinguish these two categories of blocks. Ideally, we would like to choose the threshold size around the knee-point. However, the knee-point cannot be determined until a file system is populated with blocks. On the other hand, maintaining a global threshold among all hosts requires us to run a consensus protocol every time this threshold is changed, which could limit the system's scalability and can fail during network partitions.

In this paper, we let each host maintain a per-host threshold size and periodically announce it through the multicast channel. The adjustment of the threshold is based on the size distribution of blocks stored on a host and is part of the local storage management, which will be discussed in Section 4.2.3.

Figure 7 shows how the Diffloc daemon places (`place_block`) and locates (`lookup_block`) blocks based on the per-host threshold sizes. Routine `DCH_hash` finds a BlockID's closest host according to consistent hashing. Routine `BLOOM_lookup` returns the host whose Bloom filter encodes a certain BlockID (Strictly speak-

```
Host place_block (Block b)
{
  // Should b be placed using DCH?
  Host h = DCH_hash(hosts, b.bid);
  if ( h.threshold >= b.size ) {
    // Yes (because b's size is below h's threshold).
    return h;
  }
  else { // No.
    // Find candidate hosts that can store the block.
    set<Host> candidates = find h in hosts
            where (h.threshold < b.size);
    // Select one host from the candidates
    // based on some placement policy.
    h = Place_policy(candidates, b);
    return h;
  }
}

Host locate_block (Block b)
{
  Host h = DCH_hash(hosts, b.bid); // Is b managed by DCH?
  if ( h.threshold >= b.size ) {
    // Yes (because b's size is below h's threshold).
    return h;
  }
  else { // No.
    // Re-do the lookup using BLOOM.
    h = BLOOM_lookup(b.bid);
    return h;
  }
}
```

Figure 7: Differentiated data placement and location. Routine `place_block` returns a host that is suitable for storing a certain block; routine `lookup_block` returns a host that stores a certain block with high probability.

ing, `BLOOM_lookup` may return an exception when the BlockID is not encoded in any host's Bloom filter; it may also return multiple hosts due to false positives. These conditions happen rarely and we briefly discuss how to handle them in Section 4.3.). Routine `Place_policy` implements a placement policy for large blocks. We discuss the placement policy in Section 4.2.4.

4.2.3 Local Storage Management

The local storage management module is responsible for the creation and deletion of blocks, and the incremental updating of the local Bloom filter accordingly. Specifically, we keep a reference counter for every bit of the Bloom filter. When we add a BlockID to the Bloom filter, we increment the counters for the hashing bits. When we remove a BlockID from the Bloom filter, we decrement the counters for the corresponding hashing bits. A bit will be set when its reference counter changes from 0 to 1; and it will be cleared when the reference counter changes from 1 to 0.

The second part of the storage management module maintains the size distribution of all local blocks and dynamically determine a threshold size from the distribution. The threshold size is used by the data placement and location module to differentiate between small and large blocks, as shown in Figure 7.

Instead of setting the threshold size close to the knee point of the block size distribution, it is actually sufficient to control the storage consumed by blocks below the threshold size. For example, we can limit that all small blocks to consume at most 10% of the local storage capacity.

Our solution is called *Slide Bar* and is illustrated in Figure 8: Initially, when a file system is empty, the block distribution curve (called the *profile line*) is a flat line, and we set the threshold (the vertical bar) at the right end of the profile line (Figure 8(a)). When the file system is populated by more blocks, the profile line rises. To maintain the invariance that all blocks smaller than the threshold (those in the shadow area) are constrained by a certain space limit, the threshold bar needs to slide to the left (Figure 8(b)). The threshold bar continues to slide toward left over time until the file system is fully populated (Figure 8(c)). Every time the bar moves, blocks that have changed side (from the left side of the bar to the right) correspondingly change their location scheme from DCH to BLOOM, and thus need to be added to the local Bloom filter.

The profile line could also decrease when blocks are deleted, however, we *do not* slide the threshold bar back to right. The reason for such a choice is because changing a block's location scheme from BLOOM to DCH could prevent the block from being located unless the block is migrated to a proper location. Although the profile line of a file system may transiently rise and fall when blocks are deleted or created over time; we expect the shape of the profile line to remain relatively stable when a file system is reasonably populated (e.g., 50% full). So the potential danger of shifting the threshold bar to be excessively small and lead to too many large blocks is also expected to be slim. The traces that we collected over a period of 15 days do exhibit a consistent block distribution curve. We plan to study the effectiveness of such a scheme when the system is put into use in a production environment.

