Supporting Scalable and Adaptive Metadata Management in Ultralarge-Scale File Systems

Yu Hua, *Member*, *IEEE*, Yifeng Zhu, *Member*, *IEEE*, Hong Jiang, *Senior Member*, *IEEE*, Dan Feng, *Member*, *IEEE*, and Lei Tian

Abstract—This paper presents a scalable and adaptive decentralized metadata lookup scheme for ultralarge-scale file systems (more than Petabytes or even Exabytes). Our scheme logically organizes metadata servers (MDSs) into a multilayered query hierarchy and exploits grouped Bloom filters to efficiently route metadata requests to desired MDSs through the hierarchy. This metadata lookup scheme can be executed at the network or memory speed, without being bounded by the performance of slow disks. An effective workload balance method is also developed in this paper for server reconfigurations. This scheme is evaluated through extensive trace-driven simulations and a prototype implementation in Linux. Experimental results show that this scheme can significantly improve metadata management scalability and query efficiency in ultralarge-scale storage systems.

Index Terms—File systems, Bloom filters, metadata management, scalability, performance evaluation.

1 Introduction

Metadata management is critical in scaling the overall performance of large-scale data storage systems [1]. To achieve high data throughput, many storage systems decouple metadata transactions from file content accesses by diverting large volumes of data traffic away from dedicated metadata servers (MDSs) [2]. In such systems, a client contacts MDS first to acquire access permission and obtain desired file metadata, such as data location and file attributes, and then directly accesses file content stored on data servers without going through the MDS. While the demand for storage increases exponentially in recent years, exceeding Petabytes (10^{15}) already and reaching Exabytes (10^{18}) soon, such decoupled design with a single metadata server can still become a severe performance bottleneck. It has been shown that metadata transactions account for more than 50 percent of all file system operations [3]. In scientific or other data-intensive applications [4], the file size ranges from a few bytes to multiple terabytes, resulting in millions of pieces of metadata in directories [5]. Accordingly, scalable and decentralized metadata management schemes [6], [7], [8] have been proposed to scale up the metadata throughput by judiciously distributing heavy management workloads among multiple metadata servers while maintaining a single writable namespace image.

- *Y. Hua, D. Feng, and L. Tian are with the School of Computer Science and Technology, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {csyhua, dfeng, ltian}@hust.edu.cn.*
- *Y. Zhu is with the Department of Electrical and Computer Engineering, University of Maine, 5708 Barrows Hall, Room 271, Orono, ME 04469-5708. E-mail: zhu@eece.maine.edu.*
- *H. Jiang is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0150. E-mail: jiang@cse.unl.edu.*

Manuscript received 5 Jan. 2009; revised 1 Nov. 2009; accepted 3 Feb. 2010; published online 27 May 2010.
Recommended for acceptance by G. Agrawal.
For information on obtaining reprinsts of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-01-0002.
Digital Object Identifier no. 10.1109/TPDS.2010.116.

One of the most important issues in distributed metadata management is to provide efficient metadata query service. Existing query schemes can be classified into two categories: probabilistic lookup and deterministic lookup. In the latter, no broadcasting is used at any point in the query process. For example, a deterministic lookup typically incurs a traversal along a unique path within a tree, such as a directory tree [9] or an index tree [10]. The probabilistic approach employs lossy data representations, such as Bloom filters [11], to route a metadata request to its target MDS with a very high accuracy. Certain remedy strategy, such as broadcasting or multicasting, is needed for rectifying incorrect routing. Compared with the deterministic approach, the probabilistic one can be much easily adopted in distributed systems and allows flexible workload balance among metadata servers.

1.1 Motivations

We briefly discuss the strengths and weaknesses of some representative metadata management schemes to motivate our research. Existing schemes can be classified into hash-based, table-based, static and dynamic tree partitions, and Bloom-filter-based structures, as shown in Table 1.

- Lustre [12], Vesta [13], and InterMezzo [14] utilize hash-based mappings to carry out metadata allocation and perform metadata lookups. Due to the nature of hashing, this approach can easily achieve load balance among multiple metadata servers, execute fast query operations for requests, and only generate very low memory overheads. Lazy Hybrid (LH) [2] provides a novel mechanism by allowing for pathname hashing with hierarchical directory management, but entails certain metadata migration overheads. This overhead is sometimes prohibitively high when an upper directory is renamed or the total number of MDSs is changed. In these cases, hash values have to be recomputed to reconstruct the mapping between metadata and their associated

TABLE 1
Comparison of *G-HBA* with Existing Structures Where n and d are the Total Numbers of Files and Partitioned Subdirectories, Respectively

	Examples	Load Balance	Migration Cost	Lookup Time	Memory Overhead	Directory Operations	Recovery	Scalability
Hash-based mapping	Lustre, Vesta, InterMezzo	Yes	Large	$O(1)$	0	Medium	Lustre & InterMezzo	Lustre
Table-based Mapping	xFS, zFS	Yes	0	$O(logn)$	$O(n)$	Medium	Yes	Yes
Static Tree Partition	NFS, AFS, Coda, Sprite, Farsite	No	0 (Farsite: small)	$O(logd)$	$O(1)$	Fast	Yes	Medium (Coda & Sprite: High)
Dynamic Tree Partition	OBFS, Ceph (Crush)	Yes	Large (Ceph: small)	$O(logd)$	$O(d)$	Fast	Yes	Yes
Bloom Filter-based	HBA, Summary Cache, Globus-RLS	Yes	0	$O(1)$	$O(n)$	Fast	No	Yes
	G-HBA	Yes	Small	$O(1)$	$O(n/m)$	Fast	Yes	Yes

servers, and accordingly, large volumes of metadata might need to be migrated to new servers.

- xFS [15] and zFS [16] use table-based mapping, which does not require metadata migration and can support failure recovery. In large-scale systems, this approach imposes substantial memory overhead for storing mapping tables, and thus often degrades overall performance.
- Systems using static tree partition include NFS [17], AFS [18], Coda [19], Sprite [20], and Farsite [21]. They divide the namespace tree into several non-overlapped subtrees and assign them statically to multiple MDSs. This approach allows fast directory operations without causing any data migration. However, due to the lack of efficient mechanisms for load balancing, static tree partition usually leads to imbalanced workloads, especially when access traffic becomes highly skewed [22].
- Dynamic subtree partition [23] is proposed to enhance the aggregate metadata throughput by hashing directories near the root of the hierarchy. When a server becomes heavily loaded, some of its subdirectories automatically migrate to other servers with light load. Ceph [24] maximizes the separation between data and metadata management by using a pseudorandom data distribution function (CRUSH) [25], which is derived from RUSH (Replication Under Scalable Hashing) [26] and aims to support a scalable and decentralized placement of replicated data. This approach works at a smaller level of granularity than the static tree partition scheme, and might cause slower metadata lookup operations. When an MDS joins or leaves, all directories need to be recomputed to reconstruct the tree-based directory structure, potentially generating a very high overhead in a large-scale file system.
- Bloom-filter-based approaches provide probabilistic lookup. A Bloom filter [11] is a fast and space-efficient data structure to represent a set. For each object within that set, it uses k independent hash functions to generate indices into a bit array and set the indexed bits in that array to 1. To determine the membership of a specific object, one simply checks whether or not all the bits pointed by these hash functions are 1. If *not*, this object is not in the set. If *yes*, the object is considered as a member. A false positive might happen, i.e., the object is considered as a member of the set, although it is not actually. However, the possibility of false positives is controllable and can be made very small. Due to high space efficiency and fast query response, Bloom filters have been widely utilized in storage systems, such as Summary Cache [27], Globus-RLS [28], and HBA [29]. However, these schemes use Bloom filters in a very simple way, where each node independently stores as many Bloom filters as possible in order to maintain the global image locally. Without coordination, these simple approaches can generate large memory overhead and reduce system scalability and reliability.

As summarized in Table 1 and discussed above, although each existing approach has its own advantages in some aspects, they are weak or deficient in some other aspects, in terms of performance metrics such as load-balance, migration cost, lookup time, memory overhead, directory operation overhead, scalability, etc. To combine their advantages and avoid their shortcomings, we propose a new scheme, called Group-based Hierarchical Bloom filter Array (*G-HBA*), to efficiently implement a scalable and adaptive metadata management for ultralarge-scale file systems. *G-HBA* uses Bloom filter arrays and exploits metadata access locality to achieve fast metadata lookup. It incurs small memory overheads and provides strong scalability and adaptability.

Specifically, the proposed scheme, called G-HBA, has performance advantages over other state-of-the-art schemes, in terms of memory space savings, fast query response, low migration costs, and strong scalability. The main reasons for

these advantages of G-HBA are fourfold. First, G-HBA makes use of fast and space-efficient Bloom filters to construct the indexing structure, which only needs to perform constant-time O(1) hashing to determine membership of queried files. Second, one essential characteristic of G-HBA is its ability to dynamically aggregate metadata servers into groups. The aggregation significantly decreases migration costs, since most associated metadata operations can be completed within one group. Third, in G-HBA, each group serves as a global mirror, and thus, data stored in a failed server can be easily reconstructed from its adjacent groups. Finally, the G-HBA scale can be resized dynamically via light-weight insertions and deletions. An insertion or deletion only requires one of the servers in a group to be updated.

