Efficient Access Control for Distributed Hierarchical File Systems

Kristal T. Pollack
kristal@cs.ucsc.edu

Scott A. Brandt
scott@cs.ucsc.edu

University of California, Santa Cruz

Abstract

To determine whether a user can access a file in a hierarchical file system a traversal of the directory hierarchy is required in order to check access control for all the parent directories. This traversal can be especially expensive in a distributed system where the files may be on separate devices. We present two approaches for representing the complete access control for a file and its parent directories such that it can be stored locally with each file in order to avoid traversal. We use the well-known CNF and DNF (Conjunctive and Disjunctive Normal Form) formats to store permission and ownership information compactly for the entire path to a file. An examination of the structure of an existing large shared file system demonstrates the efficacy of our solution.

1. Introduction

In a traditional UNIX-like file system access to a file is governed not only by the file's permissions, but by the hierarchy of permissions of all of the directories above it. Access control in these systems consists of both permissions (read, write, execute) and ownership (user, group) information, located in the metadata for each file. Calculating these access controls for a file requires the traversal of the directory hierarchy. The traversal of a directory hierarchy can be expensive, especially in the case of distributed file systems where traversal may require network trips between nodes. This paper proposes a way to preserve the hierarchical access controls for a distributed file system without requiring a traversal of the directory tree by compressing the access controls locally for each file. This idea builds upon the dual-entry access control lists proposed in the Lazy-Hybrid (LH) metadata management approach [2].

Requiring that all users of a large-scale distributed file system access one machine for all metadata operations is an obvious bottleneck. For this reason distribution of metadata is crucial for the overall scalability of such a system [2] [8]. In addition, large distributed systems should not have
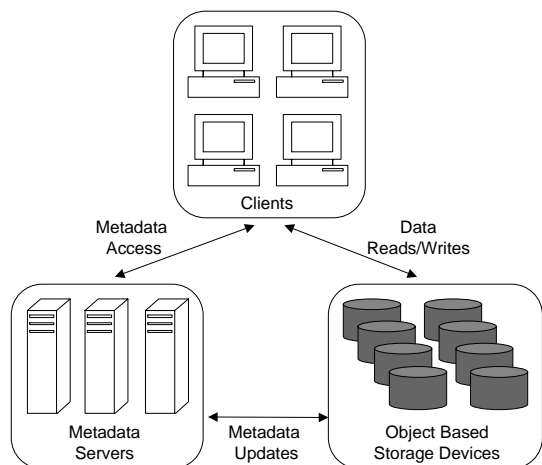

Figure 1. Distributed Object Storage system architecture

a single point of failure. Large distributed storage systems should therefore not rely on having one machine store all of the metadata. It will be necessary to have multiple machines that can perform this task. Figure 1 shows the architecture that has been emerging in many of the large distributed storage systems [2] [6]. Clients receive metadata from a cluster of specialized metadata servers which enables them to access the disks directly and perform I/O operations such as reads and writes.

To avoid unbalanced load or hot-spots on one of the metadata servers a hash-based approach has been used in many similar systems to distribute the metadata uniformly across the metadata servers. Metadata is typically distributed based on a hash of the full path of the file or some combination of the directory path and the file name. There are many benefits to having a hash-based location for a distributed file system, not only for workload balance but also for metadata location. By storing metadata based on the hash of the file name the clients of the system can quickly locate metadata for a file by applying the hash function on

the full path name. A previous study on hierarchical file systems showed that even for a non-distributed system, file location made up a high percentage of all the memory accesses in order to traverse the directories required to locate a file [4]. For distributed systems this is especially significant, since a directory traversal will likely require accessing multiple nodes in the distributed system. However, similar to file location, determining access control for files typically requires a directory traversal since it is necessary to satisfy the permissions for all directories above a file in the directory tree (these are the types of systems we will consider for this paper). This poses a significant problem for hash-based systems in that the performance gain of hashed-based location would be lost if the directory hierarchy still had to be traversed in order to calculate the access control for a file. Recent distributed file systems have lacked the ability to correctly preserve these hierarchical permissions without requiring a directory traversal and still harness the benefits of hash-based location schemes.