4.2.4 Data Placement Policy

For blocks of small files and index blocks (which are typically small), we use DCH to map them directly to different nodes. For large files, parallel I/O is feasible and we may use various methods to place their blocks. For example, data placement may seek to balance storage usage, I/O workload, or exploit locality by clustering related data items [12]. Our scheme can support arbitrary policies and the result of placement is stored in the mapping table. Its summary information encoded in the Bloom filter is propagated to all nodes that need the information.

In our evaluation study, we use a usage-based block placement policy which distributes blocks of a file to different nodes and balances space usage among all the nodes. The objective is to improve the overall system storage utilization.

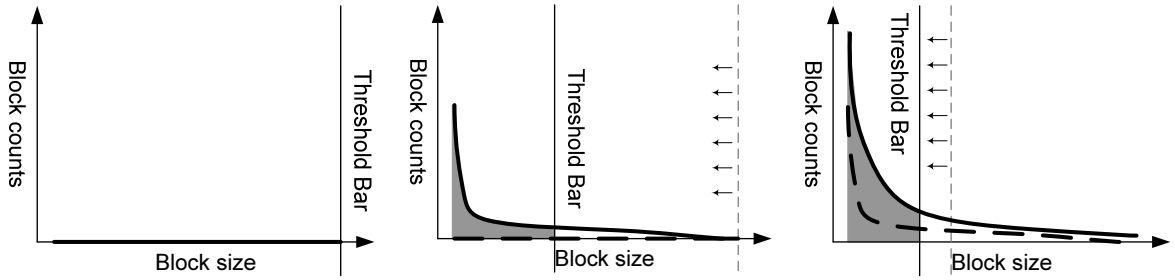
To select a node as the host of a large block, we first filter out nodes that have insufficient space to hold the block. Then, each of the remaining nodes has a probability of being chosen as the owner proportional to its available space. By using a randomized algorithm, we could avoid the so-called *flocking* effect that many blocks are simultaneously placed to the same host.

4.3 Discussions

4.3.1 Handling Node Join

When a node joins the cluster, a portion of small blocks need to be migrated to the new node as their new owner. While the migration is in progress, this newly joined node may "mask" the actual location of a certain block in the sense that the host returned by the `DCH_lookup` operation is different from the block's actual owner. To solve this problem, we retry the `DCH_lookup` operation for k times, and each time we exclude the false owner from the set of hosts. When all k attempts fail, we perform a multicast query.

Figure 9 shows the modified lookup operation which is now part of a `read_block` operation. Retries happen when the actual data access operations fail. To avoid multiple lookup requests for the same block, we also employ a local location cache that maintains the addresses of recently accessed blocks. We chose a cache size of around ten thousand entries, which is sufficient to cache the locations of actively accessed blocks. The probability of a block's location being completely masked by l newly joined server



(a) An empty file system. The threshold bar is at the far right.

(b) File system becomes populated. The threshold bar moves toward left.

(c) File system reaches full. The threshold bar reaches the stable position.

Figure 8: Determining threshold size through the *Slide-Bar* algorithm.

is $\frac{\binom{l}{k}}{\binom{l+H}{k}} (l \geq k)$. Since $H \gg l$, a small value of k can sufficiently reduce the number of multicast queries.

4.3.2 Handling Node Failure and Supporting Replication

A node failure can be detected using a heartbeat and timeout scheme as in Neptune [38]. When a node fails, the block placement scheme will not choose a failed node to store new blocks. The location algorithm will still be able to find blocks on live nodes: for large blocks, the Bloom filters are replicated on all nodes; for small blocks, the DCH scheme does not depend on the number of nodes to locate a block. There is no data migration needed after a node failure.

However, data blocks stored on the failed node *do* get lost. Replication or checkpointing can be used to improve the availability. In some cases, it could be more efficient to rely on application-level checkpointing than block-level replication because replication can slow down system performance during heavy write traffic.