1.2 Contributions

The proposed scheme in this paper, called *G-HBA*, judiciously utilizes Bloom filters to efficiently route requests to target metadata servers. Our *G-HBA* scheme exploits a Bloom-filter-based architecture, and considers dynamic and self-adaptive characteristics in ultralarge-scale file systems. Our main contributions are summarized below:

- We present a scalable and adaptive metadata management structure, called *G-HBA*, to store many metadata and support fast metadata lookups in an ultralarge-scale file system with multiple MDSs. The query hierarchy in *G-HBA* consists of four levels: local Least Recently Used (LRU) query and local query on an MDS, group multicast query within a group of MDSs, and global multicast query among all groups of MDSs. The multilevel file query is designed to be effective and accurate by capturing the metadata query locality and by dynamically balancing load among MDSs.
- We present a simple but effective group-based splitting scheme to improve file system scalability and maintain information consistency among multiple MDSs. This scheme adaptively and dynamically accommodates the addition and deletion of an MDS, in order to balance the load and reduce migration overheads.
- We design efficient approaches to querying files based on a hierarchical path. Note that the issue of *membership query* in metadata management, a focus of this paper, answers the most fundamental question, i.e., "which metadata sever in an ultralarge-scale distributed file system stores the metadata of the queried file?". Since this question helps to quickly access the target file data, fast membership queries based on *G-HBA* can directly reduce the time of accessing file data, especially in ultralarge-scale distributed file systems.
- We examine the proposed *G-HBA* structure through extensive trace-driven simulations and experiments on a prototype implementation in Linux. We examine operation latency, replica migration cost, and hit rate. Results demonstrate that our *G-HBA* design is highly effective and efficient in improving performance and scalability of file systems, and can provide scalable, reliable, and efficient service for metadata management in ultralarge-scale file systems.

The rest of the paper is organized as follows: Section 2 presents the basic scheme of *G-HBA*. Section 3 discusses some detailed design and optimization issues. The performance evaluation based on trace-driven simulations and prototype implementation can be given in Section 4 and Section 5, respectively. Section 6 summarizes related work, and Section 7 concludes the paper.

2 G-HBA Design

This section presents the design of *G-HBA* that supports fast membership queries in ultralarge-scale file systems.

2.1 Dynamic and Adaptive Metadata Management

We utilize an array of Bloom filters on each MDS to support distributed metadata lookup among multiple MDSs. An MDS, where a file's metadata reside, is called the *home MDS* of this file. Each metadata server constructs a Bloom filter to represent all files whose metadata are stored locally, and then replicates this filter to all other MDSs. A metadata request from a client can randomly choose an MDS to perform membership query against its Bloom filter array that includes replicas of the Bloom filters of the other servers. The Bloom filter array returns a hit when exactly one filter gives a positive response. A miss takes place when zero hit or multiple hits are found in the array. Since, we assume that the original metadata for any file can be stored in only one MDS, multiple hits, meaning that the original metadata of a file are found in multiple MDSs, potentially indicate a query miss.

The basic idea behind *G-HBA* in improving scalability and query efficiency is to decentralize metadata management among multiple groups of MDSs. We divide all N MDSs in the system into multiple groups, with each group containing at most M MDSs. Note that, we represent the actual number of MDSs in a group as M'. By judiciously using space-efficient data structures, each group can provide an approximately complete mapping between individual files and their home MDSs for the whole storage system. While each group can perform fast metadata queries independently to improve the metadata throughput, all MDSs within one group only store a disjointed fraction of all metadata, and they cooperate with each other to serve an individual query.

G-HBA utilizes Bloom filter (BF) based structures to achieve strong scalability and space efficiency. These structures are replicated among MDS groups, and each group contains approximately the same amount of replicas for load balancing. While each group maintains file metadata location information of the entire system, each individual MDS only stores information of its own local files and BF replicas from other groups. Within a given group, different MDSs store different replicas and all replicas in this group collectively constitute a global mirror image of the entire file system. Specifically, a group consisting of M' MDSs needs to store a total of $N - M'$ BF replicas from the other groups, and each MDS in this group maintains approximately $\frac{N-M'}{M'}$ replicas plus the BF for its own local file information.

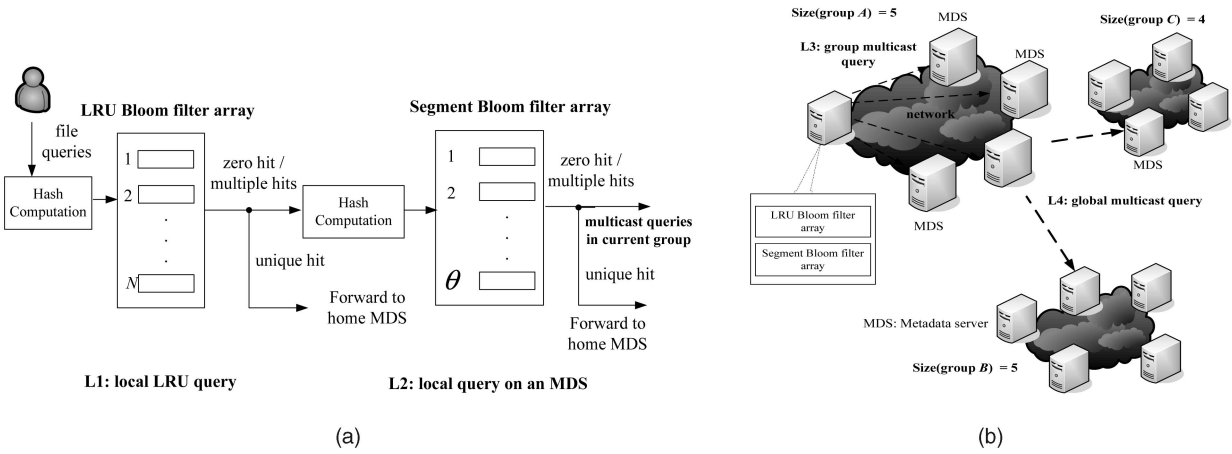


Fig. 1. The group-based HBA architecture allowing the multilevel query. (a) LRU and segment Bloom filter arrays allowing the L1 and L2 queries. (b) L3 and L4 queries, respectively, in a group and entire system.

A simple grouping in G-HBA may introduce large query costs and does not scale well. Since, each MDS only maintains partial information of the entire file system, the probability of successfully serving a metadata query by a single metadata server will decrease as the group size increases. Accordingly, an MDS has to multicast query requests more frequently to other MDSs, incurring higher network overheads and resulting in longer query delays.

Therefore, more effective techniques are needed to improve the scalability of the group-based approach. *G-HBA* addresses this issue by taking advantage of the locality widely exhibited in metadata query traffic. Specifically, each MDS is designed to maintain "hot data," i.e., home MDS information for recently accessed files, which are stored in an LRU Bloom filter array. Since "hot data" are typically small in size, the required storage space is relatively small.

2.2 Group-Based HBA Scheme

Fig. 1 shows a diagram of the *G-HBA* scheme. A query process at one MDS may involve four hierarchical levels: searching the locally stored LRU BF Array (L1), searching the locally stored Segment BF Array (L2), multicasting to all MDSs in the same group to concurrently search all Segment BF Arrays stored in this group (L3), and multicasting to all MDSs in the system to directly search requested metadata (L4). The multilevel metadata query is designed to be effective by judiciously exploiting access locality and dynamically balancing load among MDSs, as discussed in detail in Section 3.

Each query is performed sequentially in these four levels. A miss at one level will lead to a query to the next higher level. The query starts at the LRU BF array (L1), which aims to accurately capture the temporal access locality in metadata traffic streams. Each MDS maintains an LRU list that includes the most recently visited files whose metadata are maintained locally on that MDS. We further make use of the LRU BF to represent all the files cached in this LRU list. The LRU BF is then globally replicated to all MDSs of the entire system. As a replacement occurs in the LRU list on an MDS, corresponding insertion and deletion operations are then performed by this MDS to update its LRU BF. The LRU BF is then replicated to all the other MDSs when the amount of changes, in terms of the percentage of flipped bits, exceeds some threshold. The inconsistency can potentially lead to false positives, but not false negatives. The penalty of a false positive includes a waste of query message to the indicated MDS. On the other hand, a miss on LRU BF requires a query on local MDSs that must contain the queried file, thus avoiding any false negative.

If the query cannot be successfully served at L1, the query is then performed at L2, as shown in Fig. 1a. The Segment BF array (L2) stored on an MDS i includes only θ_i BF replicas, with each replica representing all files whose metadata are stored on that corresponding MDS. Suppose that the total number of MDS is N, typically θ_i is much smaller than N. And we have $\sum_{i=1}^{M'} \theta_i = N$, where θ_i is the number of BF replicas stored on MDS i. In this way, each MDS only maintains a subset of all replicas available in the systems. A lookup failure at L2 will lead to a query multicast among all MDSs within the current group (L3), as shown in Fig. 1b. At L3, all BF replicas present in this group will be checked. At the last level of the query process, i.e., L4, each MDS directly performs a lookup by searching its local BF and disk drives. If the local BF responds negatively, the requested metadata are not stored locally on that MDS, since the local BF has no false negatives [30]. However, if the local BF responds positively, a disk access is then required to verify the existence of the requested metadata, since the local BF can potentially generate false positives.