This paper proposes two methods that can be used to store access control information by compressing the permissions and ownership metadata for the entire path of a file into a single compact representation and storing it locally with the metadata for each file. This allows the complete access control information for each file to be accessed when the metadata for the file is accessed, without requiring a traversal of the directories in the path of the file. We look at representing access control as a conjunction of disjunctions (CNF) and as a disjunction of conjunctions (DNF). Since the path of a file can be of arbitrary size, so can the permission and ownership information needed to represent the correct access control for the file. However, with this information in CNF or DNF format it is simple to apply a number of logical rules to simplify the expressions into a more compact representation. We present a worst case analysis for the computation and size of the representation for both methods. We also present results from analyzing the metadata of an existing large shared file system and show that in practice our representation is quite compact. For a system with approximately 4 million files, 1500 unique users, and a maximum tree depth of 32, we can represent access control for 31% of the files with no additional information (empty set), 68% with just one user, group or user/group pair, and less than 1% with two users, groups or user/group pairs.

2. Related Work

This work is general enough to apply to any file system with hierarchical access controls but it is most relevant to distributed file systems. A problem that arises in distributed file systems is how to maintain UNIX-like hierarchical access. Previous solutions have either involved a potentially expensive directory traversal or have not supported it at all. To our knowledge no published work attempts to compress the permissions and ownership of files to avoid traversal.

One approach for a distributed file system is to distribute data by directories. This stores entire directory subtrees on separate machines, much like NFS [11], Lustre [1] and Dynamic Partitioning [13]. These systems require the traversal of directories for locating files, as well as checking ownership and permissions. Compressing the permissions and ownership as we are proposing does not benefit these systems a great deal since they must traverse the directory hierarchy regardless. However, this subtree partitioning approach for distributed systems suffers from load balancing issues that can grow more severe as the file system ages.

Another approach for distributing metadata is to use a hashing scheme where files are distributed based on a hash of some file identifier. This helps to balance the load among the devices and helps to avoid hot-spots associated with a popular directory of files. It also prevents a popular directory from becoming a bottleneck since the directory file does not have to be read every time a file underneath it is accessed. Overall this method provides an efficient approach for locating and allocating metadata since it does not require a traversal of the directory hierarchy. Many current systems have adopted this approach such as Lazy-Hybrid [2], HAP [14], Vesta [3] and RAMA [9]. The drawback to this approach is that in order to preserve hierarchical access control models the directory hierarchy is still traversed. The cost of this traversal varies depending upon where the ownership and permission information is stored in the system.

Systems such as HAP store ownership and permission information on metadata servers. Determining correct access control requires that the metadata for every directory above the file in the directory hierarchy be accessed. Due to the hash-based allocation method, the metadata for these directories may reside on different metadata servers. So each of these metadata servers would have to be contacted in order to determine the correct access control for a single file.

RAMA stores ownership and permission information with the object it describes. Therefore to check this information the location of the data needs to be determined and it must visit where the actual data is stored to read the metadata to check if access should be granted. Not only does this incur overhead in checking possibly distributed directories, but this could also pose a security issue in that the location of the stored object is given to the client before it is determined if the client is allowed to access that object.

Vesta stores metadata on metadata servers as well. The developers conceded that checking directory permission bits for every level of the directory hierarchy was a problem in Vesta [3]. Instead of traversing directories to preserve hi-

erarchical access control, they simply chose not to support it.

3. Lazy-Hybrid Metadata Management

This research is targeted at improving the Lazy-Hybrid approach [2], though it may be generalized to any file system that has hierarchical access control. The Lazy-Hybrid solution is a hashed-based technique, hashing the full path name of a file to obtain a unique identifier. Each metadata server is assigned a range of the hash values to maintain, and this assignment is stored in a global table. When a client wants to access a file the client simply hashes the full path of the file and determines which metadata server to contact based on the global table. If a metadata server is added or removed the global table is updated to reflect the change.