If replication is indeed preferred, our Diffloc scheme can be extended as follows: The replicas of the same block will have the same BlockID. The replication degrees of the root blocks are stored in the namespace server. The replication degrees of other blocks are stored in index blocks. But we *do not* keep the physical locations of the replicas of a certain block. For small blocks, we can extend the DCH algorithm by choosing the closest r hosts to be the homes of a certain BlockID, as suggested in CFS [16]. For large blocks, we can modify the placement policy so that it finds r suitable replica sites, and the Bloom filters will record those replica sites. The protocol can find all replicas of a certain block most of the time (but not always). In the rare occasion when we may miss a replica site during an update operation, we can treat it as if the missing replica site suffers a transient failure, so its blocks are not accessible temporarily. Previous researches have studied the problem of maintaining data consistency across replicas in these situations and their solutions are readily applicable. For instance, we can tag blocks with versions (which increments whenever we update the block), and let different hosts synchronize the versions of their blocks through gossiping or periodic introspection [17, 25, 31].

4.3.3 Adjusting Bloom Filter Sizes

The number of large blocks stored on a node may change over time, we dynamically adjust the size of the local Bloom filter so that the percentage of set bits of the filter to be in the range of $25-60\%$. If the percentage of set bits is lower than 25%, we shrink the size of the filter by half; if the percentage of set bits is higher than 60%, we double the size of the filter.

4.3.4 Data Migration

The goal of data migration is to maintain the invariance that small blocks are stored in their home hosts based on consistent hashing when new node joins the network. To efficiently determine which small blocks need to be migrated upon the detection of a new host on the network, we maintain a database of the BlockIDs of locally stored small blocks and sort them based on their locations on the circular space $[0, 1)$ (called the *Chord ring* in [39]). The calculation of the set of migrating blocks takes $O(\log N)$ CPU cycles. The size of this database is proportional to the number of locally stored objects. Currently we keep the database in memory, however, it can be easily modified to keep the database on persistent storage. Since data migration does not happen very frequently, we expect such a change not impact the overall system performance.

4.3.5 False Positives and False Negatives in Bloom Filters

As mentioned earlier, in rare occasions, Bloom filters may return false positives. This means that the `BLOOM_lookup` call may return multiple hosts and only one of them actually has the block. In this case, we can issue the actual request (say a read request) in parallel to all the hosts, and it will succeed when the host that does own the block returns the data.

`BLOOM_lookup` may also return an empty set, which is also called *false negative*. This could happen in the rare situation when we try to lookup a newly created block whose owner has not announced the update of its Bloom filter yet. In this case, we again resort to a multicast query.

4.3.6 Virtual Hosts

Karger et al. [23] proposed to map each host to a number of virtual hosts in the consistent hashing algorithm to balance the shares of blocks hashed to all the hosts. Under this revised scheme, the blocks hashed to a physical host would be the union of blocks hashed to all its virtual hosts. Dabek et al. [16] further proposed to set the number of virtual hosts for a physical host proportional to the storage capacity of the physical host. In the Diffloc protocol, we also apply the Dabek technique: we set the number of virtual hosts proportional to the space reserved for small blocks.

```
int read_block(Block b, char *buf, int off, int sz)
{
  // Step 1: check the local location cache.
  int rval;
  Host h = location_cache.lookup(b);

  if (h && ((rval=remote_read(h, buf, off, sz))!=NOT_FOUND))
    return rval;

  // Step 2: attempt to retrieve data through DCH.
  // Make a copy of hosts
  set<Host> hosts2 = hosts.clone();
  int max_retry = k;
  bool try_bloom = false;

  while ( max_retry-- > 0 ) {
    // Suppose b is managed by DCH.
    h = DCH_hash(hosts2, b.bid);
    if ( h.threshold >= b.size ) {
      // Found a possible owner
      if ( (rval=remote_read(h, buf, off, sz)) \
          == NOT_FOUND ) {
        // False alarm, remove h from hosts2.
        hosts2.remove(h);
      }
      else {
        location_cache.insert(b, h);
        return rval;
      }
    }
    else {
      try_bloom = true;
      break; // b is managed by Bloom.
    }
  }

  // Step 3: attempt to retrieve data through BLOOM.
  if (try_bloom) {
    h = BLOOM_lookup(b.bid);
    if ((rval=remote_read(h, buf, off, sz))!=NOT_FOUND) {
      location_cache.insert(b, h);
      return rval;
    }
  }

  // Step 4: fall back to fail-safe multicast query.
  h = mcast_lookup(b.bid); // Resort to multicast lookup.
  if ( (rval=remote_read(h, buf, off, sz)) != NOT_FOUND )
    location_cache.insert(b, h);

  return rval;
}
```

Figure 9: `read_block` reads a portion of a block starting from a certain offset.