2.3 Critical Path for G-HBA Multilevel Query Service

The critical path of a metadata query starts at L1. When the L1 Bloom filter array returns a unique hit for the membership query, the target metadata are then most likely to be found at the server whose LRU Bloom filter generates such a unique hit. If zero or multiple hits take place at L1, implying a query failure, the membership query is then performed on the L2 Bloom filter array, which maintains the mapping information for a fraction of the entire storage system by storing $\theta = \lfloor \frac{N-M'}{M'} \rfloor$ replicas. A unique hit in any L2 Bloom filter array does not necessarily indicate a query success, since 1) Bloom filters only provide probabilistic membership query, and a false positive may occur with a

very small probability, and 2) each MDS only contains a subset of all replicas, and thus, is only knowledgeable of a fraction of the entire file-server mapping. The penalty for a false positive, where a unique hit fails to correctly identify the home MDS, is that a multicast must be performed within the current MDS group (L3) to solve this misidentification. The probability of a false positive from the segment Bloom filter array of one MDS, f_g^+, is given as below:

$$f_g^+ = \binom{\theta}{1} f_0(1-f_0)^{\theta-1} \approx \theta(0.6185)^{m/n}(1-(0.6185)^{m/n})^{\theta-1}, \quad (1)$$

where θ is the number of BF replicas stored locally on one MDS, m/n is the Bloom filter bit ratio, i.e., the number of bits per file, and f_0 is the optimal false rate in standard Bloom filters [30]. By storing only a small subset of all replicas, and thus achieving significant memory space savings, the group-based approach (segment Bloom filter array) can afford to increase the number of bits per file (m/n) so as to significantly decrease the false rate of its Bloom filters, hence rendering f_g^+ sufficiently small.

When the segment Bloom filter of an MDS returns zero or multiple hits for a given metadata lookup, indicating a local lookup failure, this MDS then multicasts the query request to all MDSs in the same group, in order to resolve this failure within this group. Similarly, a multicast is necessary among all other groups, i.e., at the L4 level, if the current group returns zero or multiple hits at L3.

2.4 Updating Replicas

Updating stale Bloom filter replicas involves two steps: replica identification (localization) and replica content update. Within each group, a BF replica resides exclusively on one MDS. Furthermore, the dynamic and adaptive nature of server reconfiguration, such as MDS insertion into or deletion from a group (see Section 3.1), dictates that a given replica must migrate from one MDS to another within a group from time to time. Thus, to update a BF replica, we must correctly identify the target MDS in which this replica currently resides. This replica location information is stored in an identification (ID) Bloom Filter Array ($IDBFA$) that is maintained in each MDS. A unique hit in $IDBFA$ returns the MDS ID, thus allowing the update to proceed to the second step, i.e., updating BF replica at the target MDS. Multiple hits in $IDBFA$ lead to a light false-positive penalty, since a falsely identified target MDS can simply drop the update request after failing to find the targeted replica. The probability of such a false positive can be extremely low. A counting Bloom filter [27] replaces each bit in a standard Bloom filter with a counter to support deletion operation. Each indexed counter is incremented when adding an element, and is decremented when removing an element. In our design, when a server departure occurs, we hash the server ID into the IDBFA, and the hit counters are then decreased by 1 to remove the server. Since $IDBFA$ only maintains the information about where a replica can be accessed, the total storage requirement of $IDBFA$ is negligible. For example, when the entire file system contains 100 MDSs, $IDBFA$ only takes less than 0.1 kB of storage on each MDS.

G-HBA does not use modular hashing to determine the placement of the newest replica within one MDS group. One main reason is that, this approach cannot efficiently support dynamic MDS reconfiguration, such as an MDS joining or leaving the storage system. When the number of servers changes, the hash-based recomputations can potentially assign a new target MDS for each existing replica within the same group. Accordingly, the replica would have to be migrated from the current target MDS to a new one in the group, potentially incurring prohibitively high network overheads.

3 Dynamic and Adaptive Group Reconfigurations

In this section, we present our design to support dynamic group reconfiguration and identify the optimal group configuration.

3.1 Lightweight Migration for Group Reconfiguration

Within each group, $IDBFA$ can facilitate load balance and support lightweight replica migration during group reconfiguration. When a new MDS joins the system, it chooses a group that has less than M MDSs, acquires an appropriate amount of BF replicas, and offloads some management tasks from the existing MDSs in this group. Specifically, each existing MDS can randomly offload $Number(CurrentReplicas) - \lceil (N-M')/(M'+1) \rceil$ replicas to the new MDS. Meanwhile, the MDS IDs of replicas migrating to the new MDS need to be deleted from their original ID Bloom filters and inserted into the ID Bloom filter on the new MDS. Any modified Bloom filter in $IDBFA$ also needs to be sent to the new MDS, which forms a new $IDBFA$ containing updated information of replica location. This new $IDBFA$ is then multicast to other MDSs. In this way, we can implement a lightweight replica migration and achieve load balance among multiple MDSs of a group.

Due to system reconfiguration by the system administrator, an MDS departure triggers a similar process but in a reverse direction. It involves 1) migrating replicas previously stored on the departing MDS to the other MDSs within that group, 2) removing its corresponding Bloom filter from the $IDBFA$ on each MDSs of that group, and 3) sending a message to the other groups to delete its replica. The network overhead of this design is small, since group reconfiguration happens infrequently and the size of $IDBFA$ is small.

3.2 Group Splitting and Merging

To further minimize the replica management overhead, we propose to dynamically perform group splitting and merging. When a new MDS is added to a group G that already has $M' = M$ MDSs, a group split operation is then triggered to divide this group into two approximately equal-sized groups, A and B. The split operation will be performed under two conditions: 1) each groups must still maintain a global mirror image of the file system, and 2) workload must be balanced within each group. After splitting, A and B consist of $M - \lfloor M/2 \rfloor$ and $\lfloor M/2 \rfloor + 1$ MDSs, respectively, for a total of $(M+1)$ MDSs. The

TABLE 2
Symbol Representations

Symbol	Description
P_{LRU}	Unique hit rate in the LRU Bloom filters
P_{L2}	Unique hit rate in the 2nd level Bloom filters
D_{LRU}	Latency in the LRU Bloom filters
D_{L2}	Latency in the 2nd level Bloom filters
D_{group}	Latency in one group
$D_{net.}$	Latency in entire multicast network

group splitting process is equivalent to deleting $\lfloor M/2 \rfloor$ MDSs from G by applying the aforementioned MDS deletion operation $\lfloor M/2 \rfloor$ times. Each deleted MDS from G is then inserted into group B.

Inversely, whenever the total size of two groups is equal to or less than the maximum allowed group size M due to MDS departures, these groups are then merged into a single group by using the lightweight migration scheme. This process repeats until no merging can be performed.

3.3 Optimal Group Configuration

One of our key design issues in *G-HBA* is to identify the optimal M, i.e., the maximum number of MDSs allowed in one group. M can strike different tradeoffs between storage overhead and query latency. As M increases, the average number of replicas stored on one MDS, represented as $\frac{N-M}{M}$, is reduced accordingly. A larger M, however, typically leads to a larger penalty for the cases of false positives as well as zero or multiple hits at both the L1 and L2 arrays. This is because multicast is used to resolve these cases, and typically takes longer when more hops are involved. We discuss how to find the optimal M in the following.

To identify the optimal M, we use a simple benefit function that jointly considers storage overheads and throughput. Specifically, we aim to optimize the throughput benefits per unit memory space invested, a measure also called the *normalized throughput*. The throughput benefit $U_{G\text{-}HBA}(throu.)$ is represented by taking into account the latency that includes all delays of actual operations along the critical path of a query, such as queuing, routing, and memory retrieval. Equation (2) shows the function to evaluate the normalized throughput of *G-HBA*:

$$\Gamma = \frac{U_{G\text{-}HBA}(throu.)}{U_{G\text{-}HBA}(space)} = \frac{1}{U_{G\text{-}HBA}(laten.) * U_{G\text{-}HBA}(space)}, \tag{2}$$

where $U_{G\text{-}HBA}(space)$ and $U_{G\text{-}HBA}(laten.)$ represent the storage overhead and operation latency, respectively.

The storage overhead for *G-HBA* is represented in (3), which is associated with the numbers of stored replicas on each MDS

$$U_{G\text{-}HBA}(space) = \frac{N-M}{M}. \tag{3}$$

We then examine the operation latency, shown in (4), for *G-HBA*, by considering multilevel hit rates that may lead to different delays. Definitions for the variables used in (4) are given in Table 2.

TABLE 3
Scaled-Up RES and INS Traces

	RES (TIF=100)	INS (TIF=30)
hosts	1300	570
users	5000	9780
open (million)	497.2	1196.37
close (million)	558.2	1215.33
stat (million)	7983.9	4076.58

$$\begin{aligned} U_{G\text{-}HBA}(laten.) = {} & D_{LRU} + (1 - P_{LRU})D_{L2} \\ & + (1 - P_{LRU})\left(1 - \frac{P_{L2}}{M}\right)D_{group} \\ & + (1 - P_{LRU})\left(1 - \frac{P_{L2}}{M}\right)^{M} D_{net.} \end{aligned} \tag{4}$$

The optimal value for M, thus, is the one that maximizes the Γ function in (2).

