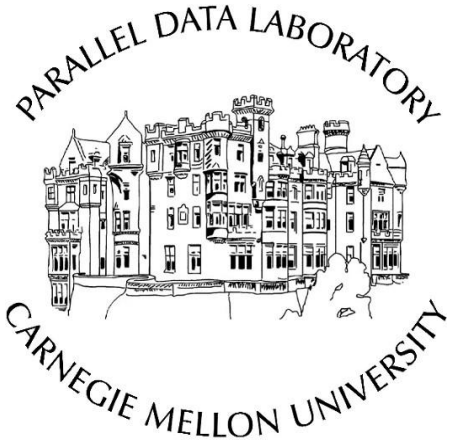

Improving small file performance in object-based storage

James Hendricks, Raja R. Sambasivan, Shafeeq Sinnamohideen, Gregory R. Ganger

CMU-PDL-06-104
May 2006

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

This paper proposes architectural refinements, server-driven metadata prefetching and namespace flattening, for improving the efficiency of small file workloads in object-based storage systems. Server-driven metadata prefetching consists of having the metadata server provide information and capabilities for multiple objects, rather than just one, in response to each lookup. Doing so allows clients to access the contents of many small files for each metadata server interaction, reducing access latency and metadata server load. Namespace flattening encodes the directory hierarchy into object IDs such that namespace locality translates to object ID similarity. Doing so exposes namespace relationships among objects (e.g., as hints to storage devices), improves locality in metadata indices, and enables use of ranges for exploiting them. Trace-driven simulations and experiments with a prototype implementation show significant performance benefits for small file workloads.

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Symantec) for their interest, insights, feedback, and support. We also thank Intel, IBM, Network Appliances, Seagate, and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by the Army Research Office, under agreement number DAAD19–02–1–0389. James Hendricks is supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense.

Keywords: Object-based storage, object ID assignment algorithms, multi-object capabilities, namespace flattening, OSD, range operations, server-driven metadata prefetching

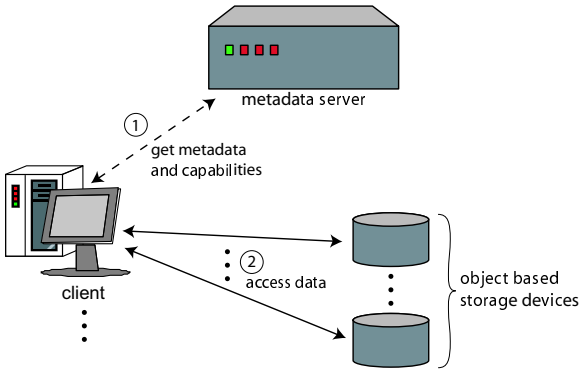

Figure 1: **Direct client access.** First, a client interacts with the metadata server to obtain mapping information (e.g., which object IDs to access and on which storage devices) and capabilities (i.e., evidence of access rights). Second, the client interacts with the appropriate storage device(s) to read and write data, providing a capability with each request.

1 Introduction

Scalable storage solutions increasingly rely on direct client access to achieve high bandwidth. In some cases, direct access is achieved by specialized SAN protocols [2, 28, 16] and, in others, by object-based storage protocols [12, 18, 23]. As illustrated in Figure 1, direct client access offers scalable bandwidth by removing the centralized server bottleneck, shifting metadata management out of the critical path. As a result, this storage architecture is becoming standard in high-end scientific computing.

But, it is destined to be a niche architecture unless it can effectively support a broader range of workloads, despite the potential additional value of object-based storage for document management [8, 26] and automation [11, 26]. Although excellent for high-bandwidth access to large files, direct access systems struggle with workloads involving access to many small files. In particular, direct client access to each file's data requires first accessing the metadata server (for mapping information and capabilities) and then accessing the storage device. With large files, the one-time-per-file metadata access can be amortized over many data accesses. With small files, however, it can double the latency for data access and become a system bottleneck.

This paper proposes architectural refinements to increase small file efficiency in such systems. At a high level, the approach combines restoring lost file inter-relationships with metadata prefetching. When a client requests metadata for one file, the metadata server provides mapping information and capabilities for it and other related files. By caching them, clients can potentially eliminate most metadata server interactions for small file data access, reducing the load on the metadata server and access latencies for clients.

As with any prefetching, a key challenge is identifying strong relationships so that the right objects are prefetched. Today's file systems use the namespace as a hint regarding inter-file relationships, organizing and prefetching metadata and data accordingly. But, in object-based storage, the level of indirection between file naming and object naming obscures this hint. We restore it via *namespace flattening*, or encoding a file's hierarchical directory position into the object ID assigned to it. Such preservation of namespace locality in object IDs naturally retains this traditional hint for storage devices, provides spatial locality in the metadata structures, and enables compact representations for groups (i.e., ranges) of related objects.