5. TRACE-DRIVEN EVALUATION

The Diffloc protocol has been fully implemented as part of the Sorrento prototype currently under development at UCSB. The overall objective of the evaluation is to demonstrate the effectiveness of the Diffloc protocol in comparison with uniform strategies. We choose to compare Diffloc with two uniform strategies: (1) DCH and (2) BLOOM with usage-based block placement. We will analyze their storage utilization, data migration overhead, and memory consumption to validate our conclusion.

We also demonstrate the access performance of the Diffloc protocol in terms of block placement and lookup time. We show that our protocol has very low overhead when varying the number of cluster nodes.

Finally, we demonstrate that the Diffloc protocol is able to redistribute workload evenly across hosts in the event of node failures, and that it can efficiently utilize storage resources for hosts with non-uniform capacity.

We use trace-driven simulation as our evaluation method. We describe a trace collection utility in Section 5.1. The simulator is presented in Section 5.2. All simulations are conducted on a cluster of 40 dual Pentium III 1.2GHz Linux PCs connected with a Fast Ethernet switch.

5.1 Trace Collection

Several factors lead us to develop our own trace collection utility:

1. Most of the file system traces available in the public domain are collected in interactive environments such as desktop applications or software development process [6, 33, 41]. However, we are more interested in the file system usage for data-intensive applications.

2. Existing traces do not have sufficient information to allow us to loyally reconstruct the environment where the trace is collected. For instance, the Sprite trace does not have fine-grained timing information for each operation. Additionally, file system requests received from distributed clients are multiplexed on the server side, so the actual application-level access pattern is disguised.

3. Existing traces are often too short or the amount of data volume involved is too small to drive meaningful evaluation for a very large storage system. We are interested in studying the long-term behavior of data intensive jobs involving terabytes of data.

Our trace utility patches the glibc library and intercepts all file system related system calls (a system call issued by a program will actually go into a wrapper function which decodes the parameters and issues the system call interrupt.). We also run a trace collection daemon on the same machine which waits on a domain socket. The tracing of an application can be enabled or disabled through environment variable settings. Applications that are being traced send the information about each file system call's parameters, return values, timing and its own identity to the trace collection daemon. Since over time, a process ID (pid) may be reused by the operating system, it is not reliable to correlate all system calls from the same application based on pid. Therefore, we further use a random generation ID (genid) for each process group. We can reliably correlate activities from the same application based on the same pid and genid combination. To minimize the tracing overhead, the trace daemon stores the trace data on a dedicated disk. We also carefully encode each trace record and compress them before writing to disk, and each record takes only 5 bytes on average. Combined with other techniques such as buffered read/write, we are able to reduce the tracing overhead between 3% and 20%. When the CPU is not the bottleneck and the application does not issue system calls too fast, which would otherwise cause the domain socket to be saturated, the overhead is not noticeable.

To collect meaningful trace data that reflect the real world situation, we run our trace utility in a subset of machines of a production environment at a search engine company (Ask Jeeves/Teoma) where billions of page documents are indexed continuously. We collected traces from 30 machines for 15 days. During the period, around 5 million files are created, consuming 1.6TB space. For the purpose of evaluating the Diffloc protocol, we filtered out data read/write activities. Since on Linux, the functions of creating a new file and looking up an existing file are overloaded in the same `open` system call, we use the following criteria to decide whether an `open` call actually creates a new file: (1) the `O_CREAT` flag is set; (2) the file returned has zero size. We further map a file to a BlockID by hashing the ⟨device-number, inode-number⟩ combination of the file. In total, we collected 5.5 million creation oper-

ations, 0.52 million deletion operations, and 72.3 million lookup operations.

5.2 Simulator Design

The simulator models the environment where a storage cluster serves block creation, deletion or location requests generated by a distributed set of processes. The main objective of the simulator is to understand the system behavior related to data placement and location activities, thus disk operations or data transfers over the network are ignored.

Each client process is driven by a trace, which contains all the data location and placement requests generated by applications on the same machine over a period of time. Additionally, the simulator models the details of protocol state maintenance activities such as heartbeat packets and Bloom filter update packets. The data collected by the simulator include disk space usage and memory/bandwidth consumption.

The simulation begins with a pre-population phase in which blocks that exist before the traces start are created.

