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 University of Science and Technology
Wuhan, China
{csyhua, dfeng, ltian}@hust.edu.cn

[‡]Dept. of Elec. & Computer
University of Maine
Orono, ME, USA
zhu@eece.maine.edu

[§]Dept. of Computer
University 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 MDS 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. Our 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 file location and 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 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 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]. Thus, scalable and decentralized metadata management schemes 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-structured global directory. The probabilistic approach employs lossy data representations, such as Bloom filters [6], 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.

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 our previous Bloom filter-based architecture by considering dynamic and self-adaptive characteristics of ultra large-scale file systems.

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 Bloom 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 from all other MDSs. 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 original metadata can be stored only in one MDS, multiple hits, meaning that original metadata are found in multiple MDSs, indicate a query miss.

The basic idea behind *G-HBA* in improving scalability and query efficiency is to decentralize metadata manage-

1063-6927/08 $25.00 © 2008 IEEE
DOI 10.1109/ICDCS.2008.32

ment 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 high scalability and space efficiency. These structures are replicated among MDS groups and each group contains approximately the same amount of replicas for load balance. 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.

The rest of the paper is organized as follows. Section 2 presents the basic scheme of *G-HBA* and its design issues. The performance evaluation based on trace-driven simulations and prototype implementation are given in Section 3 and Section 4, respectively. Section 5 summarizes related work and Section 6 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, specifically to facilitate fast membership queries in ultra large-scale storage systems.

2.1 Group-based HBA Scheme

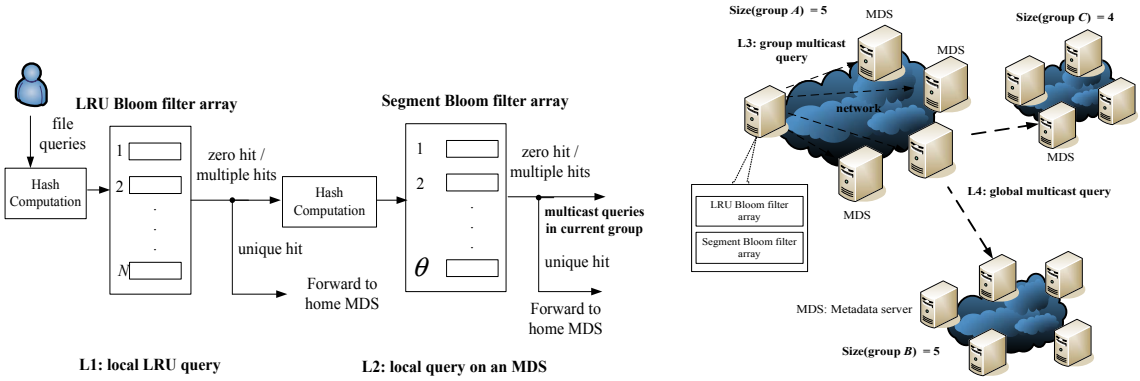
Figure 1 shows the overall scheme of our *G-HBA*. A query process at one MDS may involve four hierarchical levels: (1) searching the locally stored LRU BF Array (L1), (2) searching the locally stored Segment BF Array (L2), (3) multicasting to all MDSs in the same group to concurrently search all Segment BF Arrays stored in this group (L3), and (4) multicasting to all MDSs in the system to directly search requested metadata (L4). The multi-level metadata query is designed to judiciously exploit access locality and dynamically balance load among MDSs.

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 1(a). The segment BF array (L2) stored on 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, and typically θ_i is much smaller than N. And we have $\sum_{i=1}^{M'} \theta_i = N$. 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 1(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 responds negatively, the requested metadata is not stored locally on that MDS since the local BF has no false negatives. 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 positives.

2.2 Critical Path

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 MDS 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 segment Bloom filter array, which maintains mapping information for a fraction of the entire storage system by approximately storing $\theta = \lfloor \frac{N-M'}{M'} \rfloor$ replicas. A unique hit in any L2 segment 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 $f_g^+ = C_\theta^1 f_0(1-f_0)^{\theta-1} = \theta(0.6185)^{m/n}(1-(0.6185)^{m/n})^{\theta-1}$, 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 [7]. 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 re-



(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 1. The group-based HBA architecture allowing the multi-level query.

quest 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.

