Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems

Yu Hua* Yifeng Zhu† Hong Jiang‡ Dan Feng* Lei Tian*‡

*School of Computer
Huazhong Univ. of Sci. & Tech.
Wuhan, China
{csyhua, dfeng, ltian}@hust.edu.cn

†Dept. of Elec. & Computer
Univ. of Maine
Orono, ME, USA
zhu@eece.maine.edu

‡Dept. of Computer
Univ. of Nebraska-Lincoln
Lincoln, NE, USA
{jiang, tian}@cse.unl.edu

Abstract

This paper presents a scalable and adaptive decentralized metadata lookup scheme for ultra large-scale file systems ($\geq$ Petabytes or even Exabytes). Our scheme logically organizes metadata servers (MDS) into a multi-layered 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 algorithm is also developed in this paper for server reconfigurations. This scheme is evaluated through extensive trace-driven simulations and prototype implementation in Linux. Experimental results show that this scheme can significantly improve metadata management scalability and query efficiency in ultra large-scale storage systems.

1 Introduction

Metadata management is critical in scaling the overall performance of large-scale data storage systems [1]. To achieve high data throughput, many systems decouple metadata transactions from file content accesses by diverting large volumes of data traffic away from dedicated metadata servers (MDS) [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 metadata server. While the storage demand increases exponentially in recent years, exceeding petabypes (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 over 50% 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 [6–8] 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.

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 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 easily adopted in distributed systems and allows flexible workload balance among metadata servers.

1.1 Motivations

We briefly discuss the strengths and weakness 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 re-computed to reconstruct the mapping between metadata and their associated servers and accordingly large volume of metadata might 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 di-

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

vide 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 sub-tree 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 sub-directories automatically migrate to other servers with light load. Ceph [24] maximizes the separation between data and metadata management by using a pseudo-random 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 thus might cause slower metadata lookup operations. When an MDS joins or leaves, all directories need to be re-computed to reconstruct the tree-based directory structure, potentially generating a very high overhead in large-scale file systems.

- 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 corresponding bits in that bit 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 actually not. However, the possibility of false positives is controllable and can be very small. Due to the 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 approaches can generate large memory overhead and reduce system scalability and reliability.

As summarized in Table 1 and discussed above, while 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, 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 ultra large-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.

1.2 Contributions

A large-scale distributed file system must provide a fast and scalable metadata lookup service. In large-scale storage systems, multiple metadata servers are desirable for improving scalability. The proposed scheme in this paper, called Group-based Hierarchical Bloom Filter Array (*G-HBA*), judiciously utilizes Bloom filters to efficiently route requests to target metadata servers. Our *G-HBA* scheme extends the current Bloom filter-based architecture by considering dynamic and self-adaptive characteristics in ultra large-scale file systems. Our main contributions are summarized below.

- We present a scalable and adaptive metadata management structure, called *G-HBA*, as shown in Figure 1, to store mass metadata and support fast and accurate metadata lookup in a large-scale file system with N MDSs. The query hierarchy in *G-HBA* consists of four levels: local

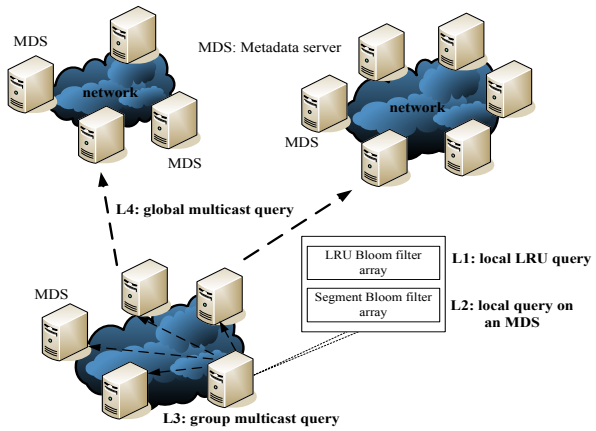


Figure 1: The overall architecture of Group-based Hierarchical Bloom Filter Array (*G-HBA*) for scalable and adaptive metadata management.

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 multi-level file query is designed to be effective and accurate by capturing the metadata query locality and 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 by adaptively and dynamically accommodating the addition and deletion of an MDS while balancing the load and reducing 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 basic question of which metadata sever in an ultra large-scale distributed file system stores the metadata of the queried file, thus helping to quickly access the target file data. Hence, membership queries based on *G-HBA* can in turn support accurate and fast query services for efficient file data management, especially in ultra large-scale distributed file systems.

- We examine the proposed *G-HBA* structure through extensive trace-driven simulations and experiments on our prototype implementation in Linux, in terms of operation latency, replica migration cost and hit rate. Experimental 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 ultra large-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 are given in Section 4 and Section 5, respectively. Section 6 summarizes related work and Section 7 concludes the paper.