5.3 Effectiveness of the Differentiated Protocol Design

We first evaluate the effectiveness of the Diffloc protocol against DCH and BLOOM. Essentially, we want to validate the "+/−" signs in Figure 6 with quantitative results. We run the simulation for a 30-node storage cluster, where each node manages a 100GB disk. On each cluster node, there is also a client process driven by a trace collected using our tracing utility. Each cluster node multicasts heartbeat packets and Bloom filter updates every second.

We compare the three schemes in terms of storage utilization, data migration overhead, and memory consumption. We set the number of virtual nodes corresponding to each physical host to be 32. We measure the data migration cost by adding a new node into the system at the end of the simulation and calculating how much data needs to be migrated to the new node. We report memory consumption based on the memory used at the end of the simulation because the memory consumption grows as the file system is populated.

Storage utilization cannot be derived directly from the experiments because the amount of data created through trace replay is not large enough to saturate any host, while the measurement of *storage utilization* requires at least one host to run out of space by definition. We work around this problem by using a different metric that serves as an estimation of *storage utilization*. Our solution is based on the observation that good storage utilization can only be achieved when the storage consumption increases at roughly the same speed on all nodes. Suppose $p_1, p_2, ..., p_H$ are the percentages of used space of the H hosts by the end of the trace replay, then we define an estimation of storage utilization as:

$$\frac{\sum_{i=1}^{H} p_i}{H \times \max_{i=1}^{H} p_i}$$

The above formula gives us the storage utilization as if we could *shrink* individual host's storage capacity proportionally until some host runs out of space.

The experimental results are shown in Figure 10. We can see that Diffloc improves the storage utilization from 87.2% of DCH to 97.7%, and is comparable with BLOOM, which places objects purely based on usage. In terms of migration overhead, Diffloc reduces the amount of migrated data by more than 98% compared to DCH, from 3.3% (53.0GB) to 0.071% (1142MB) of total data volume. The reason for both improvements lies in the fact that the imbalanced storage usage and the migration overhead under DCH is due to its uncontrollable block placement policy. By applying DCH for small blocks which consume less than 10% of the total space, Diffloc effectively confines the extent of the imbalanced storage usage and data migration within a much smaller scale. Overall, the percentage of usable space and data migration overhead are significantly improved system-wide.

Metric	DCH	BLOOM	Diffloc
Storage utilization	87.2%	98.7%	97.7%
Migration overhead	53.0GB	0	1.142GB
Memory consumption	24KB	8.3MB	190KB

Figure 10: Effectiveness of the differentiated protocol design.

Secondly, Diffloc reduces the memory consumption by more than 97% compared to BLOOM. The reason lies in the fact that the memory consumption under BLOOM is roughly proportional to the number of blocks. By only using BLOOM to track the locations of large blocks, which are relatively few in number, Diffloc is able to significantly reduce the management overhead. Note that the results are only for a modest-sized cluster with 30 nodes. Even though the memory consumption for BLOOM seems affordable with today's commodity hardware; this figure could reach several hundred megabytes for large-scale clusters with hundreds of millions of blocks.

In conclusion, our Diffloc protocol takes advantage of both DCH and BLOOM schemes, makes effective use of available storage, and maintains low management overhead.

5.4 Access Efficiency

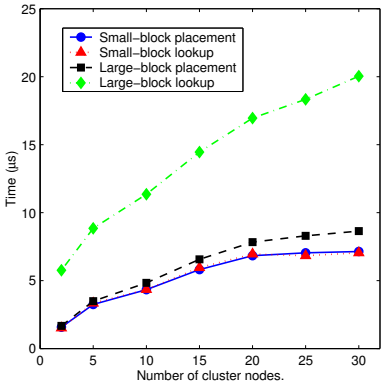


Figure 11: Protocol access performance.

We further evaluate the access performance of the protocol. We vary the number of cluster nodes from 2 to 30, and we run equal number of client processes as the cluster nodes. We measure the actual timing for invoking the block placement and lookup operations. The timing information is obtained through the `rdtsc` instruction, which reads the time stamp counter available on the x86 architecture. We separate the timing for operations on small and large blocks. The results are shown in Figure 11.

As we can see, the service time for block lookup and placement grows with the number of cluster nodes. For small blocks, the lookup and placement time grows logarithmically against the number of cluster nodes. For large blocks, the lookup time grows almost linearly, while the placement time grows logarithmically. In general, the overhead is very low and mainly involves CPU computation. These results confirm the theoretical analysis of Section 3.2.