This paper describes the architecture and protocol changes required to support server-driven metadata prefetching. As well, efficiency requires capabilities that authorize access to multiple objects and, of course, appropriate client cache management. No interface changes are required for namespace flattening.

Measurements of a prototype implementation show significant benefits for workloads dominated by small file access. Client applications, such as CVS and system compilation, can achieve significantly higher

throughput. Metadata server loads decrease by 27–94%, which would allow the system to scale 1.4–17× larger before facing metadata server scalability problems. In addition to benchmark experiments with the prototype, analysis of real NFS traces confirm the value of namespace-based prefetching and the potential metadata server load reductions. In these traces, 5–34% of all metadata server interactions can be eliminated via the prefetching mechanisms.

2 Small files and object stores

This section reviews object-based storage, its struggles with small files, how we propose to address the struggles, and related work.

2.1 Object-based storage

A storage *object* is a byte-addressed sequence of bytes, plus a set of attributes, accessed via a file-like interface (e.g., CREATE, DELETE, READ, WRITE, and so on). An object store is much like a filesystem, but without the ASCII names. Objects are named by object IDs drawn from a flat numerical namespace (e.g., 64-bit numbers).

Object-based storage was originally conceived [12] as an architecture (illustrated in Figure 1) for achieving cost-effective scalable bandwidth to storage. The metadata server (called a "file manager" by Gibson et al.) would store file system metadata and handle metadata actions, such as creation, deletion, and lookup. To access data, a client would fetch mapping information and capabilities from the metadata server and, then, read/write data directly from/to the object storage devices. By doing so, clients could potentially exploit the full switching bandwidth of the network interconnect for their data accesses. This is in contrast to the conventional server model, which is limited by the bandwidth available from a file server interposed between clients and many disks.

Object storage is gaining popularity and traction. A working group of the Storage Networking Industry Association produced a draft specification, and the ANSI T10 body has reviewed and ratified it as an interface standard [23]. Research on object-based storage continues [9, 20, 29] and early products [18, 24, 26] have appeared. In addition to scalable bandwidth, some are beginning to exploit object-based storage as a mechanism to bundle data and application-defined attributes for long-term maintenance (e.g., for regulatory compliance) [8, 26]. But, for object-based storage to be viable outside of niche domains, it must be able to support small file workloads effectively.

2.2 Problems for small files

Although not required by the architecture, almost all object-based storage systems map each file to an object (or multiple, with data striped among them). These systems struggle with workloads that access large numbers of small files, such as software development and user workspaces, for two reasons: per-file metadata server interactions and loss of namespace locality at the storage devices.

Per-file metadata server interactions: To access a file's data, a client must first have the corresponding mapping information and capabilities. To get them, the client must interact with the metadata server. Only then can the client communicate directly with the storage devices to access the data.

This metadata server interaction happens once for each access. For a client accessing the contents of a large file, this interaction is usually a minor overhead amortized over many data accesses. For a small file, on the other hand, there can be as few as one data access. Two performance problems can result: increased latency for client access and heavy load on the metadata server. Since accessing each file's data requires first interacting with the metadata server, client latency can be doubled (two RPC roundtrips) and the metadata server can be asked to service as many requests as all the storage devices combined.

Loss of storage locality: In object-based storage, the storage devices allocate on-disk locations for the objects they store. The performance consequences and goals for this are essentially the same as those for local file systems. For large objects, good performance will usually be achieved by ensuring that each object's contents are placed sequentially. For small objects, good performance requires inter-object locality, keeping objects that are likely to be accessed together close together on disk.

Most file systems achieve small file locality by exploiting the relationship hints exposed by the directory structures. Object-based storage systems make this difficult by hiding this information from the storage devices—only the metadata server knows about the directory structures. The storage devices see object IDs instead and, thus, can't effectively employ the locality-enhancing tricks common in file systems.

2.3 Solving the problems

Server-driven metadata prefetching and namespace flattening can address the above problems with minimal changes to the object-based storage architecture.

Server-driven metadata prefetching: Rather than returning metadata for just one object, when queried, the metadata server should return metadata for other related objects as well. By doing so, it allows the client to populate its cache with additional mapping information and capabilities—a form of prefetching, but orchestrated by the metadata server. The client still determines what to keep and replace, but the server determines what to prefetch. When the necessary metadata is in its cache, a client can access the storage device immediately. Thus, if the right additional metadata is returned, the number of metadata server interactions should drop dramatically.