4 Performance Evaluation

We examine the performance of *G-HBA* through trace-driven simulations and compare it with HBA [29], the state-of-the-art BF-based metadata management scheme, and one that is directly comparable to *G-HBA*. We use three publicly available traces, i.e., Research Workload (RES), Instructional Workload (INS) [3], and HP File System Traces [31]. In order to emulate the I/O behaviors in an ultralarge-scale file system, we choose to intensify these workloads by a combination of spatial scale-up and temporal scale-up in our simulation, and also, in prototype experiments presented in the next section. We decompose a trace into subtraces and intentionally force them to have disjoint group ID, user ID, and working directories by appending a subtrace number in each record. The timing relationships among the requests within a subtrace are preserved to faithfully maintain the semantic dependencies among trace records. These subtraces are replayed concurrently by setting the same start time. Note that the combined trace maintains the same histogram of file system calls as the original trace, but presents a heavier workload (higher intensity) as shown in [29], [32]. As a result, the metadata traffic can be both, spatially and temporally scaled-up by different factors, depending on the number of subtraces replayed simultaneously. The number of subtraces replayed concurrently is denoted as the *Trace Intensifying Factor* (TIF). The statistics of our intensified workloads can be summarized in Tables 3 and 4. All MDSs are initially populated

TABLE 4
Scaled-Up HP Trace

	Original	TIF=40
request (million)	94.7	3788
active users	32	1280
user accounts	207	8280
active files (million)	0.969	38.76
total files (million)	4.0	160.0

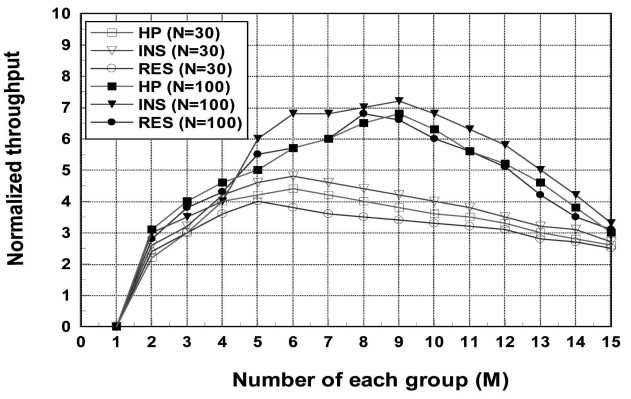


Fig. 2. Normalized throughput of *G-HBA* when the total number of MDSs is 30 and 100 MDSs, respectively.

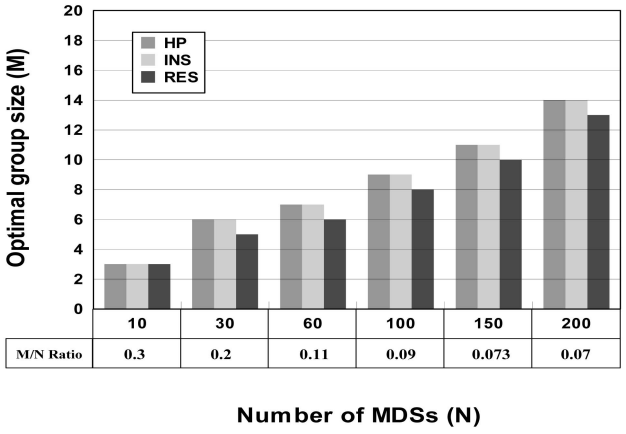


Fig. 3. Optimal group size as a function of the number of nodes.

randomly. Each request can randomly choose an MDS to carry out query operations. In addition, we use eight hash functions in Bloom filters to guarantee the false positive probability 0.039 percent.

The INS and RES traces are collected in two groups of Hewlett-Packard series 700 workstations running HP-UX 9.05. The HP File System trace is a 10-day trace of all file system accesses with a total of 500 GB of storage. Since, the three traces above have collected all I/O requests at the file system level, we filter out requests, such as *read* and *write*, which are not related to the metadata operations.

We have developed a trace-driven simulator to emulate dynamic behaviors of large-scale metadata operations and evaluate the performance in terms of hit rates, query delays, network overheads of replica migrations, and response times for updating stale replicas. The simulation study, in this paper, will focus on the increasing demands for ultralarge-scale storage systems, such as Exabyte-scale storage capacity, in which a centralized BF-based approach, such as the HBA scheme [29], will be forced to spill significant portions of replicas into the disk space as the fast increasing number of replicas overflows the main memory space.

HBA is, as its name suggests, a hierarchical scheme that maintains two-level BF arrays to support membership queries in a file system by exploiting the temporal access locality of file access patterns. Specifically, the first level represents the metadata location of most recently visited files on each MDS, and the second level maintains metadata distribution information of all files. The first level contains only very "hot" files' location information, but uses higher bit/item ratio in its BF to achieve lower false-hit rate of the BF, thus increasing query accuracy. The second level, on the other hand, contains location information of "all" files in the system, and thus, uses a lower bit/item ratio in its BF to increase space efficiency without significantly sacrificing query accuracy, since this level of BF array is much less frequently queried than the first level due to access locality.

4.1 Impact of Group Size M on *G-HBA* Performance

In this section, we present the details of identifying the optimal value of group size M by optimizing the normalized throughput of *G-HBA* given in (2) in Section 3.3. We generate the normalized throughput with the aid of simulation results, including hit rates and latency of multilevel query operations. Other simulation results are directly measured by statistical average values from 20 simulation runs.

The maximum group size, M, can potentially impose significant impact on the system performance of *G-HBA*, in terms of hit rates and query latency. While a larger M may save more memory space, as each MDS in *G-HBA* only needs to store $\frac{N-M}{M}$ BF replicas, it can increase the query latency, since fewer Bloom filters on each MDS can reduce local query hit rates at the L2 level. Therefore, an optimal M has to be identified.

Fig. 2 shows the normalized throughput as a function of M when the number of MDSs is 30 and 100, respectively, under the intensified HP, RES, and INS workloads. The optimal M is 6 for HP and INS, and 5 for RES when the number of MDSs is 30. The optimal M is 9 for HP and INS, and 8 for RES trace when the number of MDSs is scaled-up to 100.

Fig. 3 further shows the relationship between the optimal group size M and the total number of MDSs. We observe that M is not very sensitive to the workloads studied in this paper. In addition, when the number of MDSs is large, the optimal M value does not change significantly. These observations give us useful insights when determining the logical grouping structure for ultralarge-scale storage systems. It is recommended that some predefined M be used initially, and this suboptimal M be deployed until the total number of MDSs reaches some threshold.

4.2 Average Latency

Figs. 4, 5, and 6 plot the average latency of metadata operations as a function of the operation intensity (number of operations) under the HP, RES, and INS workloads, respectively. We utilize different memory sizes to evaluate the operation latency. With large memory, such as 800 MB in Fig. 4, 900 MB in Fig. 5, and 1.2 GB in Fig. 6, HBA outperforms *G-HBA* slightly since HBA, being able to store all the replicas in the main memory, is able to complete all operations within the memory locally, while *G-HBA* must examine replicas stored in other MDSs of the same group. However, as the available memory size decreases, more and more BF replicas are spilled into the hard disks, causing the average latency of the HBA scheme to increase rapidly since more disk accesses are involved in storing or retrieving BF

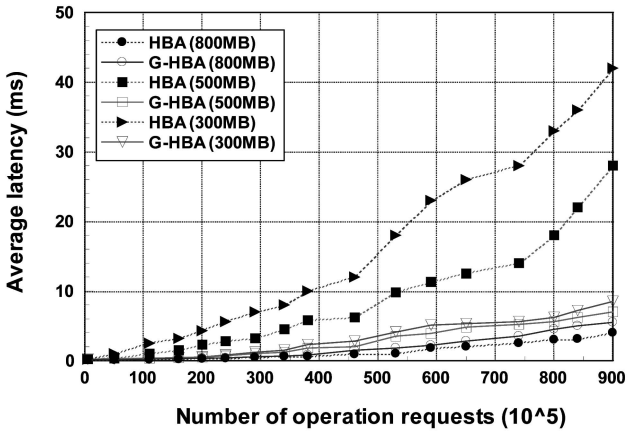


Fig. 4. Average latency comparisons of HBA and *G-HBA* with different memory sizes under the RES trace.

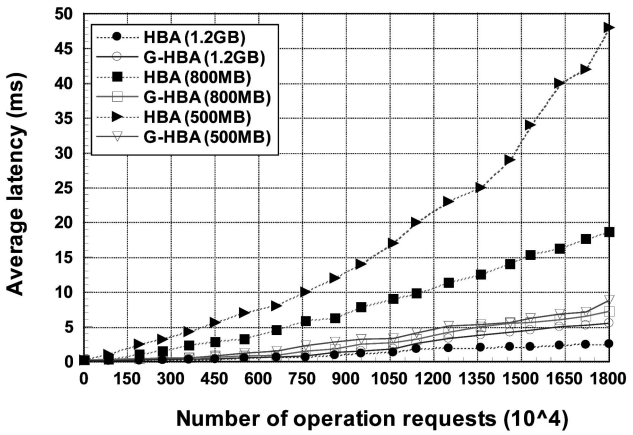


Fig. 6. Average latency comparisons of HBA and *G-HBA* with different memory sizes under the HP trace.

replicas in the hard disks. In contrast, *G-HBA* demonstrates the advantage of its space efficiency, as each MDS only needs to maintain a small subset of all replicas, i.e., $\frac{N-M'}{M'}$ replicas, enabling most, if not all, of the replicas to be stored in the memory, and thus outperforming HBA significantly.

4.3 Overhead of MDS Group Reconfiguration

Fig. 7 shows the overhead of adding a new MDS to the system, in terms of the amount of replica migration traffic for the HBA, hash-based placement, and *G-HBA* schemes. When a new MDS joins a system with N MDSs, HBA needs to migrate *all* existing N replicas to the new MDS, to maintain a global mirror image containing all metadata location information of the entire file system.

Hash-based placement, as discussed in Section 2.4, needs to recompute the locations (target MDSs) for $(N - M')$ replicas. Whenever the new position differs from the current one, a migration has to be performed. The number of replicas that need to be migrated is bounded by $(N - M')$. When the number of MDSs increases, the probability of mismatch also increases, resulting in more replicas being migrated. *G-HBA* only needs to migrate $\frac{N-M'}{M'+1}$ replicas to the newly inserted MDS, and thus significantly reduces network overheads in ultralarge-scale file systems.