2.3 Updating Replica

Updating stale Bloom filter replicas involves two steps, replica identification (localization) and replica content update. Within each MDS 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, 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. 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. 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.

G-HBA does not use modular hashing to determine the placement of the newest replica within one MDS group. A 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.

2.4 Light-weight Migration

Within each group, *IDBFA* can facilitate load balance and support light-weight replica migration during group reconfiguration. When a new MDS joins the system, it first joins a group that has less than M MDSs, then acquires an appropriate amount of BF replicas, and finally 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 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 MDS, 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.

2.5 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 multicasts are used to resolve these cases and a mul-

ticast typically takes longer when more hops are involved. In what follows we discuss how to find the optimal M.

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 1 shows the function to evaluate the normalized throughput of *G-HBA*.

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

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

$$U_{G\text{-}HBA}(space) = \frac{N-M}{M} \quad (2)$$

We then examine the operation latency, shown in Equation 3 for *G-HBA*, by considering multi-level hit rates that may lead to different delays. Definitions for the variables used in Equation 3 are given in Table 1.

$$\begin{aligned} U_{G-HBA}(laten.) = {} & D_{LRU} + (1-P_{LRU})D_{L2} + \\ & (1-P_{LRU})(1-\frac{P_{L2}}{M})D_{group} + \\ & (1-P_{LRU})(1-\frac{P_{L2}}{M})^{M} D_{net.} \end{aligned} \quad (3)$$

Table 1. 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 1.

3 Performance Evaluation

We examine the performance of *G-HBA* through trace-driven simulation and compare it with HBA [8], 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 [9]. 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. [8, 10]. 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 2 and Table 3. All MDSs are initially populated randomly. Each request can randomly choose an MDS to carry out query operations.

Table 2. 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

Table 3. Scaled-up HP traces.

	Original	TIF=40
requests (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

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 these 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.

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 [8] 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.

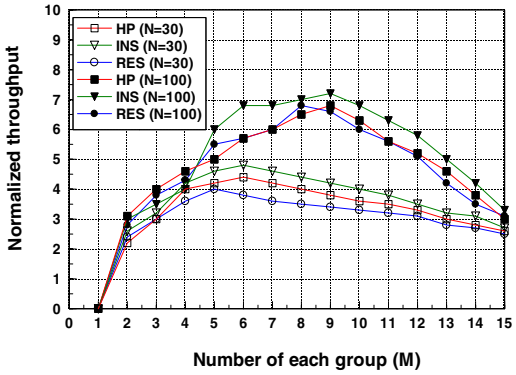


Figure 2. Normalized throughput of *G-HBA* when the total number of MDSs is 30 and 100 MDSs, respectively.

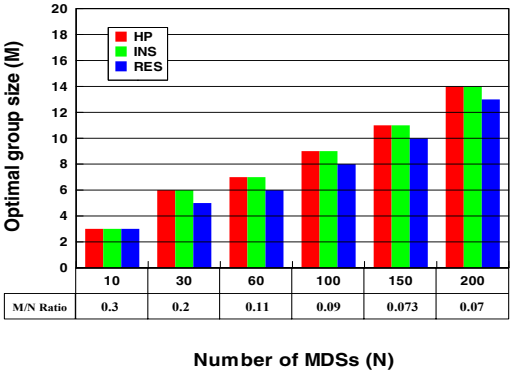


Figure 3. Optimal group size as a function of the number of nodes.

3.1 Impact of Group Size M on 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 Equation 1. 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 have a 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 2 shows the normalized throughput of space savings when the numbers of MDSs are 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 when the number of MDSs is scaled up to 100.

Figure 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 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 changes and reaches some threshold.