2 G-HBA Design

In this section, we present a novel approach, called Group-based Hierarchical Bloom filter Array (*G-HBA*), to carry out scalable and adaptive metadata management and to facilitate fast membership queries in ultra large-scale storage systems.

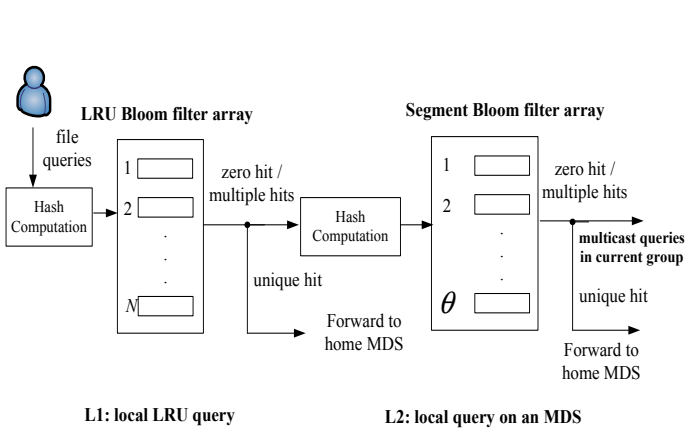
2.1 Dynamic and Adaptive Metadata Management

We utilize an array of Bloom filters on each MDS to support distributed metadata management of multiple MDSs. An MDS where a file's metadata resides is called the *home MDS* of this file. Each metadata server further 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 the client can randomly choose an MDS to perform membership query against its Bloom filter array that includes replicas of the Bloom filters from 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.

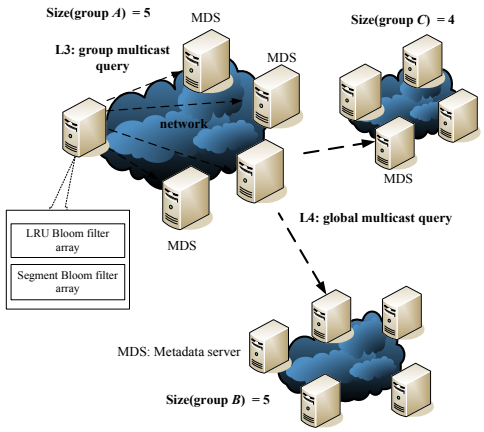
The basic idea behind *G-HBA* in improving scalability and query efficiency is to decentralize metadata management among multiple groups of MDSs. We divide all N MDSs in the system into multiple groups with each group containing at most M MDSs. Note that we represent the actual number of MDSs in a group 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 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.

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. Accord-



(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.

Figure 2: The group-based HBA Architecture allowing the multi-level query.

ingly an MDS has to multicast query requests more frequently to other MDSs, incurring higher network overheads and resulting in longer query delays.

Accordingly, more effective techniques are needed to improve the scalability of the group-based approach. *G-HBA* addresses this issue by taking advantages 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, that 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

Figure 2 shows the 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 multi-level 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. If the query cannot be successfully served at L1, the query is then performed at L2, as shown in Figure 2(a). The Segment BF array (L2) stored on an MDS includes only θ BF replicas, with each replica representing all files whose metadata are stored on that corresponding MDS. Suppose the total number of MDS is n, typically θ is much smaller than n. And we have $\sum_{i=1}^{n} \theta_i = n$ where θ_i is the number 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 Figure 2(b). At L3, all BF replicas available in this group will be checked. At the last level of the query process, i.e., L4, each MDS directly performs lookup by searching its local BF and disk drives. If the local BF responses negatively, the requested metadata is not stored locally on that MDS since the local BF has no false negative. However, if the local BF responses positively, a disk access is then required to verify the existence of requested metadata since the local BF can potentially generate false positive.

2.3 Critical Path in the Multi-level Query Service of G-HBA

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 is 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 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 miss-identification. The probability of a false positive from the segment Bloom filter array of one MDS, f_g^+, is given as below.

$$\begin{aligned} f_g^+ &= \binom{1}{\theta} f_0 (1-f_0)^{\theta-1} \\ &= \theta (0.6185)^{m/n} (1-(0.6185)^{m/n})^{\theta-1} \end{aligned} \quad (1)$$

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

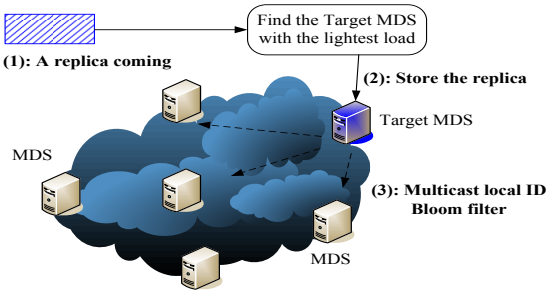


Figure 3: A replica joins the current group.