4.4 Latency of Updating Stale Replicas

Figs. 8 shows the average latency of updating stale replicas under these three workloads. In HBA, a replica update, initiated from any MDS, triggers a system-wide multicast to update all MDSs in the system. In *G-HBA*, however, we only need to update the stale replica in each group (i.e., one MDS in each group), making *G-HBA* faster and more efficient.

4.5 Multilevel Query Hit Rate and Latency

Fig. 9 shows the hit rates of *G-HBA* as the number of MDSs increases. We examine the hit rates based on the four-level query critical path, presented in Section 2.3. A query checks L1 first. If zero or multiple hits occur, L2 is checked. A miss in L2 will lead to a lookup in L3. Finally, if the query against L3 still fails, we multicast the query message within the entire file system (i.e., L4) to obtain query results where every MDS in the system checks the query against its local Bloom filter. Since L1, i.e., the LRU Bloom filter array, is able to efficiently exploit the temporal locality of file access patterns, a large number of queries to the other levels are filtered out by L1. Our experiments show that more than 80 percent of query operations can be successfully served by L1 and L2. With the help of L3, more than 90 percent

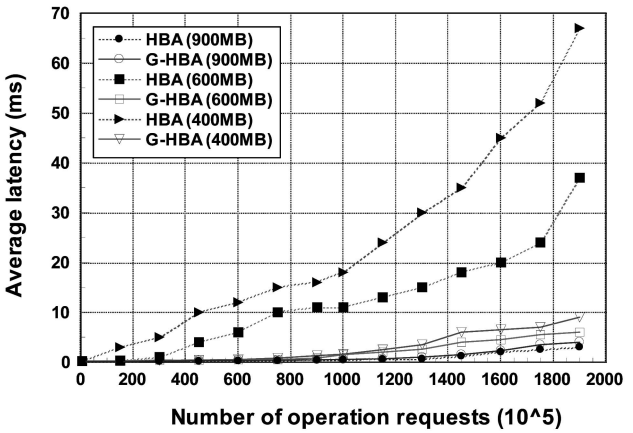


Fig. 5. Average latency comparisons of HBA and *G-HBA* with different memory sizes under the INS trace.

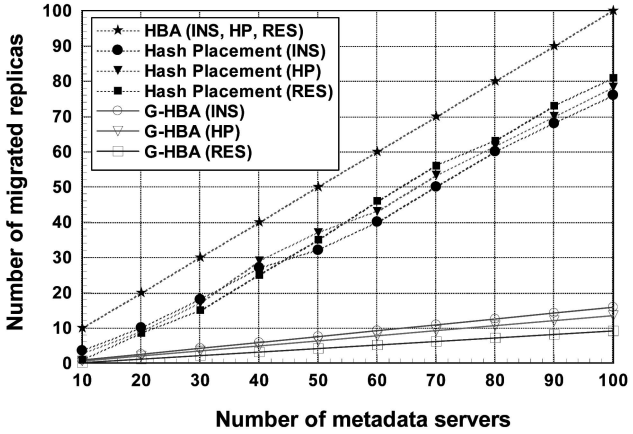


Fig. 7. Number of migrated replicas using HBA, hash-based placement, and *G-HBA* schemes.

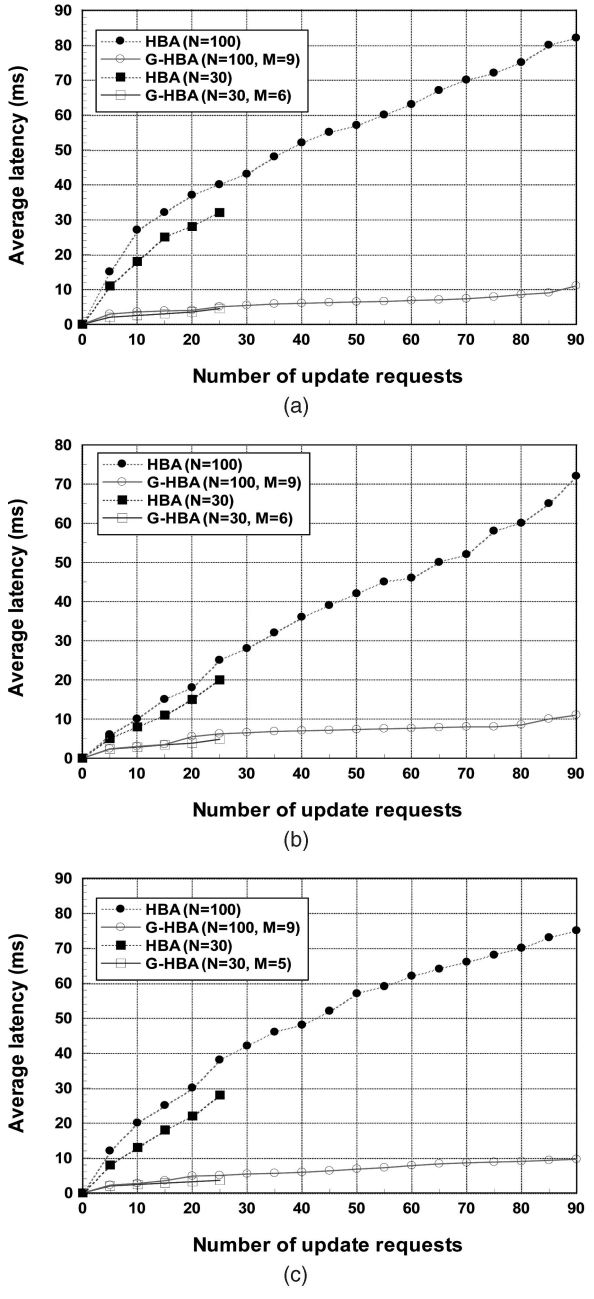


Fig. 8. Latency of updating stale replicas of HBA and *G-HBA* schemes using HP, RES, and INS traces. (a) HP trace. (b) INS trace. (c) RES trace.

requests are absorbed internally within one group, even with a system of 100 MDSs.

It is also observed that the percentage of queries served by L4 increases as the number of MDSs increases. This is because false positives and false negatives increase in a large system due to the large amount of stale replicas under the same constraints of network overheads [32]. The staleness is caused by nonreal-time updating in real systems. Here, a false positive occurs when a request returns an MDS ID that actually does not have the requested metadata. A false negative means that a query request fails to return an MDS ID that actually holds the requested metadata.

The final L4 query can provide guaranteed query services by multicasting query messages within entire

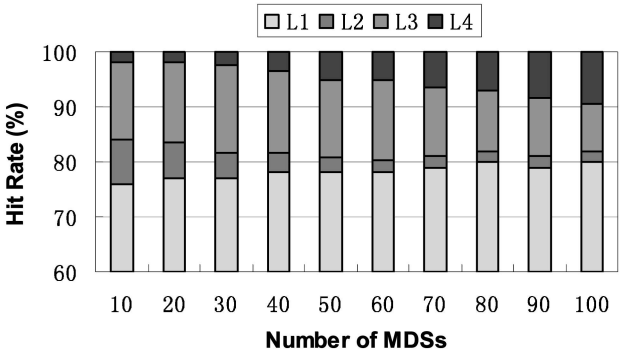


Fig. 9. Percentage of queries successfully served by different levels.

system. Since the operations take place in local MDSs, there are no false positives and negatives from stale data in distributed environments. Thus, if we still have multiple hits, they must come from Bloom filters themselves. Associated operations in a local MDS need to first check local Bloom filters that reside in memory, to determine whether the MDS may obtain the query result. If a local hit takes place, further checking may involve accessing the disk to conduct lookups on real metadata. Or else, we definitely know the queried data are nonexisting. Although the L4 operations require more costs, the probability of resorting to L4 is very small, as shown in our experiments.

Our design can provide failure-over support when an MDS departs or fails. Once an MDS failure is detected, the corresponding Bloom filters are removed from the other MDSs to reduce the number of false positives. This design is desirable in real systems, since the metadata service still remains functional when some MDSs fail, albeit at a degraded performance and coverage level.

In addition, we further test the query latency required by different levels, as shown in Fig. 10. We observe that the latency in L1 and L2 is much smaller than that in L3, since the latter needs to multicast query requests within one group to perform memory query. The L4 level produces the maximum latency due to multicasting within entire file system that involves all metadata servers. Therefore, G-HBA can obtain quick query response, since most query requests are satisfied at L1 or L2 level.

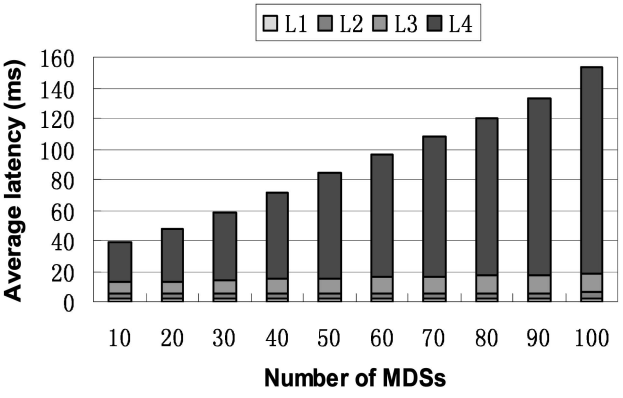


Fig. 10. Query latency in different levels.