In order to avoid path traversal while calculating the permissions for files, the Lazy-Hybrid (LH) method uses a dual-entry access control list to manage access to a file. This stores the *path permissions* and the *file permissions* for a file. The path permissions are the intersection of the parent directory's path permissions and the file permissions of the current file. If the permissions of the parent directory or the file change the access control of the file can be recomputed by taking the intersection of the parent's path permissions and the file's permissions. In this way the access control hierarchy is preserved for the permission bits and only requires traversal in the case of a directory permission change, and even this type of operation can be postponed with logging. LH avoids directory traversal for permission bits, but does not consider ownership information stored along the hierarchy. This is commonly overlooked and was similarly neglected in the Logic File System [10]. It is important to preserve the ownership information with the permission bits as the permission bits for a file directly correspond to the user and group that have ownership of the file. The next section shows why preserving both the permission bits and the ownership information of all the files is important and outlines a clean solution.

4. Design

In order to avoid directory traversal while maintaining hierarchical access controls we need to store access control information for the parent directories of a file along with those of the file itself. A method for storing the permission bits compactly was given in the Lazy-Hybrid method [2]. However, we cannot just store information about permission bits of each parent directory since these permission bits only define the access rights of the specific user and group that have ownership of that file. Therefore this ownership information needs to be stored as well.
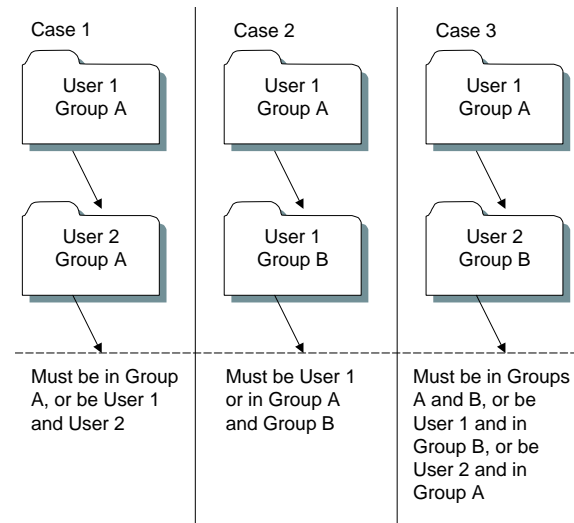

Figure 2. Three cases where it is necessary to know the access control of parent directories. For these cases the permission bits of all the files are set to allow access to the user and the group.

Figure 2 shows three general cases where storing the ownership information for parent directories is necessary for correctness. We will discuss later how often cases like these come up in a real system. Case 1 shows the situation in which the user that has ownership is different for the parent directory and the child directory while the group is the same. Access to the child directory and all of its sub-files will be restricted to any member of the group or a user that is the user for the parent directory and the child directory. However, since a user cannot be both User 1 and User 2 at the same time, access is simply restricted to the group. Case 2 shows the situation in which the group that has ownership is different for the parent directory and the child directory while the user is the same. Access to the child and all of its sub-files will be restricted to the user that has ownership and any user that is a member of both the group that owns the parent and the group that owns the child. Case 3 is the situation in which both the user and the group that have ownership of the parent directory and the child directory are different. This is effectively a combination of the two previous cases. Access to the child and all of its sub-files is restricted to a user that: is a member of both the group that owns the parent and the group that owns the child, owns the child directory and is a member of the group that owns the parent directory, or owns the parent directory and is a member of the group that owns the child directory. We can express these access requirements concisely as some combination of "and"s and "or"s, as shown at the bottom of Figure 2.