The common model of prefetching has the client specify what to get. Server-driven prefetching is more natural in this context for several fundamental and practical reasons. First, and perhaps foremost, knowing what to prefetch requires knowing what exists. The metadata server has this information already and clients do not, unless they track the existence and inter-relationships of files redundantly (e.g., on their local disk). This is in contrast to large file streaming, which can be done by simply asking for the next sequential range of data. Second, server vendors differentiate on performance, among other things. Giving the server control over metadata prefetching increases the likelihood that it will be utilized and tuned aggressively. Third, the metadata server knows what can be prefetched with minimal cost. For example, it can choose what to prefetch based on what metadata blocks are in its cache and where metadata block boundaries are located.

We promote server-driven metadata prefetching in this paper, because it requires less mechanism and less overhead than traditional client-driven prefetching. But, the key insight is that batched metadata prefetching is needed to address the "per-file metadata server interactions" problem. Client-driven prefetching could likely be engineered to work just as well, with enough effort (e.g., duplication of metadata at clients).

Namespace Flattening: Namespace flattening translates the traditional file system approach for improving small file performance to object-based storage. Rather than assigning object IDs via some namespace-independent allocation policy (e.g., a monotonically increasing number), object IDs are chosen to reflect locality in the file namespace. This is analogous to inode number selection policies in file systems, which almost always utilize directory structure information to enhance locality. In fact, object-based storage has been explained [20] as splitting the file system at the inode layer, with the "upper half" being at the metadata server and the "lower half" being at the object storage device.

Encoding namespace relationships into object IDs provides several benefits. First, storage devices can treat the object ID as a locality hint, with closeness indicating relationships that could be exploited for internal layout and cache management policies. This is analogous to how most file systems map namespace locality to block number locality in their on-disk layouts. Second, index structures for object metadata at storage devices, which are typically organized as tables or B-trees indexed by object ID, will also naturally have better locality. Third, a set of related files can be identified by a compact object ID range rather than

an enumerated list.

The use of data identifiers as hints about spatial locality has a long history in storage systems—in fact, it is the foundation on which disk system performance is tuned. File systems and databases assign related data to numerically close block numbers, and storage devices provide higher performance for accesses to numerically close block numbers. Making the object ID a hint about locality follows this same approach, allowing basic (but non-mandatory) cooperation with no changes to the object storage interface or standardization of hint attributes. As with block number locality in traditional storage, the only effect of any client or storage device not conforming to the implicit convention is lower performance. Also, though we promote particular namespace flattening schemes in this paper, it should be noted that the convention does not specify how object IDs are assigned; it simply suggests that numerically close object ID numbers might indicate locality.

2.4 Related work

This section discusses related work. Note that prefetching has a long history in storage systems, and we will not cover all such work.

Namespace-based locality has long been recognized as a good indicator of inter-file relationships and exploited in file system disk layouts. FFS [19] introduced the cylinder group, which many file systems now call "allocation groups," as a mechanism for placing related file system structures in a common disk region with a few simple rules: the inode for a new file is allocated in the same cylinder group as the directory that names it, and the first few data blocks of a file are allocated in the same cylinder group as the inode that describes it. Until space within in a cylinder group runs short, these rules effectively place all of the metadata and data for all small files within any given directory in a small region of the disk. C-FFS [10] and ReiserFS [21] go further by co-locating related data and metadata into sequential runs of blocks on disk according to namespace locality, rather than just attempting to get them nearby.

Namespace-based locality is a reasonable assumption when no other information about future access patterns is available. Many have explored approaches to using application hints [4, 25] and observed access patterns [13, 17] for controlling prefetching, caching, and disk layout. These approaches to identifying inter-file relationships can be more accurate than namespace-based locality, and they could be used with server-driven metadata prefetching (instead of namespace flattening). But, namespace-based locality remains the predominant approach in real systems in part because it avoids the additional mechanisms and (for hints) API+application changes. We believe that enabling exploitation of namespace-based locality is the right place to start, given the minimal changes required.

This paper proposes a new approach to addressing client latency and metadata server load for object-based storage systems handling small-file workloads. At least three approaches have been taken to addressing such issues in other systems. First, metadata can be partitioned among multiple servers [3, 30]; doing so scales throughput, but does not address the extra roundtrip or locality issues and it requires multi-server consistency for some operations (e.g., RENAME and snapshot). Second, requests can be batched to reduce their quantity (e.g., NFSv3's READDIRPLUS or NFSv4's compound RPCs); server-driven metadata prefetching can be viewed as a form of batching, though orchestrated by the server rather than by clients. Third, the metadata server could store the data for small files, rather than using objects for them; this could eliminate the extra round-trip associated with accessing small files, but it would exacerbate rather than reduce metadata server load issues. These three approaches are complementary to the approach proposed in this paper, rather than competitors, and large-scale systems will likely require a combination of several of them.