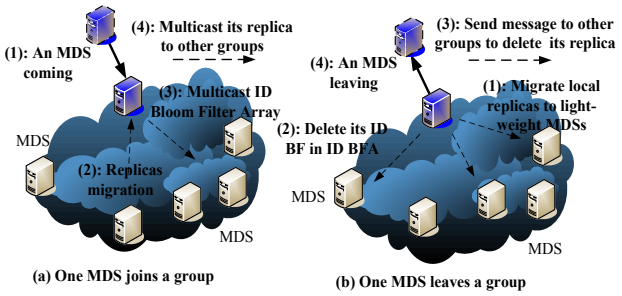


Figure 4: One MDS joins or leaves the current group.

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 Replica

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 often migrate from one MDS to another within a group. 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, as shown in Figure 3. 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 false positive can be extremely low. Counting Bloom filters are used in *IDBFA* to support server departure. 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 0.1KB 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 re-computations 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, incurring forbidden network overheads potentially.

3 Operations and Analysis

In this section, we present our design to support dynamic group reconfiguration and identify the optimal group configuration. Related theoretical background are also presented.

3.1 Light-weight Migration for Group Reconfiguration

Within each group, *IDBFA* can facilitate load balance and support light-weight replica migration during group reconfiguration. Figure 4 shows the process of one MDS joining or leaving a given group. When a new MDS joins the system, it chooses a group that has less than M MDSs and acquires an appropriate amount of BF replicas and off-loads 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 light-weight replica migration and achieve load balance among multiple MDSs of a group.

An MDS departure triggers a similar process but in a reverse direction. It involves (1) migrating replicas previously stored on the 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 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 light-weight migration scheme. This process repeats until no merging can be performed. Figure 5 shows group splitting and merging.

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 multicast 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 is represented by taking into account the latency that includes all delays of actual operations, such as queuing, routing and memory retrieval. Equation 2 shows the function to evaluate the normalized throughput of *G-HBA*.

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

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 Equation 3, which is associated with the numbers of stored replicas on each MDS.

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

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

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

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

The optimal value for *M* thus is the one that maximizes the *Gamma* function in Equation 2.

3.4 Algebraic Operations and Analysis

A number of algebraic operations of Bloom filters are used in *G-HBA* to support group reconfiguration. The following summarizes the fault rate analysis for the algebraic operations in Bloom filter arrays. Due to the space limitation, we cannot present the proofs in this paper but detailed proofs can be found in Ref. [31].

Suppose the length of a Bloom filter is *m* bits and it represents a set *S* with *n* items. In addition, *k* independent hash functions are used. Ref. [30] has shown that the false positive probability is $f_0 = (1-e^{-\frac{kn}{m}})^k$ and this probability is minimum at $(1/2)^k$ or $(0.6185)^{m/n}$, when $k = (m/n)\ln 2$.

S can be represented by a Bloom filter using a mapping relation: $S \rightarrow BF(S)$. We use two Bloom filters $BF(A)$ and $BF(B)$ to represent sets *A* and *B* with the same number of bits and hash functions.

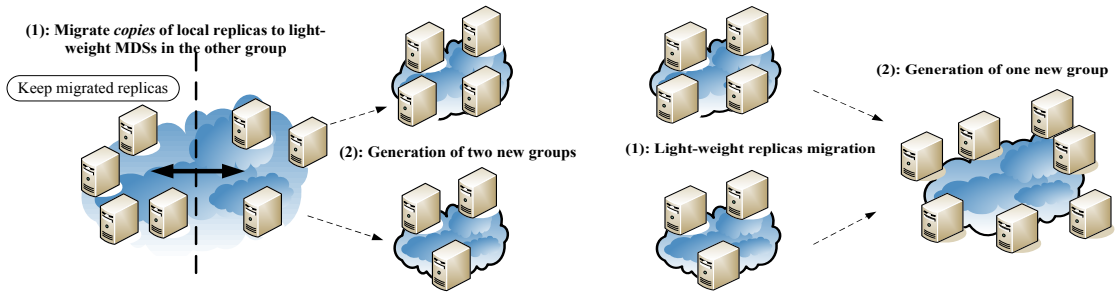
Property 1. *The union of two Bloom filters, $BF(A)$ and $BF(B)$, can be represented as $BF(A \cup B)$ by logical OR operation of their bit vectors.*

Property 2. *The intersection of two Bloom filters, $BF(A)$ and $BF(B)$, can be represented as $BF(A \cap B)$ by logical AND operation of their bit vectors.*