000	$\bot$
001	$\neg u \wedge \neg g$
010	$\neg u \wedge g$
011	$\neg u$
100	u
101	$u \vee \neg g$
110	$u \vee g$
111	$\top$

Since we can express the access control requirements as a series of users and groups joined by "and"s and "or"s we can express them in Conjunctive Normal Form (CNF, "and"s of "or"s) or Disjunctive Normal Form (DNF, "or"s of "and"s). This is a straightforward and convenient representation that is commonly used when dealing with predicates. In order to determine which users and groups belong in these expressions we simply need to record the user and group settings for the execute permission bit (the "x" bit in UNIX). We only care about the execute bit because it is the only permission that carries down recursively, determining whether or not a user can access a file below a directory. Table 1 shows each possible setting for the execute permission and the corresponding predicates that represent the correct access control. In other words, what we have done is merged the ownership and permission information into a single expression. It is somewhat similar to the concept of access control lists used in other prominent file systems, and in fact, we can use this representation to express those as well.

4.1. CNF

Storing the access control information in CNF format is very straightforward. We can store the user and group at each level as a disjunctive clause (user or group) depending on the corresponding execute bits, as shown in Table 2. This is a fairly intuitive representation since visiting each clause is like visiting each directory and verifying that the user meets the requirements to access that directory. Table 1 shows the subsequent user/group combinations for each possible permission mask. There are two cases for which we must store the information in two singular clauses, when the user and the group are denied access (001) and when only the group is allowed access (010). Each clause represents one of the requirements a user must satisfy in order to access a file. If a user satisfies
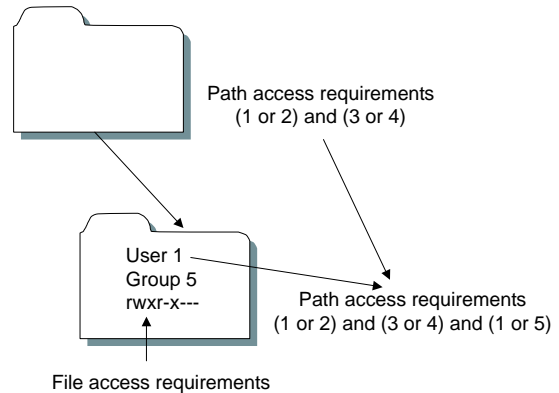

Figure 3. Calculating the path access requirements for a file.

Table 2. CNF for the cases shown in Figure 2

Case 1	$(A \vee 1) \wedge (A \vee 2)$
Case 2	$(1 \vee A) \wedge (1 \vee B)$
Case 3	$(A \vee 1) \wedge (B \vee 2)$

each clause then the user has access rights to the file. This is similar to the simple idea of storing the ownership and permission data for each of the parent directories in a list and checking that a requirement is satisfied for each directory before granting permission. Our method of storing the ownership and permission data is much more efficient as it merges these two sources, thereby eliminating unnecessary information. We will show later that once we have the access control in this CNF representation we can further compact the representation by using some additional logic.

We can use this CNF representation to represent the access control for an entire hierarchical file system by using a strategy very similar to the dual-entry access control list used in the Lazy Hybrid method [2]. For each file we have *file access requirements* and *path access requirements*. The file access requirements are simply the ownership information and the permission bits that would normally be stored with the metadata for a file in a file system. The path access requirements are a CNF representation of the file access requirements combined with the path access requirements of all the parent directories above it. Since the path access requirements are stored for every file in the directory tree, only the path access requirements of the immediate parent of the file need to be merged with the file access require-

Table 3. DNF for the cases shown in Figure 2.

Case 1	$(1 \wedge 2) \vee (A)$
Case 2	$(A \wedge B) \vee (1)$
Case 3	$(1 \wedge 2) \vee (A \wedge 2) \vee (B \wedge 1) \vee (A \wedge B)$

ments in order to calculate the file's path access requirements. This operation is illustrated in Figure 3.