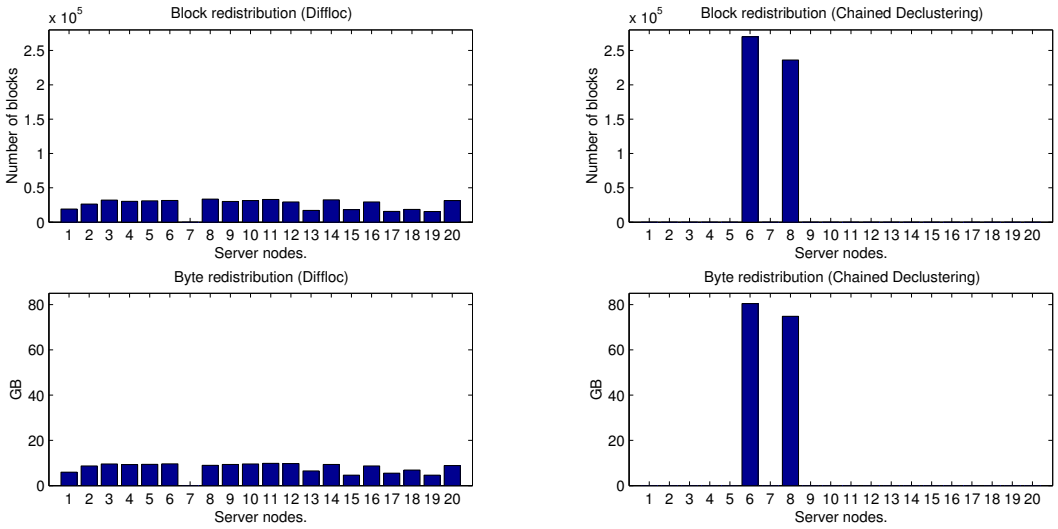


Figure 12: Workload redistribution after server 7 fails (Diffloc vs DCH). A total of 0.506 million blocks (155GB) data were stored on server 7. We show the distribution of the replicas of those blocks on other servers both in terms of block counts and byte counts. The subsequent load increase in terms of requests would be proportional to the block counts distribution; however, the increase in terms of I/O bandwidth would be proportional to the distribution of byte counts.

5.5 Workload Redistribution after Node Failure

We have also implemented the replication extension described in Section 4.3.2. Unlike Petal [26], which places block replicas on adjacent servers through *Chained Declustering*, Diffloc places block replicas in a pseudo-random fashion. Such a design offers better failure resilience because after losing a host, the workload that were destined to that host will be redistributed to almost all other nodes instead of the adjacent nodes.

We compare the load redistribution behavior of Diffloc against a simplified Chained Declustering scheme. We run the simulation for a 20-node storage cluster, the request workload is driven by 30 client traces. We set the replication degree to two and the capacity of each node to 200GB. At the end of the simulation, we calculate the load redistribution when server 7 fails. The load redistribution is expressed as a histogram of the distribution of blocks which are replicas of blocks on server 7. The results are shown in Figure 12. As we can see, when server 7 fails, the rest servers would take over similar shares of load increase in Diffloc. On the other hand, Chained Declustering would only redistribute load to server 6 and 8, which may causes significant load increase on these nodes.

5.6 Data Distribution with Non-uniform Hosts

We further illustrate that the Diffloc protocol can effectively utilize storage resources of non-uniform hosts. Unlike Chord [39], Diffloc does not require the setting of virtual nodes to be proportional to a host's capacity to utilize the storage resources. Specifically, the setting of virtual nodes would only affect the distribution of small blocks, and would have little impact to the whole system's storage utilization, which is dominated by large blocks.

In the following experiment, we employ 20 storage nodes with non-uniform capacity: half of them with 50GB disk each, and the other half with 100GB each. We map each host to 32 virtual nodes regardless of its capacity so that the number of small blocks would be similar on different hosts. We show the distribution of small and large blocks when the total data volume reaches 0.4TB, 0.8TB, or 1.2TB. As a comparison, we also show the distribution of blocks when a uniform DCH scheme is used, in which case the cluster saturates at 0.89TB. Figure 13 shows the experimental results. As we can see, Diffloc progressively places large blocks based on storage availability, which leads to high storage utilization. However, in the case of DCH, more than half of the capacity of the 100GB nodes are not used.