TABLE 5
Scaled-Up EECS Trace

	Original	TIF=300
file system size (GB)	42	12600
active files (million)	0.17	51
total files (million)	0.57	171
number of metadata ops (million)	6.5	1950

5 Prototype Implementation and Evaluation

We have implemented a prototype of *G-HBA* in the Linux kernel 2.4.21 environment that consists of 60 nodes, each with Intel Core 2 Duo CPU and 1 GB memory. Each node in the system serves as an MDS. The prototype contains the functional components for handling multilevel query, splitting/merging, updating, and migration. All these components are implemented in the user space. A client captures the file system operations from the user traces and then delivers the query requests to the MDSs. Both, clients and servers, use multiple threads to exchange messages and data via TCP/IP. The IP encapsulation technique helps forward the query requests among multilevel MDSs, as shown in Section 2.3.

Since the HP trace contains more metadata operations and is more recent than the other two traces (i.e., INS and RES), we choose to use the HP trace that is scaled-up with an intensifying factor of 60, using the scaling approach described in Section 4.

We also use the I/O traces collected on the EECS NFS server (EECS) at Harvard [33], to evaluate our G-HBA performance. The EECS workload is dominated by metadata requests, and has a read/write ratio of less than 1.0. It has a total of 4.4 million operations, in which there are more than 75 percent metadata operations, such as `lookup`, `getattr`, and `access` calls. The total size of all traces is more than 400 GB, and only 42 GB traces (14 days) are randomly selected in our evaluation, as shown in Table 5. We further divide the storage system into groups based on the optimal M values, obtained through the optimal value calculation, described in Section 4.1, i.e., the optimal group size is 7 for both HP and EECS traces when the total number of MDSs is 60.

Our implementation experiments focus on evaluating the dynamic operations, which are rarely studied by other related research works. We use the traces to initialize our grouping scheme, and we artificially insert the node insertion and deletion requests within the uniform temporal intervals to trigger the associated MDS insertion and deletion operations. In addition, we use bit/file (b/f) ratios of 8 for HP trace and 10 for EECS trace.

5.1 Lookup Latency

Fig. 11 shows the experimental results in terms of query latency under the intensified HP trace. The results from our prototype implementation, consistent with the simulations in Section 4.2, further prove the efficiency of our proposed *G-HBA* architecture. *G-HBA* can decrease the query latency of *HBA* by up to 48.6 percent under the heaviest workload in our experiments, demonstrating its scalability. We also evaluate the lookup latency through examining EECS trace, where *G-HBA* obtains 41.5 percent latency saving under the heaviest workload, as shown in Fig. 12.

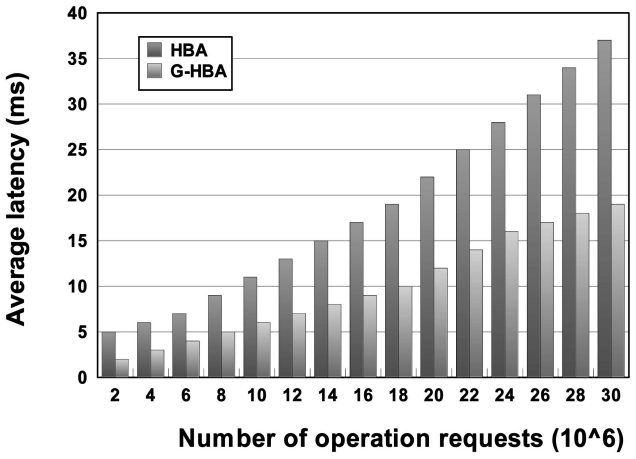


Fig. 11. Average query latency using intensified HP trace.

5.2 Overhead of Adding/Deleting MDSs

We evaluate the overhead of dynamic operations for adding new nodes and deleting existing nodes by examining the number of messages, generated during the process of MDS insertion and deletion. When adding a new node to a group, the group can directly accept it, if the group size has not reached the limit. Otherwise, the group is split into two, as shown in Section 3.2. After adding a node, the BF replica of the new node needs to be multicast to other groups in the system. Furthermore, some replicas of the existing MDSs within the same group need to be migrated to the new MDS to keep load balanced. In this experiment, we randomly choose a group to add a new node, which may or may not cause the group to be split. The operations of adding and deleting MDSs are associated with group-based reconfiguration, i.e., group splitting and merging.

Since each node in the *HBA* scheme maintains a global mapping image of the entire system, an MDS insertion or deletion requires it to exchange its own Bloom filter replica with all other MDSs. In contrast, *G-HBA*'s simple and

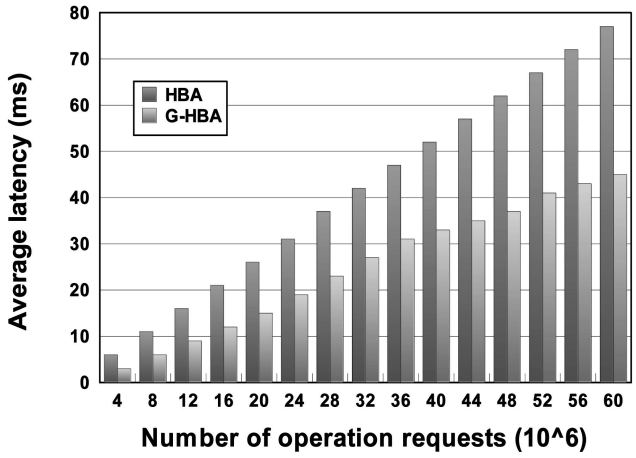


Fig. 12. Average query latency using intensified EECS trace.

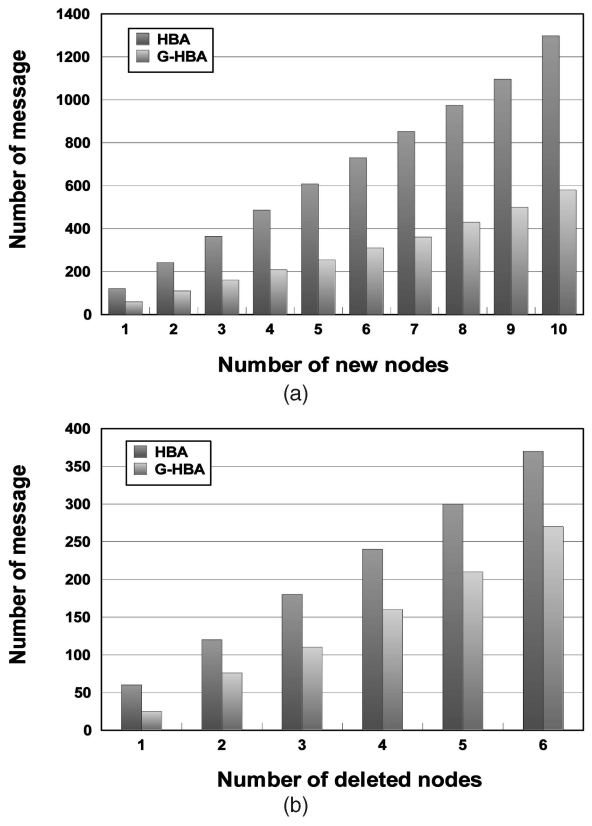


Fig. 13. The message numbers when adding and deleting nodes using HP trace. (a) Inserting nodes. (b) Deleting nodes.

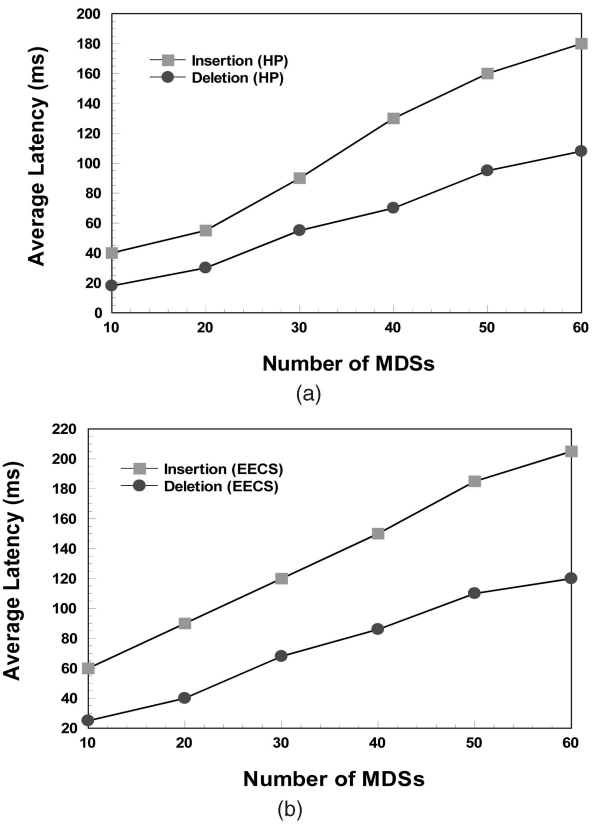


Fig. 14. Average latency when adding and deleting one node using HP and EECS traces. (a) HP trace. (b) EECS trace.

efficient group-based operations entail multicasting the BF replica of the new MDS to only one node of each group, achieving significant message savings.

Fig. 13 shows the number of messages, generated during the MDS insertion and deletion. The target node is selected randomly. We collect the number of total messages associated with all dynamic operations, including possible group splitting and merging. We observe that G-HBA outperforms HBA in terms of required messages, obtaining significant bandwidth savings. Table 6 further presents the message overheads of inserting and deleting nodes when using EECS trace.