Property 3. *The XOR operation of sets A and B is represented as $A \oplus B = (A-B) \cup (B-A) = (A \cap \overline{B}) \cup (B \cap \overline{A})$.*

It is easy to see that the false positive probability of $BF(A \cup B)$ is larger than that of $BF(A)$ or $BF(B)$. We have also found that the false positive probability of $BF(A \cap B)$ is smaller than that of $BF(A) \cap BF(B)$ with probability $(1-(1-\frac{1}{m})^{k|A-(A\cap B)|})(1-(1-\frac{1}{m})^{k|B-(A\cap B)|})$. In addition, if Bloom filters $BF(A \oplus B)$, $BF(A)$ and $BF(B)$, have the same bits and hash functions, then $BF(A \oplus B) = BF(A-B) \cup BF(B-A)$.

The *XOR* operation based on union and intersection is particularly useful for updating stale replicas in remote MDSs. We can carry out *XOR* operations on local Bloom filter and its replica to examine the number of different bits. If the number is larger than some threshold, we can generate the update messages to replace stale replicas with new ones.



(a) Group splitting (b) Groups merging

Figure 5: The processes of one group splitting and two groups merging.

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 [32]. In order to emulate the I/O behaviors in an ultra large-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 Ref. [29, 33]. 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 *Trace Intensifying Factor* (TIF). The statistics of our intensified workloads are summarized in Table 3 and Table 4. All MDSs are initially populated randomly. Each request can randomly choose an MDS to carry out query operations.

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

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 500GB of storage and was updated last on Aug 9, 2002. 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, that are not related to the metadata operations.

Table 4: Scaled-up HP traces.

	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

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 ultra large-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 overflow the main memory space.

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

In the section, we present the details of identifying the optimal value of group size M by optimizing the normalized throughput of *G-HBA* given in Equation 2. We generate the normalized throughput with the aid of simulation results, including hit rates and latency of multi-level query operations. Other simulation results are directly measured by statistical average values from twenty 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.

Figures 6 shows the normalized throughput of space savings when the number of MDSs is 30 and 100, respectively, under

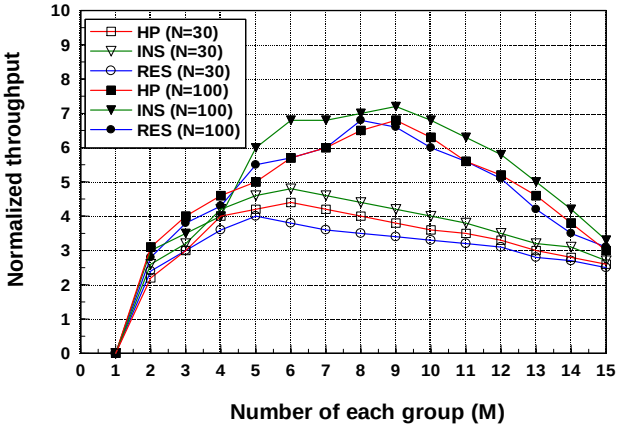


Figure 6: Normalized throughput of *G-HBA* when the total number of MDSs is 30 and 100 MDSs, respectively.

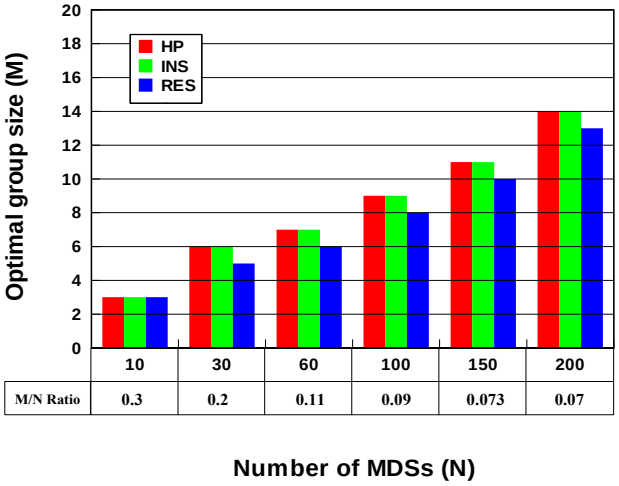


Figure 7: Optimal group size as a function of the number of nodes.

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 these three traces when the number of MDSs is scaled up to 100.

Figure 7 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 ultra large-scale storage systems. It is recommended that some predefined M be used initially and this sub-optimal M be deployed until the total number of MDSs reaches some threshold.