It should be noted that DCH may be modified to use a different number of virtual nodes for each host, which can adjust the block distribution among non-uniform hosts in term of the number of blocks. However, such an extension does not consider the size variations among blocks, nor could it effectively control the block distribution to meet other objectives.

6. CONCLUDING REMARKS

This paper presents the design and implementation of a differentiated data location protocol for self-organizing storage clusters. We use consistent hashing to place and locate small blocks. For large blocks, we can support any placement scheme with Bloom filter-based tracking. The main advantage of this protocol is that it is scalable with very low management overhead and makes effective use of storage resource in comparison with uniform strategies. The effectiveness of the protocol is validated through trace-driven simulation studies.

Our work is in large part motivated by previous work on parallel/distributed file systems and cluster-based storage systems [2, 4, 5, 10, 19, 20, 24, 26, 32, 37]. Previous studies mainly focus on infrastructural support of unifying distributed storage resources and have not addressed adequately how blocks should be placed and located efficiently in response to node additions and departures. Thus our work is complementary. The protocol described in this paper is part of the Sorrento project (Homepage: `http://www.cs.ucsb.edu/projects/Sorrento/`), with the goal of making storage clusters self-organizing and minimizing human administration. Much future work remains. We plan to conduct a detailed study of workload characteristics of storage clusters used by a wide range of data-intensive applications. We also plan to integrate our protocol in a parallel file system and evaluate its effectiveness using application benchmarks with data replication.

Acknowledgment. This work was supported in part by NSF

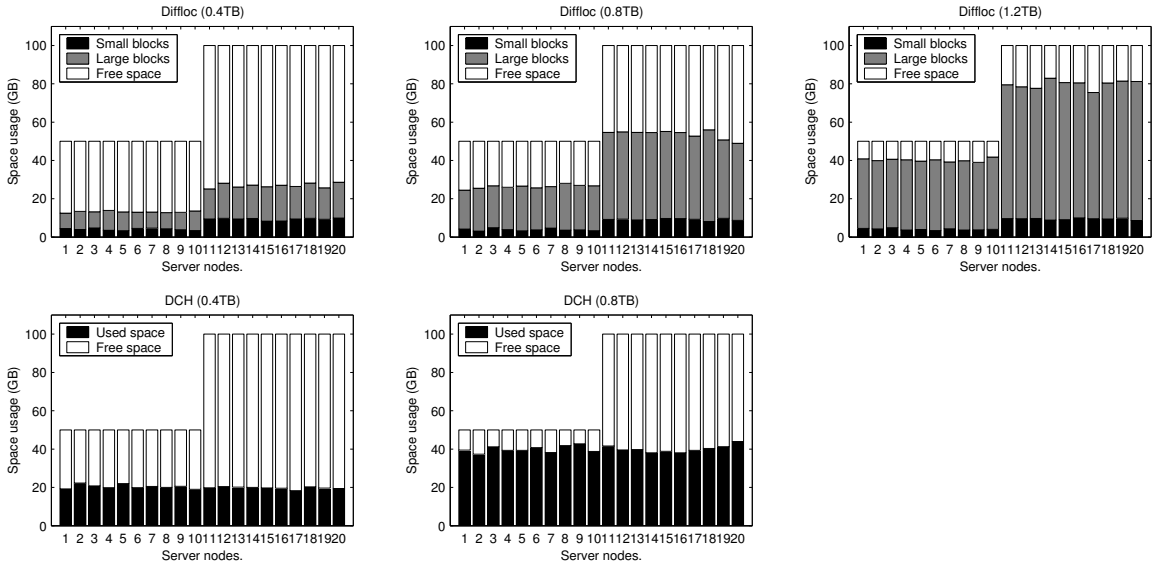


Figure 13: Data distribution on non-uniform hosts. We draw the snapshots when the total data volume reaches 0.4TB, 0.8TB, and 1.2TB. We compare Diffloc against DCH. Note that since DCH saturates around 0.89TB, we do not have the distribution corresponding to the 1.2TB case for DCH.

ACIR-0082666, 0086061, 0234346, and EIA-0080134. We would like to thank the anonymous referees, Aziz Gulbeden, Ambuj Singh, and Alan Sussman for their valuable comments and help.

7. REFERENCES

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

[2] CXFS: A high-performance, multi-OS SAN file system from SGI. SGI White Paper. URL `http://www.sgi.com/products/storage/cxfs.html`.