5.3 Reconfiguration Latency Per Node

In this section, we examine the latency of adding or deleting one node as the system scales-up its size. In large-scale systems, the insertion of a node entails first executing the associated operations with a chosen group, including possible group splitting, and then multicasting the new message to all other groups of the entire system. The node deletion follows the similar operations but may trigger group merging operations. Fig. 14 shows the average latency when one node is added and deleted under the G-HBA scheme when using HP and EECS traces. The deletion operation produces smaller latency than node insertion, since the latter needs to first choose a group to obtain the given node, which is not required by the former.

TABLE 6
The Message Numbers when Adding and Deleting Nodes Using EECS Trace

		1	2	3	4	5	6	7
Insertion	HBA	136	253	381	507	631	746	852
	G-HBA	73	136	192	261	317	371	428
Deletion	HBA	82	159	231	306	372	446	535
	G-HBA	46	81	118	159	187	212	257

5.4 Memory Overhead Per MDS

We utilize the memory requirement, normalized to a pure Bloom Filter Array (BFA) with bit/file (b/f) ratios of 8 and 10, respectively, for HP and EECS traces to evaluate the memory overhead. The baseline system is to build only a Bloom filter for each MDS to represent all files stored locally, and then replicate this filter to all other MDSs. In the baseline system, each MDS stores a BFA that consists of all Bloom filters, including its local filter and the replicas of the Bloom filters from all other MDSs. A metadata request can obtain its lookup results from a randomly selected MDS, based on the membership query on all Bloom filters. This is the basic approach adopted by *HBA*, where an additional LRU Bloom filter array is incorporated to exploit the temporal locality of file access patterns to reduce the metadata operation time.

Each BFA maintains a global image of the entire system, and HBA needs to maintain an extra LRU Bloom filter array. *G-HBA* utilizes the group-based scheme to reduce space overhead and MDS insertion/deletion overhead. HP and EECS traces share the same group setting, and thus,

TABLE 7
Relative Space Overhead Normalized to BFA with One and Two Bit/File Ratios in HP and EECS Traces

Server #	BFA (b/f)	BFA $(2b/f)$	HBA	G-HBA
20	1.0	2.0	1.0002	0.2002
40	1.0	2.0	1.0004	0.1670
60	1.0	2.0	1.0006	0.1434
80	1.0	2.0	1.0008	0.1258
100	1.0	2.0	1.0010	0.1121

they have the same overheads of relative space. Table 7 shows a comparison among BFA with (b/f), BFA with $(2b/f)$, HBA and *G-HBA*, in terms of normalized memory requirement per MDS as a function of the number of MDSs. Clearly, *G-HBA* has a significantly lower memory overhead than both BFA and HBA, and its relative memory overhead decreases as the number of MDSs increases.

6 Related Work

Current file systems, such as OceanStore [34] and Farsite [21], can provide highly reliable storage, but cannot efficiently support fast query services of namespace or directory, when the number of files becomes very large due to access bottlenecks. Parallel file systems and platforms based on the object-based storage paradigm [35], such as Lustre [12], Panasas file system [36], and zFS [16], use explicit maps to specify where objects are stored at the expense of high storage space. These systems offer only limited support for distributed metadata management, especially in environments where workloads must rebalance, limiting their scalability and resulting in load asymmetries.

In large-scale storage architectures, the design for metadata partitioning among metadata servers is of critical importance for supporting efficient metadata operations, such as reading, writing, and querying items. Directory subtree partitioning (in NFS [17] and Coda [19]) and pure hashing (in Lustre [12] and RAMA [37]) are two common techniques used for managing metadata. However, they suffer from concurrent access bottlenecks. Existing parallel storage systems, such as PVFS [38] and Galley [39], can support data striping among multiple disks to improve data transfer rates, but lack efficient support for scalable metadata management, in terms of failure recovery and adaptive operations. Spyglass [40] utilizes namespace locality and metadata skewed distribution to carry out the mapping from namespace hierarchy into a multidimensional K-D tree to support fast metadata searching service. XFS [41], running on large SMPs, uses B^+ tree to increase the scalability of file systems and reduce algorithmic complexity from linear to logarithmic. The main advantage of Bloom filters over distributed hashing or B^+ tree is the space saving, which allows us to place more file metadata into high-speed memory and decrease bandwidth costs in updating replicas.

Metadata management in large-scale distributed systems usually provides query services to determine whether the metadata of a specific file reside in a particular metadata server, which, in turn, helps locate the file itself. Bloom filter [11], as a space-efficient data structure, can support query (membership) operations with $O(1)$ time complexity, since a query operation needs to probe *constant-scale* bits. Standard Bloom filters [11] have inspired many extensions and variants, such as the compressed Bloom filters [42], the space-code Bloom filters [43], the spectral Bloom filters [44], the distributed Bloom filters [45], and the beyond Bloom filters [46]. The counting Bloom filters [27] are used to support the deletion operation and represent a set that changes over time. Multi-Dimension Dynamic Bloom Filters (MDDBF) [47] can support representation and membership queries based on the multiattribute dimension. We have developed a novel Parallel Bloom Filters (PBF) and an additional hash table [48] to maintain multiple attributes of items and verify the dependency of multiple attributes, thereby significantly decreasing false positive rates.

Existing state-of-the-art work motivates our work that further improves upon them. Compared with existing work, G-HBA exhibits different characteristics from existing schemes in terms of function, data structure, and I/O interface, except HBA. Both, HBA and G-HBA, are designed to support membership queries to determine which MDS stores the queried file metadata. Note that these two schemes can determine the ID of the home MDS in which the file metadata reside, not its actual address. In addition, both HBA and G-HBA make use of the same data structure, i.e., Bloom filters, to obtain space savings and provide fast query response. Furthermore, the inputs in both HBA and G-HBA are the query requests for files and the outputs are the MDS ID of the MDS that stores the queried files. Finally, the general-purpose G-HBA and HBA are orthogonal to the existing schemes in that they are not designed to totally replace the latter but to improve their query performance and to provide good compatibility by using simple I/O interfaces. Based on the above reasons, we argue that it would be more objective and meaningful to compare G-HBA with HBA.

In addition, compared with the conference version in [49], this paper presents detailed comparisons against other state-of-the-art architectures of metadata management in terms of multiple metrics, describes dynamic operations, and shows extensive experimental results.

7 Conclusion

This paper presents a scalable and adaptive metadata lookup scheme, called *G-HBA*, for ultralarge-scale file systems. *G-HBA* organizes MDSs into multiple logic groups and utilizes grouped Bloom filter arrays to efficiently direct a metadata request to its target MDS. The novelty of *G-HBA* lies in that it judiciously confines most of metadata query and Bloom filter update traffic to a single server group. Compared with HBA, *G-HBA* is more scalable due to the facts that: 1) *G-HBA* has a much less memory space overhead than the former, and, thus, can potentially avoid accessing disks during metadata lookups in ultralarge-scale storage systems. 2) *G-HBA* significantly reduces the amount of global broadcast among all MDSs, such as that induced by Bloom filter updates. 3) *G-HBA* supports dynamic workload rebalancing when the server number changes,

by using a simple but efficient migration strategy. Extensive trace-driven simulations and a real implementation show that our *G-HBA* is highly effective and efficient in improving the performance, scalability, and adaptability of the metadata management component for ultralarge-scale file systems.

ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 60703046; US National Science Foundation (NSF) under Grants NSF-CNS-1016609, NSF-CCF-0621526, NSF-CCF-0937993, NSF-IIS-0916859, NSF-IIS-0916663, NSF-CCF-0937988, and NSF-CCF-0621493; National High Technology Research and Development 863 Program of China under Grant 2009AA01A401 and 2009AA01A402; National Basic Research 973 Program of China under Grant No. 2011CB302301 and National Natural Science Foundation of China under Grant No. 61025008 and the Program for Changjiang Scholars and Innovative Research Team in University under Grant IRT-0725. The authors greatly appreciate HP Labs, the University of California Berkeley, the Harvard University for file system traces, and anonymous reviewers for constructive comments. The work of Lei Tian was done in part while he was working at the CSE Department of UNL.

REFERENCES

[1] J. Piernas, T. Cortes, and J.M. Garcia, "The Design of New Journaling File Systems: The DualFS Case," *IEEE Trans. Computers,* vol. 56, no. 2, pp. 267-281, Feb. 2007.

[2] S.A. Brandt, E.L. Miller, D.D.E. Long, and L. Xue, "Efficient Metadata Management in Large Distributed Storage Systems," *Proc. 20th IEEE/NASA Goddard Conf. Mass Storage Systems and Technologies (MSST),* 2003.

[3] D. Roselli, J.R. Lorch, and T.E. Anderson, "A Comparison of File System Workloads," *Proc. Ann. USENIX Technical Conf.,* 2000.

[4] L. Guy, P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger, "Replica Management in Data Grids," technical report, GGF5 Working Draft, 2002.

[5] S. Moon and T. Roscoe, "Metadata Management of Terabyte Datasets from an IP Backbone Network: Experience and Challenges," *Proc. Workshop Network-Related Data Management (NRDM),* 2001.

[6] M. Cai, M. Frank, B. Yan, and R. MacGregor, "A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management," *J. Web Semantics: Science, Services and Agents on the World Wide Web,* vol. 2, no. 2, pp. 109-130, 2005.

[7] C. Lukas and M. Roszkowski, "The Isaac Network: LDAP and Distributed Metadata for Resource Discovery," *Internet Scout Project,* http://scout.cs.wisc.edu/research/isaac/ldap.html, 2001.

[8] D. Fisher, J. Sobolewski, and T. Tyler, "Distributed Metadata Management in the High Performance Storage System," *Proc. First IEEE Metadata Conf.,* 1996.