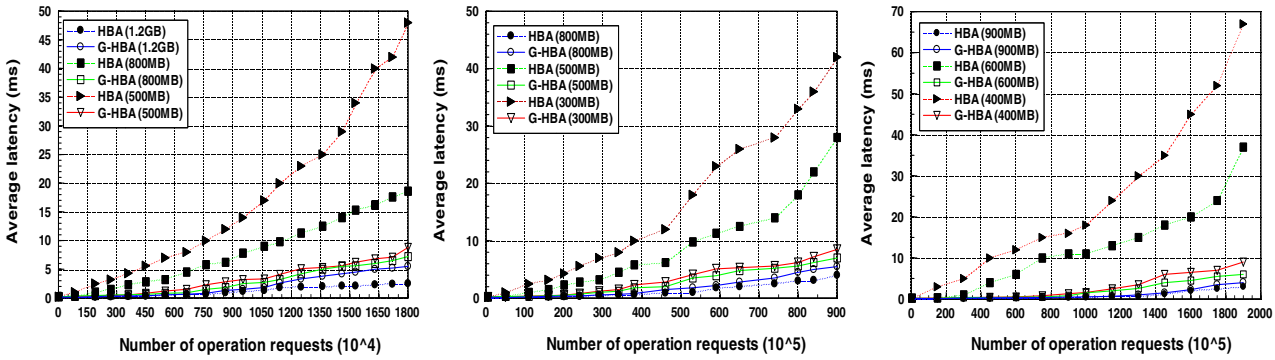
3.2 Average Latency

Figures 4(a), 4(b) and 4(c) 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 4(a), 800MB in Figure 4(b) and 900MB in Figure 4(c), 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.

3.3 Overhead of MDS Group Reconfiguration

Figure 5 shows the overhead of adding a new MDS to the system, in terms of the amount of replica migration traffic, for HBA, hash 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 placement, as discussed in Section 2.3, needs to re-compute 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 approximate $\frac{N-M'}{M'+1}$ replicas



(a) HP trace. (b) RES trace. (c) INS trace.

Figure 4. Average latency comparisons under three traces.

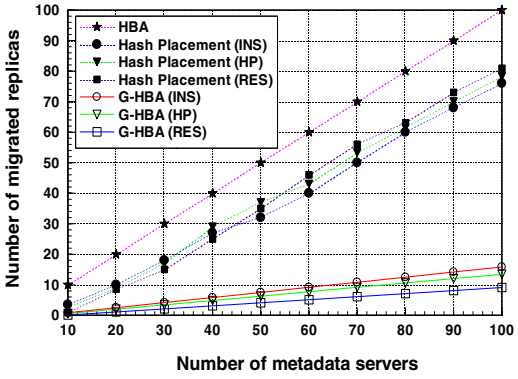


Figure 5. Number of migrated replicas.

to the newly inserted MDS and thus significantly reduces network overheads in ultra large-scale file systems.

3.4 Latency of Updating Stale Replicas

Figures 6 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.

3.5 Query Hit Rate

Figure 7 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 2.2. 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

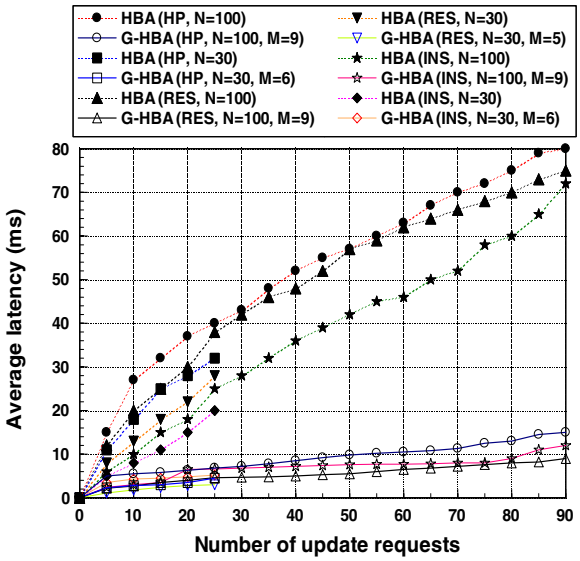


Figure 6. Latency of updating stale replicas of HBA and *G-HBA* schemes.

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 [10]. 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 the entire system.