3 Improving small file efficiency

This section describes the mechanics of server-driven metadata prefetching and two example namespace flattening algorithms.

3.1 Server-driven metadata prefetching

To access data in an object-based storage system, a client must first fetch metadata and capabilities from the metadata server; only then can it interact directly with the object storage devices. This section reviews object metadata and describes the mechanics of server-driven metadata prefetching, including the changes required at each major component of the object-based storage architecture. It also describes multi-object capabilities as a means of avoiding increased cryptographic or network costs for prefetched capabilities.

3.1.1 Object metadata and capabilities

Metadata for each object includes mapping information and descriptive information. Capabilities are credentials, created by the metadata server, that can be shown to a storage device to demonstrate a client's right to access particular objects.

Mapping information: Mapping information describes the location(s) of data corresponding to a particular file and, if more than one location is involved, how data is spread among those locations. A location is composed of the identity of a storage device and an object ID on that device. A file's data can be spread over multiple storage devices in many ways, much as in disk arrays, such as striping with parity (RAID 5) or replication.

Descriptive information: Different systems store different descriptive information in the metadata managed by the metadata server. Examples includes object length, access control lists (ACLs), and access/modification times. ACLs are almost always managed by the metadata server, since this information allows it to determine which requests to service and which capabilities to give out. Conversely, authority over length and time values may lie with the metadata server or with the storage devices. The former is simpler but requires extra interactions between clients and the metadata server in order to update these values. The latter allows the values to be updated as reads and writes occur to the storage devices, eliminating metadata server interactions regarding length and times. But, when data for a file is spread across several storage devices, obtaining authoritative values for length and times is simpler if the metadata server manages them.

Capabilities: A capability provides cryptographic proof to a storage device that a client has permission to perform a particular operation. The metadata server constructs capabilities for clients after deciding that the client should have access. The capability generally consists of a MAC of the mapping information and access rights conveyed, as well as freshness fields to avoid replay attacks. The key used for generating the MAC is a shared secret between the metadata server and the storage device; this allows the storage device to verify that the metadata server created the capability as no other entity is privy to the shared key.

3.1.2 Changes in each component

The object-based storage architecture has three primary components: the metadata server, clients, and storage devices. This section describes how each changes in realizing server-driven metadata prefetching.

Metadata server: The metadata server services metadata queries and updates from clients. In traditional object-based storage systems, each client request interacts with the metadata server with respect to one object. For server-driven metadata prefetching, the metadata server is extended to respond to each lookup request with the metadata and capabilities for the client-specified object as well as other objects it believes that the client is likely to access. The collective response is thus an array, rather than a singleton.

It is desirable to minimize prefetching overheads as the additional responses will not always be valuable. The increased network bandwidth can be reduced by compression techniques, if needed, but should not be substantial in practice—metadata is small even relative to small files. Disk and cryptography overheads, on the other hand, are of greater concern. In terms of disk overhead, the issue is extracting the prefetched metadata from their persistent data structures (e.g., per-object inodes in a table or B-tree). A synergy with namespace flattening helps here, as the spatial locality it creates solves this problem—looking up a sequence of values in a table or B-tree is very efficient. In terms of cryptography overhead, each capability traditionally provides access to a single object, which means that the prefetching requires generation of many capabilities. Section 3.1.3 describes a way of mitigating this cost by having each capability authorize access to multiple objects.

Clients: The primary change to clients is that they now receive additional responses from the metadata server. So, they should cache these responses in addition to the one requested. The client cache management should be straightforward. For example, the client could maintain two distinct caches (one for requested entries and one for server-pushed entries). The client should pay attention to the hit rates of each, however, to avoid using too much space when the server-pushed entries are not useful. The pathological case would be a working set that would just barely fit in the full-sized client cache but no longer fits because of uselessly prefetched values. A well-constructed client cache should notice that the server-pushed entries are not useful, given the workload, and retain fewer of them.

Storage devices: Metadata prefetching and namespace flattening change nothing in the interfaces or internal functionality of storage devices. The only other change for storage devices will be better performance. In particular, if namespace locality corresponds to access locality, then numerically similar object IDs will exhibit temporal locality. This locality will, in turn, translate into disk locality assuming traditional FFS-like disk management structures for objects and their device-internal metadata (e.g., mappings of object offsets to disk locations).