During this operation, or afterwards, a few simple rules can be applied to our CNF expressions to further reduce them. For instance, we can check previous clauses in our expression when we are merging the path and file access requirements to see if we can eliminate any redundant information. If the clause we are about to store is a subset of an existing clause we can remove the existing one, as the current clause is more restrictive and eliminates the need to store the less restrictive clause since every clause must be satisfied for access to be granted. This is especially useful for the common pattern of permissions getting more restrictive deeper in the hierarchy. We can further compress the path access requirements for a file if we can distinguish between users and groups once they are stored. This should be fairly easy to do in implementation. The reason this helps us is because it is impossible for a user to be two users at once, i.e., a user A cannot be user A and user B at the same time. In the case where a user has exclusive rights to a file, a clause with only that user will be added. As long as this clause exists, only the user specified in it can access the current file or any of its sub-files. This is true because in CNF representation a user must satisfy every clause to obtain access. Therefore, when a clause with a single user in it is added we can simplify our representation by applying the rule that a user cannot be two different users at once. If there ever exists two singular clauses that contain two different users then we know the file is unreachable for any user that is not a super-user of the system. However, if no singular user clause exists, when we add one to the path access requirement we can remove all the users from the other clauses and any future clauses that contain a both a user and group. We can do this because other users in the path access requirement are irrelevant since the user must satisfy the singular user clause to obtain access. Note, we cannot do the same for singular group clauses since a user can be a member of more than one group at the same time. Additional rules can be applied to these expressions to further reduce them at the cost of adding complexity to the calculation of access control requirements.

4.2. CNF vs. DNF

The approach proposed for composing access controls using CNF is quite similar to DNF, so for the sake of brevity we have omitted the details for DNF. In order to compare the effectiveness of the CNF and DNF representations for access control it is necessary to consider: the amount of space required to store these access requirements, the time it takes to grant or deny access to a user, and the time it takes to locally update path access requirements. The time it takes to calculate path access requirements depends on how complex of an implementation we want in order to further compact the result. Both approaches will run in linear time with respect to the size of the parent path access requirement.

Due to the somewhat opposite representations of CNF and DNF, the performance of these two approaches for granting or denying user access is inversely proportional. When verifying that a user can access a file the DNF method may be faster on average. This is because all of the clauses may not have to be checked in the DNF representation. If a user satisfies any clause the user will be granted access. For CNF all the clauses must be checked since a user must satisfy all clauses to be granted access. However, when denying a user access to a file the CNF method may be faster on average. For DNF every clause would have to be checked to determine that a user cannot obtain access. For CNF a user can be denied at any point a clause is encountered that they do not satisfy. Therefore it is not obvious which approach would perform better in terms of computation when checking access rights in general. However, it can be said that for both of approaches the time to check access rights will be bounded linearly by the size of the path access requirement. The size of the path access requirement is in turn bounded by the length of the path to the file, as with standard permissions, but the average case is much smaller and all checks can always be performed in a single metadata server.