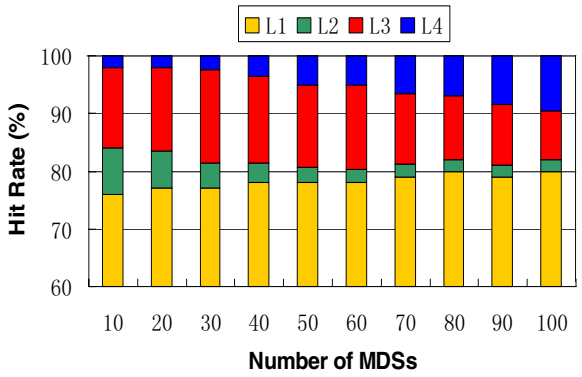


Figure 7. Percentage of queries successfully served by different levels.

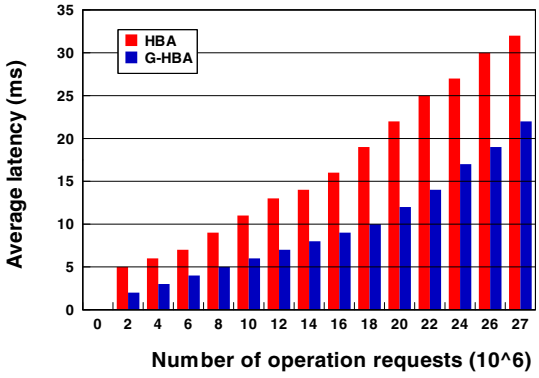


Figure 8. Average query latency using intensified HP traces.

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 take 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 of such operations is very small as shown in our experiments.

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.

4 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 Sections 2.5 and 3.1. 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 3.

4.1 Lookup Latency

Figure 8 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 3.2, 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.

4.2 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 Bloom 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, exploiting 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. Table 4 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 metadata servers. 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.

5 Related Work

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 and pure hashing are two common techniques used for managing metadata, including NFS [11], Coda [12]

Table 4. 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

and RAMA [13]. However, they suffer from concurrent access bottlenecks. Due to space limitation, other details can refer to our technical report [14].

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. Standard Bloom filters [6] have inspired many extensions and variants, including the counting Bloom filters [15], Multi-Dimension Dynamic Bloom Filters (MD-DBF) [16] and Parallel Bloom Filters (PBF) [17]. Whenever space is a concern, a Bloom filter can be an excellent alternative to storing a complete explicit list.

6 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 logical 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 an MDS group. Compared with HBA, *G-HBA* is more scalable. Extensive trace-driven simulations and real prototype implementations show that our *G-HBA* is highly effective and efficient in improving the performance, scalability and adaptability of the metadata management for ultra large-scale file systems.

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 and 60503059, US National Science Foundation (NSF) under Grants 0621493 and 0621526, the Program for New Century Excellent Talents in University NCET-04-0693 and NCET-06-0650, and HUST-SRF No.2007Q021B. The work of Lei Tian was done in part 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," *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 290–298, 2003.

[3] D. Roselli, J. Lorch, and T. Anderson, "A comparison of file system workloads," *USENIX Annual Technical Conference*, pp. 41–54, 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," *Workshop on Network-Related Data Management (NRDM)*, 2001.

[6] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

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

[8] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical Bloom filter arrays (HBA): A novel, scalable metadata management system for large cluster-based storage," *IEEE International Conference on Cluster Computing*, pp. 165–174, 2004.

[9] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," *Conference on File and Storage Technologies (FAST)*, pp. 15–30, 2002.

[10] Y. Zhu and H. Jiang, "False rate analysis of Bloom filter replicas in distributed systems," *International Conference on Parallel Processing (ICPP)*, pp. 255–262, 2006.

[11] B.Pawlowski, C.Juszczak, P.Staubach, C.Smith, D.Lebel, and D.Hitz, "NFS version3: Design and implementation," *USENIX Summer Conference*, pp. 137–151, 1994.

[12] 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.

[13] E. Miller and R. Katz, "RAMA: An easy-to-use, high-performance parallel file system," *Parallel Computing*, vol. 23, no. 4-5, pp. 419–446, 1997.

[14] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems," *Technical Report, University of Nebraska-Lincoln*, 2007.

[15] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[16] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network application of dynamic Bloom filters," *IEEE International Conference on Computer Communications (INFOCOM)*, 2006.

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