4.2 Average Latency

Figures 8, 9 and 10 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 1.2GB in Figure 8, 800MB

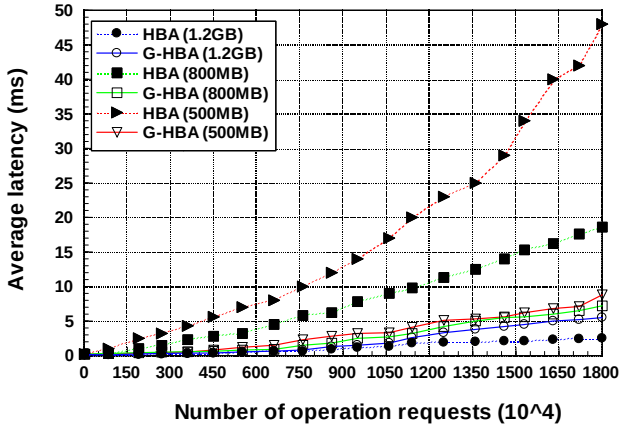


Figure 8: Average latency comparisons of HBA and *G-HBA* with different memory sizes under the HP trace.

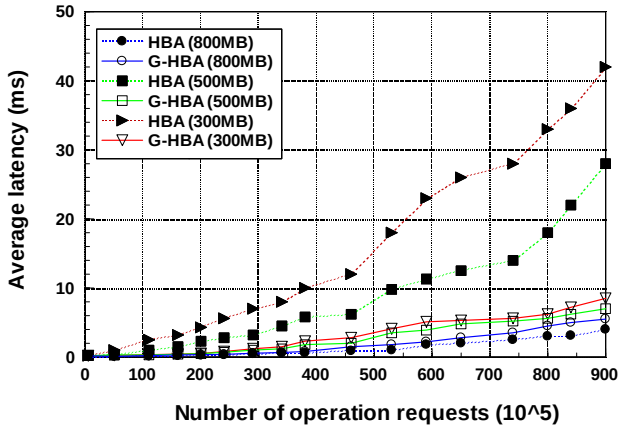


Figure 9: Average latency comparisons of HBA and *G-HBA* with different memory sizes under the RES trace.

in Figure 9 and 900MB in Figure 10, 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, the average latency of the HBA scheme increases rapidly since more disk accesses are involved to store or retrieve BF replicas. 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

Figure 11 shows the overhead of adding a new MDS to the system, in terms of the amount of replica migration traffic, for 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 re-compute the locations (target MDSs) for $(N - M')$ replicas.

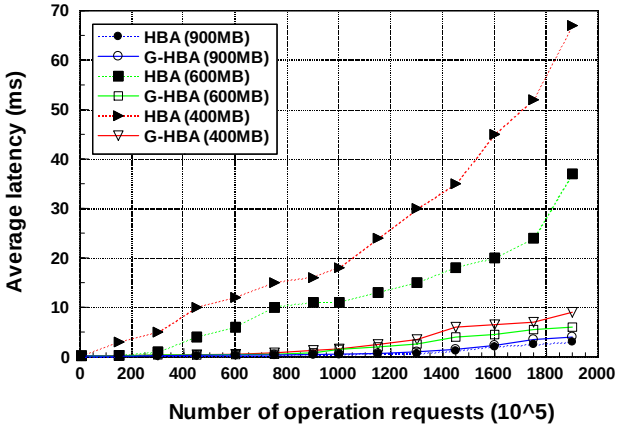


Figure 10: Average latency comparisons of HBA and *G-HBA* with different memory sizes under the INS trace.

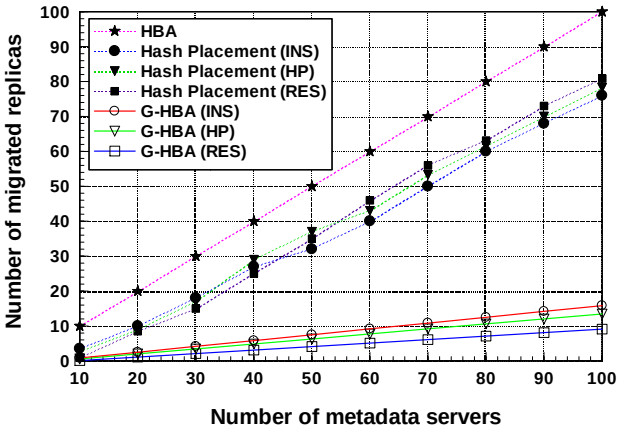


Figure 11: Number of migrated replicas using HBA, hash-based placement and *G-HBA* schemes.

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 ultra large-scale file systems.