The storage requirements for the CNF and DNF approaches are somewhat opposite as well. Figure 4 shows the worst cases for each of them, Scenario 1 is the worst case for DNF and Scenario 2 is the worst case for CNF. However, for Scenario 1, the worst case for DNF, using the CNF approach we get the optimal representation. Likewise, for Scenario 2, the worst case for CNF, using the DNF approach we get the optimal representation. Therefore which solution is better really depends on the system that it will be used on. In order to make a general estimate we can look at the upper bounds of the storage requirement. For the CNF approach the size of the path access requirement is bounded linearly by the depth of the file. It will never be worse than storing the user and the group for each parent directory. For the DNF approach the size of the path
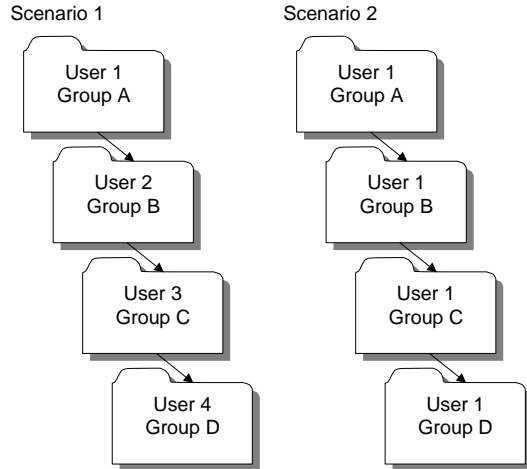
Type	Scenario 1	Scenario 2
CNF	$(1 \vee A) \wedge (2 \vee B)$ $\wedge (3 \vee C) \wedge (4 \vee D)$	$(1 \vee A) \wedge (1 \vee B)$ $\wedge (1 \vee C) \wedge (1 \vee D)$
DNF	$(1 \wedge B \wedge C \wedge D)$ $\vee (2 \wedge A \wedge C \wedge D)$ $\vee (3 \wedge A \wedge B \wedge D)$ $\vee (4 \wedge A \wedge B \wedge C)$ $\vee (1 \wedge 2 \wedge 3 \wedge 4)$	$(1) \vee (A \wedge B \wedge C \wedge D)$

Figure 4. Worst cases for DNF (Scenario 1) and CNF (Scenario 2) methods.

access requirement is bounded quadratically by the depth of the file. The size of the clauses is bounded linearly, and the number of clauses is bounded linearly, therefore this makes the total size of the path access requirement bounded quadratically.

We mentioned earlier that the time to check access rights for a file, and the time to calculate the access control for a file are bounded linearly by the size of the path access requirements. We also described how the path access requirements are bounded linearly for CNF and quadratically for DNF by the depth of the file. Applying transitivity, we can say the time to check access rights for a file, and the time to calculate the access control for a file are bounded linearly by the depth of the file for CNF and quadratically for DNF. Therefore CNF would appear to be the obvious choice, however this is not taking into account the structure and workload of the specific system we would apply it to.

4.3. Handling Updates

Directory traversal must be performed only in the case of an update to a permission or owner which changes who has execute permission on a directory. Previous work has shown that permission changes do not occur very often [12] (less than 1% of metadata operations). Intuitively, when such a change does occur, it is not as likely to occur at a very high level in the tree (e.g. /home/students) where it would affect many files. We can propagate these updates lazily, similar to the Lazy-Hybrid method, to prevent a sudden flurry of activity on the metadata servers by marking those files affected and calculating the new access controls when the file is next requested. The number of network trips necessary to perform the complete update is amortized to only one network trip per file affected. This is because when a file is updated, all of it's ancestors that need to be updated are updated at this time as well. An update only travels as far up the tree as needed. This means that the next time a file is accessed in a directory where a file has been updated already the update for that file will not have to propogate up the tree. The parent directory and all of it's ancestors will have already been updated the first time a file was accessed in that directory after the update. Another interesting thing to note about updates is that when updating the the access control for a directory, if the updated path access requirement is the same as it was before the update then all of the files underneath it do not have to be updated.

4.4. Links

The same approach for updating can be used in order to maintain the correct access control for links. The problem of locating a file through a link in a hash-based system is outside the scope of this paper and has been dealt with in several of the hash-based systems described. However, in order to maintain correct hierarchical permissions in a hash-based system our method can still be applied in an interesting way. Hard links are particularly interesting because when a hard link is added to a file the access control of the file changes.

A system that allows hard links really means that a file may have multiple paths to it. With each of these paths comes a set of access requirements, which can each be expressed as we have previously proposed. We can "or" the path access requirements of all of these paths together in order to get the correct access control for a file with multiple paths (links). So when we create a hard link to a file we take the path access requirements of the path we are adding to the file and merge it with the file access requirements as we would if it were the only path. Then we "or" this expression with the previous path access requirement for the file in order to get the new path access requirement. We can then simplify this expression with any rules we have in place for the system, as well as the common logical rules such as De Morgan's rule. This update is then propogated down the hierarchy if necessary.