[9] A. Foster, C. Salisbury, and S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets," *J. Network and Computer Applications,* vol. 23, pp. 187-200, 2001.

[10] M. Zingler, "Architectural Components for Metadata Management in Earth Observation," *Proc. First IEEE Metadata Conf.,* 1996.

[11] B. Bloom, "Space/time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM,* vol. 13, no. 7, pp. 422-426, 1970.

[12] P.J. Braam, "Lustre Whitepaper," http://www.lustre.org, 2005.

[13] P.F. Corbett and D.G. Feitelson, "The Vesta Parallel File System," *ACM Trans. Computer Systems,* vol. 14, no. 3, pp. 225-264, 1996.

[14] P.J. Braam and P.A. Nelson, "Removing Bottlenecks in Distributed File Systems: Coda and Intermezzo as Examples," *Proc. Linux Expo,* 1999.

[15] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli, and R.Y. Wang, "Serverless Network File Systems," *ACM Trans. Computer Systems,* vol. 14, no. 1, pp. 41-79, 1996.

[16] O. Rodeh and A. Teperman, "zFS—A Scalable Distributed File System Using Object Disks," *Proc. 20th IEEE/NASA Goddard Conf. Mass Storage Systems and Technologies (MSST),* pp. 207-218, 2003.

[17] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "Nfs Version3: Design and Implementation," *Proc. USENIX Technical Conf.,* pp. 137-151, 1994.

[18] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S. Rosenthal, and F.D. Smith, "Andrew: A Distributed Personal Computing Environment," *Comm. ACM,* vol. 29, no. 3, pp. 184-201, 1986.

[19] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. Computers,* vol. 39, no. 4, pp. 447-459, Apr. 1990.

[20] M.N. Nelson, B.B. Welch, and J.K. Ousterhout, "Caching in the Sprite Network File System," *ACM Trans. Computer Systems,* vol. 6, no. 1, pp. 134-154, 1988.

[21] A. Adya, R. Wattenhofer, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, and M. Theimer, "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," *ACM SIGOPS Operating Systems Rev.,* vol. 36, pp. 1-14, 2002.

[22] V. Cate and T. Gross, "Combining the Concepts of Compression and Caching for a Two-Level Filesystem," *ACM SIGARCH Computer Architecture News,* vol. 19, no. 2, pp. 200-211, 1991.

[23] S. Weil, K. Pollack, S.A. Brandt, and E.L. Miller, "Dynamic Metadata Management for Petabyte-Scale File Systems," *Proc. ACM/IEEE Conf. Supercomputing,* 2004.

[24] S. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," *Proc. Seventh Symp. Operating Systems Design and Implementation (OSDI),* 2006.

[25] S. Weil, S.A. Brandt, E.L. Miller, and C. Maltzahn, "Crush: Controlled, Scalable, Decentralized Placement of Replicated Data," *Proc. ACM/IEEE Conf. Supercomputing,* 2006.

[26] R.J. Honicky and E.L. Miller, "Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution," *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2004.

[27] L. Fan, P. Cao, J. Almeida, and A.Z. Brode, "Summary Cache: A Scalable Wide Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking,* vol. 8, no. 3, pp. 281-293, June 2000.

[28] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf, "Performance and Scalability of a Replica Location Service," *Proc. 13th IEEE Int'l Symp. High Performance Distributed Computing (HPDC),* 2004.

[29] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 6, pp. 750-763, June 2008.

[30] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.,* vol. 1, pp. 485-509, 2005.

[31] E. Riedel, M. Kallahalla, and R. Swaminathan, "A Framework for Evaluating Storage System Security," *Proc. Conf. File and Storage Technologies (FAST),* pp. 15-30, 2002.

[32] Y. Zhu and H. Jiang, "False Rate Analysis of Bloom Filter Replicas in Distributed Systems," *Proc. Int'l Conf. Parallel Processing (ICPP),* pp. 255-262, 2006.

[33] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive NFS Tracing of Email and Research Workloads," *Proc. Second Symp. File and Storage Technologies (FAST),* pp. 203-216, 2003.

[34] 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," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* Nov. 2000.

[35] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi., "Towards an Object Store." *Proc. 20th IEEE/NASA Goddard Conf. Mass Storage Systems and Technologies (MSST),* pp. 165-176, Apr. 2003.

[36] B. Welch and G. Gibson., "Managing Scalability in Object Storage Systems for HPC Linux Clusters." *Proc. 21st IEEE/12th NASA Goddard Conf. Mass Storage Systems and Technologies (MSST),* pp. 433-445, Apr. 2004.

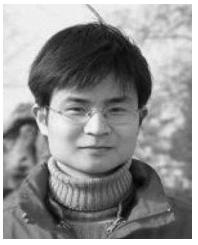
[37] E.L. Miller and R.H. Katz, "Rama: An Easy-to-Use, High-Performance Parallel File System," *Parallel Computing,* vol. 23, pp. 419-446, 1997.
[38] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," *Proc. Ann. Linux Showcase and Conf.,* pp. 317-327, 2000.
[39] N. Nieuwejaar and D. Kotz, *The Galley Parallel File System,* ACM Press, 1996.
[40] A.W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E.L. Miller, "Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems," Technical Report UCSC-SSRC-08-01, 2008.
[41] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *Proc. USENIX Ann. Technical Conf.,* pp. 1-14, 1996.
[42] M. Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Trans. Networking,* vol. 10, no. 5, pp. 604-612, Oct. 2002.
[43] A. Kumar, J. Xu, and E.W. Zegura, "Efficient and Scalable Query Routing for Unstructured Peer-to-Peer Networks," *Proc. IEEE INFOCOM,* 2005.
[44] S. Cohen and Y. Matias, "Spectral Bloom Filters," *Proc. ACM SIGMOD,* 2003.
[45] Y. Zhang, D. Li, L. Chen, and X. Lu, "Collaborative Search in Large-Scale Unstructured Peer-to-Peer Networks," *Proc. Int'l Conf. Parallel Processing (ICPP),* 2007.
[46] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," *Proc. ACM SIGCOMM,* 2006.
[47] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Application of Dynamic Bloom Filters," *Proc. IEEE INFOCOM,* 2006.
[48] B. Xiao and Y. Hua, "Using Parallel Bloom Filters for Multi-Attribute Representation on Network Services," *IEEE Trans. Parallel and Distributed Systems,* vol. 21, no. 1, pp. 20-32, Jan. 2010.
[49] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and Adaptive Metadata Management in Ultra Large-Scale File Systems," *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS),* pp. 403-410, 2008.

Yu Hua received the bachelor's and PhD degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He joined the Department of Computing at the Hong Kong Polytechnic University as a research assistant in 2006. Now he is an assistant professor in the School of Computer Science and Technology at the Huazhong University of Science and Technology, China. His research interests include distributed computing and network storage. He has more than 20 papers to his credit in major journals and international conferences including *IEEE Transactions on Computers (TC)*, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, SC-09, ICDCS-08, ICPP-08, HiPC-06, and ICC-06. He has been on the program committees of multiple international conferences. He is a member of the IEEE.

Yifeng Zhu received the BSc degree in electrical engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1998, and the MS and PhD degrees in computer science from the University of Nebraska, Lincoln, in 2002 and 2005, respectively. He is currently an associate professor in the Department of Electrical and Computer Engineering at the University of Maine. His research interests include parallel I/O storage systems, supercomputing, energy-aware memory systems, and wireless sensor networks. He served as the program chair of IEEE NAS'09, SNAPI'07, guest editor of an special issue of *IJHPCN*, and the program committee of various international conferences, including ICDCS, ICPP, and NAS. He received the Best Paper Award at IEEE CLUSTER'07 and served as the PI or co-PI of several research grants awarded by the US National Science Foundation, including CSR, HECURA, ITEST, REU, and MRI. He is a member of the ACM, the IEEE, and the Francis Crowe Society.

Hong Jiang received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree in computer science from the Texas A&M University, College Station, in 1991. Since August 1991, he has been at the University of Nebraska-Lincoln (UNL), where he served as vice chair of the Department of Computer Science and Engineering (CSE) from 2001 to 2007, and is a professor of CSE. At the UNL, he has graduated 10 PhD students, who upon their graduations either landed academic tenure-track positions (e.g., Stevens Institute of Tech., New Mexico Tech., Univ. of Maine, Univ. of Alabama, etc.) or were employed by major US IT corporations (e.g., Microsoft, Seagate, etc). His present research interests include computer architecture, computer storage systems and parallel I/O, parallel/distributed computing, cluster and grid computing, performance evaluation, real-time systems, middleware, and distributed systems for distance education. He serves as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has more than 170 publications in major journals and international conferences in these areas, including *IEEE-TPDS*, *IEEE-TC*, JPDC, ISCA, MICRO, FAST, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP, etc., and his research has been supported by the US National Science Foundation (NSF), DOD, and the State of Nebraska. He is a senior member of the IEEE and a member of the ACM and ACM SIGARCH.

Dan Feng received the BE, ME, and PhD degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 80 publications to her credit in journals and international conferences, including JCST, FAST, ICDCS, HPDC, SC, ICS and ICPP. She is a member of the IEEE.



Lei Tian received the BE degree in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 2001, and the ME and PhD degrees in computer architecture from the HUST, China, in 2004 and 2010, respectively. His research interests include RAID-structured storage systems, distributed storage systems, and large-scale metadata management. He has more than 20 publications to his credit in journals and international conferences including FAST, MSST, ICS, SC, HPDC, and ICDCS.