4.4 Latency of Updating Stale Replicas

Figures 12 shows the average latency of updating stale replicas under the three traces. 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 Query Hit Rate

Figure 13 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

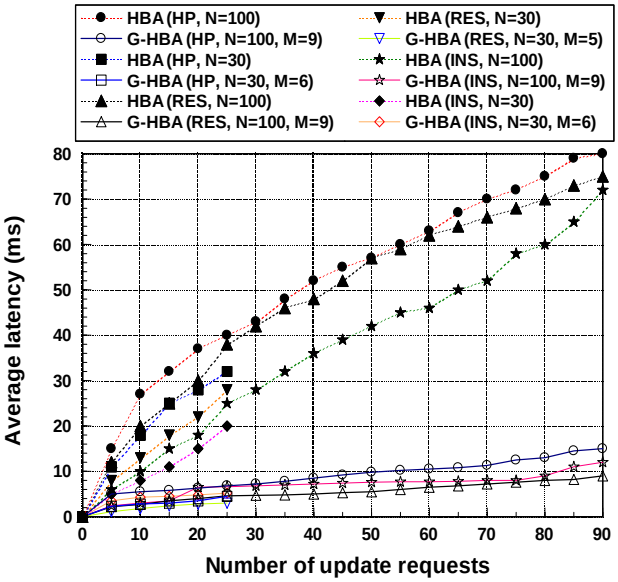


Figure 12: Latency of updating stale replicas of HBA and *G-HBA* schemes using HP, RES and INS traces.

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 shows that more than 80% of query operations can be successfully served by L1 and L2. With the help of L3, more than 90% 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 [33]. The staleness is caused by non-real-time updating in real systems. Here, a false positive happens 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 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 local hits takes place, further checking may involve lookups on disk to conduct lookups on real data. Or else, we definitely know the queried data is non-existing. Although the L4 operations require more costs, the probability is very small as shown in our experiments.

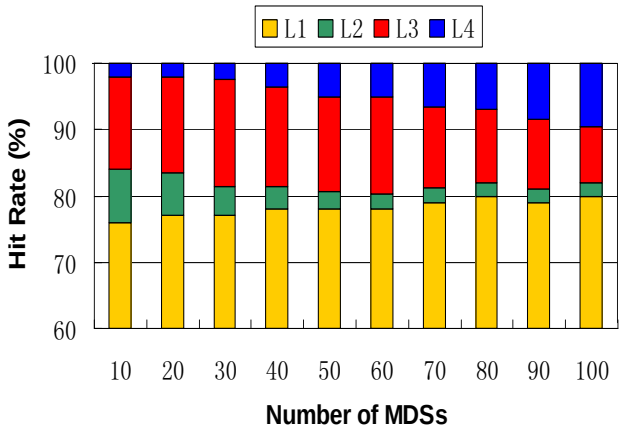


Figure 13: Percentage of queries successfully served by different levels.

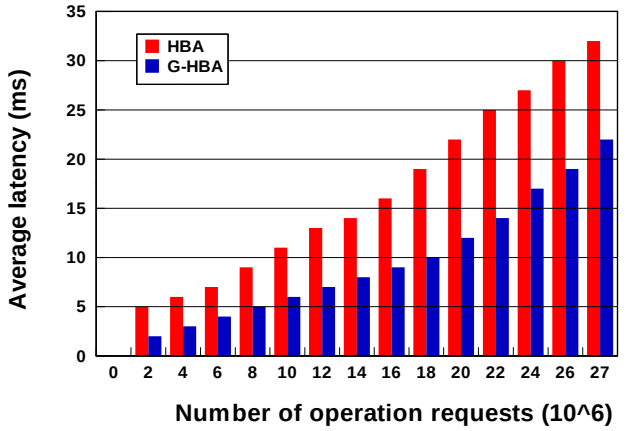


Figure 14: Average query latency using intensified HP traces.

Our design can provide fail-over support when an MDS departs or fails. Heart-beats are exchanged periodically among MDSs within each group. 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.

5 Prototype Implementation and Evaluation

We have implemented the proposed *G-HBA* structure running on a Linux environment that consists of 60 nodes, each equipped with Intel Core 2 Duo CPU and 1GB memory. Each node in the system serves as an MDS. We divide the storage system into groups based on the optimal M value of 7 obtained through the optimal value calculation described in Section 4.1. Thus, each group can maintain at most 7 MDSs. We choose to use the HP traces that are scaled up with a factor of 60 using the scaling approach described in Section 4.

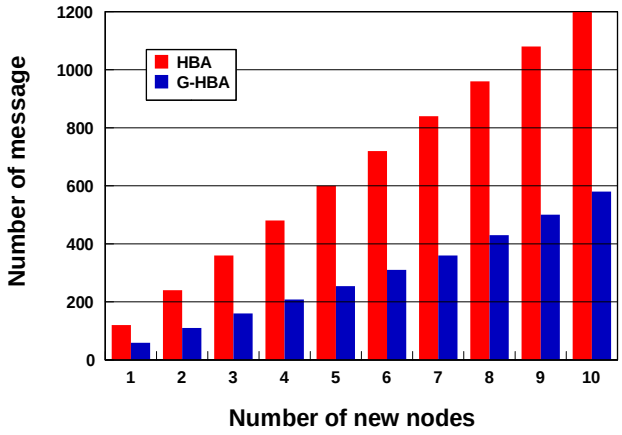


Figure 15: Average number of messages when adding new nodes.