For symbolic links, no special consideration is necessary since the link points to the entire path of a file. There are two distinct and separate paths in this case: the path to the link, and the path to the file. These paths are independent of one another, so a change to one has no effect on the other.

5. Data Collection

In order to evaluate the usefulness of our solution for real systems we recorded a snapshot of the file system layout of a large shared system. The file system we collected this data from was a Sun NFS system that contained a large shared research directory and the home directories of faculty, graduate students, staff and others in the CS department at UC Santa Cruz. Additionally, we collected data from common systems folders on a large shared Solaris system. The data was collected using a modified version the GNU find utility [5] that recorded the following information for each file: full path name, size, inode number, atime, ctime, mtime, user, group, mode and number of links. Every element in the full path name was individually encrypted using a keyed MD5 hash. This was done in order to protect user privacy on the system [7] while still allowing the file hierarchy to be reconstructed.

6. Results

Using the raw data collected we were able to reconstruct the hierarchical access control information for the file systems. From this data we can see the benefit of storing the hierarchical access control information for a file as a single object and avoiding directory tree traversals. We can also see the benefits of merging the ownership information with the permission bits at each level in the hierarchy.

The system we examined contained 3,896,681 files in 287,796 directories. There were 1,296 unique users and 141 unique groups. Figure 5 shows a histogram of the depth of each file in the system we examined. The depth of the file directly corresponds to the number of files that must be checked before a user can obtain access to the file. By storing this information compactly on one device we save on average 7-9 network trips to contact devices needed to verify access rights for a user. We also reduce the load on each device since requests only have to contact one device in order to access a file.

A *choke point* is a point in the path to a file where further access is permitted for only a single user, choking off access to all other users. Figure 6 shows a histogram of the choke point depth for the files in the system that had a choke point. This is important because as we showed in the design section, having only one user that can access a file
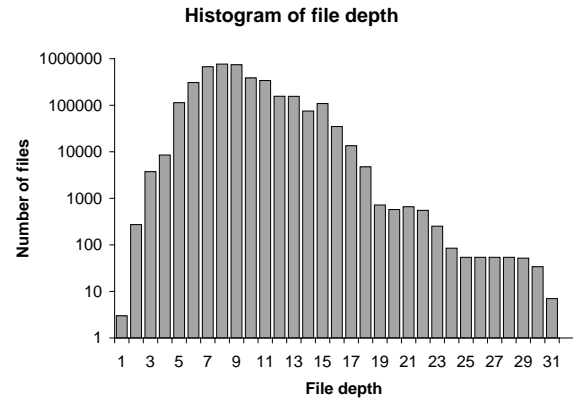

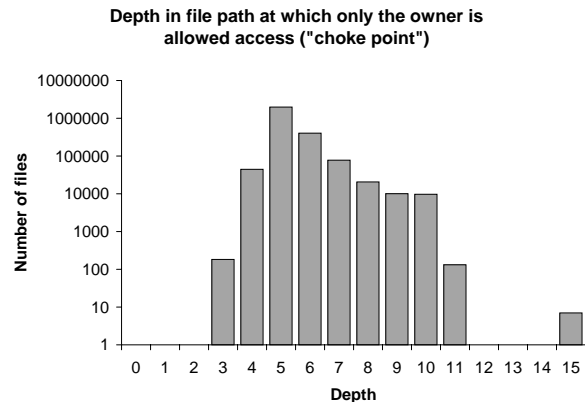**Figure 5. Histogram of the depths of all files.**

Figure 6. Histogram of the depth where a choke point occurred.

greatly reduces the amount of data that needs to be stored. We find that 65% of the files (2,550,969 files) have a choke point.

Figures 7 and 8 show the results of converting the access controls to our CNF representation. The light grey bars show the number of users/groups that must be stored in the clauses before simplification. The dark grey bars show the number of users/groups that must be stored after simplification using only the two rules discussed. We see a dramatic reduction in the amount of data that must be stored. We can represent 31% of the files with no additional information (empty clause), 68% of the files with just one clause, and less than 1% of files with two clauses.