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

[4] D. Anderson, J. Chase, and A. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI 00)*, October 2000.

[5] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 95)*, December 1995.

[6] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM symposium on Operating systems principles (SOSP 91)*, pages 198–212, Pacific Grove, CA, 1991. ACM Press. ISBN 0-89791-447-3.

[7] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[8] S. A. Brandt, L. Xue, E. L. Miller, and D. D. E. Long. Efficient metadata management in large distributed file systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 290–298, April 2003.

[9] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform requirements. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 02)*, pages 53–62, Winnipeg, Manitoba, Canada, 2002. ACM Press.

[10] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[11] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, Boston, MA, December 2002.

[12] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: a high-performance remote-sensing database. In *Proceedings of the 13th International Conference on Data Engineering (ICDE 97)*, Birmingham, U.K., 1997.

[13] J. Chase, D. Anderson, P. Thakur, and A. Vahdat. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 01)*, October 2001.

[14] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of SuperComputing*, 2002.

[15] P. F. Corbett, D. G. Feltelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. D. Herr, J. Kavaky, T. R. Morgan, and A. Ziotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, 1995. ISSN 0018-8670.

[16] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 01)*, Chateau Lake Louise, Banff, Canada, Octorber 2001.

[17] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC 87)*, pages 1–12. ACM Press, 1987.

[18] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[19] J. Hartman, I. Murdock, and T. Spalink. The Swarm scalable

storage system. In *Proceedings of International Conference on Distributed Computing Systems*, pages 74–81, 1999.

[20] J. Hartman and J. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13 (3):274–310, 1995. ISSN 0734-2071.

[21] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA 02)*, pages 41–52, August 2002.

[22] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 03)*, Nice, France, April 2003.

[23] 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 *Proceedings of ACM Symposium on Theory of Computing (STOC 97)*, pages 654–663, 1997.

[24] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP 91)*, volume 25, pages 213–225. ACM Press, 1991.

[25] 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 *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 00)*. ACM, November 2000.

[26] E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 96)*, pages 84–92, Cambridge, MA, 1996.

[27] E. Lee, C. Thekkath, C. Whitaker, and J. Hogg. A Comparison of Two Distributed Disk Systems. Technical Report 155, Compaq (DEC) System Research Center, April 1998.

[28] W. Litwin, M.-A. Neimat, and D. Schneider. LH* — Linear Hashing for distributed files. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of data*, pages 327–336, Washington, DC, 1993. ACM Press. ISBN 0-89791-592-5.

[29] W. Litwin, M.-A. Neimat, and D. Schneider. LH* — A scalable, distributed data structure. *ACM Transactions on Database Systems (TODS)*, 21(4):480–525, 1996.

[30] W. Litwin and T. Schwarz. LH*RS : A high-availability scalable distributed data structure using reed solomon codes. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of data*, pages 237–248, 2000.

[31] T. Liu and M. Martonosi. Impala: A middleware system for managing autonomic parallel sensor systems. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 03)*, San Diego, CA, June 2003.

[32] S. Mullender and A. Tanenbaum. A distributed file service based on optimistic concurrency control. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP 85)*, pages 51–62, Orcas Island, WA, 1985. ACM Press. ISBN 0-89791-174-1.

[33] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer Magazine*, 21(2), 1988.

[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 *Proceedings of the International Conference on Computer Design (ISCA 97)*, 1997.

[35] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 01)*, pages 161–172, San Diego, CA, August 2001. ACM Press. ISBN 1-58113-411-8.

[36] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *Proceedings of the 2nd Conference on File and Storage Technologies (FAST 03)*, pages 59–72, San Francisco, CA, March 2003.

[37] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 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 *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS 01)*, pages 197–208, San Francisco, CA, March 2001.

[39] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 01)*, pages 149–160, San Diego, CA, August 2001. ACM Press.

[40] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 97)*, pages 224–237, 1997.

[41] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM symposium on Operating systems principles (SOSP 99)*, pages 93–109, Charleston, SC, 1999. ACM Press. ISBN 1-58113-140-2.

[42] J. Waxman and J. McArthur. Storage area networking — Opportunity for the indirect channel. IDC White Paper, 2000.

[43] Z. Zhang and K. Ghose. yFS: A journaling file system design for handling large data sets with reduced seeking. In *Proceedings of the 2nd Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003.