5.1 Lookup Latency

Figure 14 shows the experimental results in terms of query latency under the intensified HP traces. The results from our prototype, consistent with the simulations in Section 4.2, further prove the efficiency of our proposed *G-HBA* structure. *G-HBA* can decrease the query latency of *HBA* by up to 31.2% under the heaviest workload in our experiments, demonstrating its scalability.

5.2 Overhead of Adding MDSs

We evaluate the overhead of dynamic operations for adding new nodes by examining the number of messages generated during the process of an MDS insertion. When adding a new node to a group, the group can directly accept it if there is room. 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. Further, some replicas of the existing MDSs of the same group need to be migrated to the new MDS to keep load balance. In this experiment, we randomly choose a group to add a new node, which may or may not cause the group to be split. Figure 15 shows the number of messages generated during the MDS insertion, averaged over ten insertion operations.

Since each node in the *HBA* scheme maintains a global image of the entire system, an MDS insertion requires it to exchange its own Bloom filter replica with all other MDSs. In contrast, *G-HBA*'s simple and efficient groupe-based operations entail multicasting the BF replica of the new MDS to only one node of each group, achieving significant message savings.

5.3 Memory Overhead Per MDS

We utilize the relative memory requirement normalized to a pure Bloom Filter Array with a bit/file ratio of 8 (BFA8) to facilitate fair comparisons. The basic idea of Bloom Filter Array (BFA) is to build a Bloom filter for each MDS to represent all files stored locally and then replicate this filter to all other MDSs. Thus, 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 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 added to that exploits 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 groupe-based scheme to reduce space overhead and MDS insertion/deletion overhead. Table 5 shows a comparison among BFA8, BFA16, 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 memory overhead decreases as the number of MDSs increases.

Table 5: Relative space overhead normalized to BFA with a ratio of 8 in HP traces.

Server #	BFA 8	BFA 16	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

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], 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. XFS [40] running on large SMPs allows the pervasive use of B^+ tree to increase the scalability of file systems that manage large files, large numbers of files, large directories and fast crash recovery, reducing algorithmic complexity in a file system from linear to logarithmic. GPFS [41] is a fully developed file system by IBM for high-end supercomputers. It utilizes distributed locking and recovery technologies to manage large clusters of up to 512 compute nodes, and 1024 disks to support large scientific applications.

Metadata management in large-scale distributed systems usually provides query services to determine whether the metadata of a specific file resides in a particular metadata server, which in turn helps locate the file itself. Bloom filter, 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 in Bloom filters.

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], distributed Bloom filter [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] supports representation and membership queries based on the multi-attribute dimension. We have developed a novel Parallel Bloom Filters (PBF) and an additional hash table [31] to maintain multiple attributes of items and verify the dependency of multiple attributes, thereby significantly decreasing false positive rates. We have also developed analytical models to accurately estimate false positive and negative rates in Bloom filter replicas [33]. Whenever space is a concern, a Bloom filter can be an excellent alternative to storing a complete explicit list.

7 Conclusion

This paper presents a scalable and adaptive metadata lookup scheme named Group-based Hierarchical Bloom filter Arrays (*G-HBA*) for ultra large-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 limits most of metadata query and Bloom filter update traffic within in a 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 and thus can potentially avoid accessing disks during metadata lookups in exabyte-scale storage systems. 2) *G-HBA* significantly reduces global broadcasts among all MDSs, such as 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 real implementations show that our *G-HBA* is highly effective and efficient in improving the performance, scalability and adaptability of the metadata management component for ultra large-scale file systems.

There are several possible directions for future work beyond the current G-HBA. One is to further improve the query performance by considering the issue of efficiently placing and retrieving memory-resident data [48], since most of the G-HBA accessed MDS location data are residing in memory by design for fast query operations. Another is to enhance the replacement efficiency of our currently used LRU, and consider the distributed and cooperative caching [49–51].

Acknowledgment

This work was supported in part by National Basic Research 973 Program under Grant 2004CB318201, National Natural Science Foundation of China (NSFC) under Grant 60703046, US National Science Foundation (NSF) under Grants 0621493 and 0621526. The work of Lei Tian was done while he was working at the CSE Dept. of UNL.

References

[1] J. Piernas, "The Design of New Journaling File Systems: The DualFS Case," *IEEE Transactions on Computers*, vol. 56, no. 2, pp. 267–281, 2007.

[2] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue, "Efficient metadata management in large distributed storage systems," *MSST*, 2003.

[3] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," *Annual USENIX Technical Conference*, 2000.