We also ran a simulation using the Lazy-Hybrid metadata management to see how updates affect system throughput and response time. The results are not presented here in the interest of space, but we found that updates did not pose a significant performance problem.
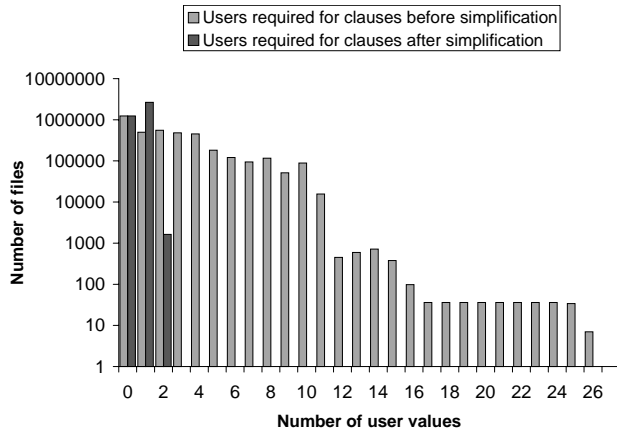
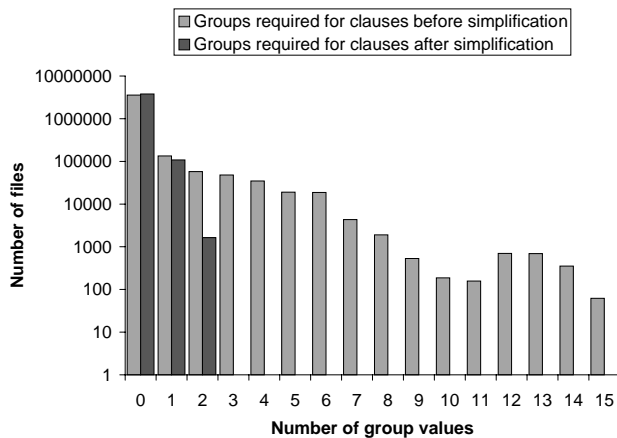**Figure 7. The number of user predicates that must be stored in a CNF representation.**

Figure 8. The number of group predicates that must be stored in a CNF representation.

7. Conclusion

We have presented two methods for representing hierarchical access control for a file in a format that can be stored locally for the file. This allows access control to be determined for a file without having to traverse the directory tree, an operation that can be expensive in a distributed system. We maintain access control by first merging the permissions and ownership information and then merging this information together for each file in the hierarchy. We do this by using an approach based on CNF (Conjunctive Normal Form) or another based on DNF (Disjunctive Normal Form). Data collected from an existing large shared file system demonstrates that this is an effective method for compactly storing the access control information for each file.

Acknowledgments

References

[1] P. J. Braam. The Lustre storage architecture. Aug. 2002.

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

[3] P. F. Corbett and D. G. Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[4] R. A. Floyd and C. S. Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):238–247, 1989.

[5] F. S. Foundation. GNU find, 2003.

[6] G. A. Gibson and R. V. Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[7] G. Kuenning and E. L. Miller. Anonymization techniques for URLs and filenames. Technical report, University of California, Santa Cruz, Sept. 2003.

[8] D. D. E. Long, S. A. Brandt, E. L. Miller, P. Mantey, A. Brandwajn, and K. Obraczka. Scalable file systems for high-performance computing. Apr. 2001.

[9] E. L. Miller and R. H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4):419–446, July 1997.

[10] Y. Padioleau and O. Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, June 2003.

[11] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.

[12] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, June 2000.

[13] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, Nov. 2004.

[14] J. Yan, Y.-L. Zhu, H. Xiong, R. Kanagavelu, F. Zhou, and L. Weon. A design of metadata server cluster in large distributed object-based storage. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 199–205, 2004.