[4] L. Guy, P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger, "Replica Management in Data Grids," *Global Grid Forum*, vol. 5, 2002.

[5] S. Moon and T. Roscoe, "Metadata Management of Terabyte Datasets from an IP Backbone Network: Experience and Challenges," *NRDM*, 2001.

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

[7] C. Lukas and M. Roszkowski, "The Isaac Network: LDAP and Distributed Metadata for Resource Discovery," *Internet Scout Project. Online: 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," *The First IEEE Metadata Conference*, 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," *Journal of Network and Computer Applications*, vol. 23, pp. 187–200, 2001.

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

[11] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, 1970.

[12] P.J.Braam, "Lustre whitepaper," *http://www.lustre.org/docs/whitepaper.pdf*, 2005.

[13] P.F.Corbett and D.G.Feitelson, "The vesta parallel file system," *ACM Transactions on 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 & intermezzo as examples," *Proceedings of 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 Transactions on 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," *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pp. 207–218, 2003.

[17] B.Pawlowski, C.Juszczak, P.Staubach, C.Smith, D.Lebel, and D.Hitz, "Nfs version3: Design and implementation," *Proceedings of the Summer 1994 USENIX Technical Conference*, 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," *Communications of the 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 Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[20] M.N.Nelson, B.B.Welch, and J.K.Ousterhout, "Caching in the sprite network file system," *ACM Transactions on 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 Review*, 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," *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, 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," *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[25] S. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, 2006.

[26] R. J. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," *IEEE IPDPS*, April 2004.

[27] L. Fan, P. Cao, J. Almeida, and A. Z.Broder, "Summary cache: a scalable wide area web cache sharing protocol," *IEEE/ACM Trans. on Networking*, vol. 8, no. 3, 2000.

[28] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf, "Performance and scalability of a replica location service," *HPDC*, 2004.

[29] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical Bloom filter arrays (HBA): A novel, scalable metadata management system for large cluster-based storage," *IEEE Cluster Computing, 2004 (Journal version accepted by IEEE TPDS, 2007)*.

[30] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: a survey," *Internet Mathematics*, vol. 1, pp. 485–509, 2005.

[31] Y. Hua and B. Xiao, "A multi-attribute data structure with parallel Bloom filters for network services.," *IEEE HiPC*, pp. 277–288, 2006.

[32] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," *FAST*, pp. 15–30, 2002.

[33] Y. Zhu and H. Jiang, "False rate analysis of Bloom filter replicas in distributed systems," *ICPP*, pp. 255–262, 2006.

[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.," *ACM 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.," *MSST*, pp. 165–176, Apr. 2003.

[36] B. Welch and G. Gibson., "Managing scalability in object storage systems for hpc linux clusters.," *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, 1997.

[38] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," *Proceedings of the 4th Annual Linux Showcase and Conference*, pp. 317–327, 2000.

[39] N. Nieuwejaar and D. Kotz, *The galley parallel file system*. ACM Press New York, NY, USA, 1996.

[40] S. A., D. D., W. Hu, A. C., N. M., and P. G., "Scalability in the XFS file system.," *In Proceedings of the USENIX 1996 Technical Conference*, pp. 1–14, 1996.

[41] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters.," *Proceedings of the Conference on File and Storage Technologies (FAST02)*, pp. 231–244, 2002.

[42] M. Mitzenmacher, "Compressed Bloom filters," *IEEE/ACM Trans. on Networking*, vol. 10, no. 5, pp. 604–612, 2002.

[43] A. Kumar, J. Xu, and E. W. Zegura, "Efficient and scalable query routing for unstructured peer-to-peer networks," *INFOCOM*, 2005.

[44] C. Saar and M. Yossi, "Spectral Bloom filters," *SIGMOD*, 2003.

[45] Y. Zhang, D. Li, L. Chen, and X. Lu, "Collaborative Search in Large-scale Unstructured Peer-to-Peer Networks," *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," *SIGCOMM*, 2006.

[47] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network application of dynamic Bloom filters," *INFOCOM*, 2006.

[48] J. Pisharath, A. Choudhary, and M. Kandemir, "A window-based approach to retrieving memory-resident data for query execution," *IDEAS*, pp. 283–288, 2004.

[49] M. Korupolu and M. Dahlin, "Coordinated placement and replacement for large-scale distributed caches," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 6, pp. 1317–1329, 2002.

[50] M. Korupolu, C. Plaxton, and R. Rajaraman, "Placement algorithms for hierarchical cooperative caching," *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 586–595, 1999.

[51] Y. Zhu and H. Jiang, "RACE: A Robust Adaptive Caching Strategy for Buffer Cache," *to appear in IEEE Transactions on Computers*, no. 1, January 2008.