3.1.3 Multi-object capabilities

Traditional object storage uses one capability to authorize access to each object. Thus, the proposed metadata prefetching for N objects would require generation of N capabilities. Since capability generation is a non-trivial computational expense, this would be a significant overhead when some of the prefetched information ends up not being needed.

We propose extending the model so that a capability can authorize access to multiple objects. Multi-object capabilities will reduce prefetching overhead as well as reducing overall metadata server CPU load by reducing the total number of capabilities generated. It is crucial, however, that the capability's size remain small—recall that clients must send a capability as part of every request to an object storage device. Ideally, the capability would be compact and constant-sized, regardless of the number of objects covered.

This ideal can be achieved for capabilities that cover a range of object IDs in which the mapping information is formulaic and the access rights granted are consistent. A range of objects can be specified with just start and end object IDs. The mapping information for a collection of related objects is often similar and could be specified as a list of storage devices and a simple scheme for determining location (e.g., index into list with hash of object ID). Altogether, such a capability would be approximately twice the size of a single object capability (less than 100 bytes).[1]

Ranges are a natural choice when namespace flattening is employed, since sequences of object IDs share locality in the directory hierarchy. A user that has permission to access one file in a directory almost always has like permission to other files in that and nearby directories. Note that the permissions in question

[1] If the metadata server manages object length and times, rather than the storage devices, they will be returned as part of the metadata. But, they need not be part of the capability.

are for the client machine, so many file permission nuances (e.g., execute permission) do not affect the ability to generate a capability over multiple objects.

That said, careful definition of access rights is required when specifying the objects covered with a range. Specifically, the access rights must rely on the storage device to assist in deciding whether access is allowed. For reads, the access right can be "allow read access to object if it exists" to allow sparse ranges. For over-writes, the access right can be "allow write access to existing bytes in existing objects" to allow writes without unbounded space growth. When capacity usage must be controlled (e.g., by quotas), individual interactions with the metadata server would be required when creating new data.

The one additional change needed for multi-object capabilities is in the freshness information. In the OSD specification [23], the storage device stores a version number for each object. The metadata server updates this version number whenever an operation might restrict a previously-issued capability for an object (e.g., the object is deleted or access rights change). When a client requests a capability, the metadata server includes this version number in the capability. For the storage device to verify that a capability is valid, the version number the client provides in the capability must equal the version number for a given object stored by the storage device. To avoid requiring a separate version number for each object in a group capability, we change this as follows. We require that the metadata server use a monotonically increasing version number both for updating the version number at the storage device and for issuing capabilities. We require the storage device to verify that a version number is greater than rather than equal to the version number stored. Given these two changes, a group capability will now contain a version number that is valid for all objects updated before the capability was issued. In addition to being constant-sized, this allows a capability to remain valid for some objects in a group even if it is no longer current for other objects.

3.2 Namespace flattening

Namespace flattening is an object ID assignment strategy that encodes the directory structure into the object ID space. For context, two common assignment policies are pseudo-random and create-order. The pseudo-random policy assigns object IDs with a random number generator or by computing a hash of the filename or contents; neither preserves any locality information. The create-order policy keeps a counter and simply assigns the next value to each created object. This will preserve locality information when creation order matches access order, but tends to suffer from fragmentation over time. This section explains the fragmentation problem in more detail and describes some namespace flattening algorithms.

3.2.1 Fragmentation in the object ID space

Object ID assignment algorithms are subject to an analogue of the inode fragmentation problem in traditional block-based filesystems, as analyzed by Smith et al. [27]. Since object IDs are assigned when objects are created, subsequent operations that modify the namespace (e.g., CREATEs and REMOVEs) will disrupt the relationship between namespace locality and object ID closeness for simple policies like create-order. Over time, this disruption tends to slowly randomize the relationship, reducing locality to the extent that the namespace is a good predictor. (Recall that all modern file systems rely on the namespace in this way.)

A simple algorithm such as create-order suffers especially from fragmentation related problems because it creates a perfectly dense object ID space, leaving no room for growth or churn (i.e., creations and deletions of files) in directories. For example, with create-order, two files created in the same directory on different days are likely to be assigned very different object IDs. The graphs in Figure 2 illustrate create-order's susceptibility to fragmentation. The graphs show the object ID assigned by create-order versus the order of traversal when using the `find` command to find a non-existent file in a linux source tree checked out from a CVS repository. The leftmost graph shows the initial state of the linux source tree as soon as it is checked out from the repository. The `cvs checkout` command creates files in depth-first order, so the assigned object

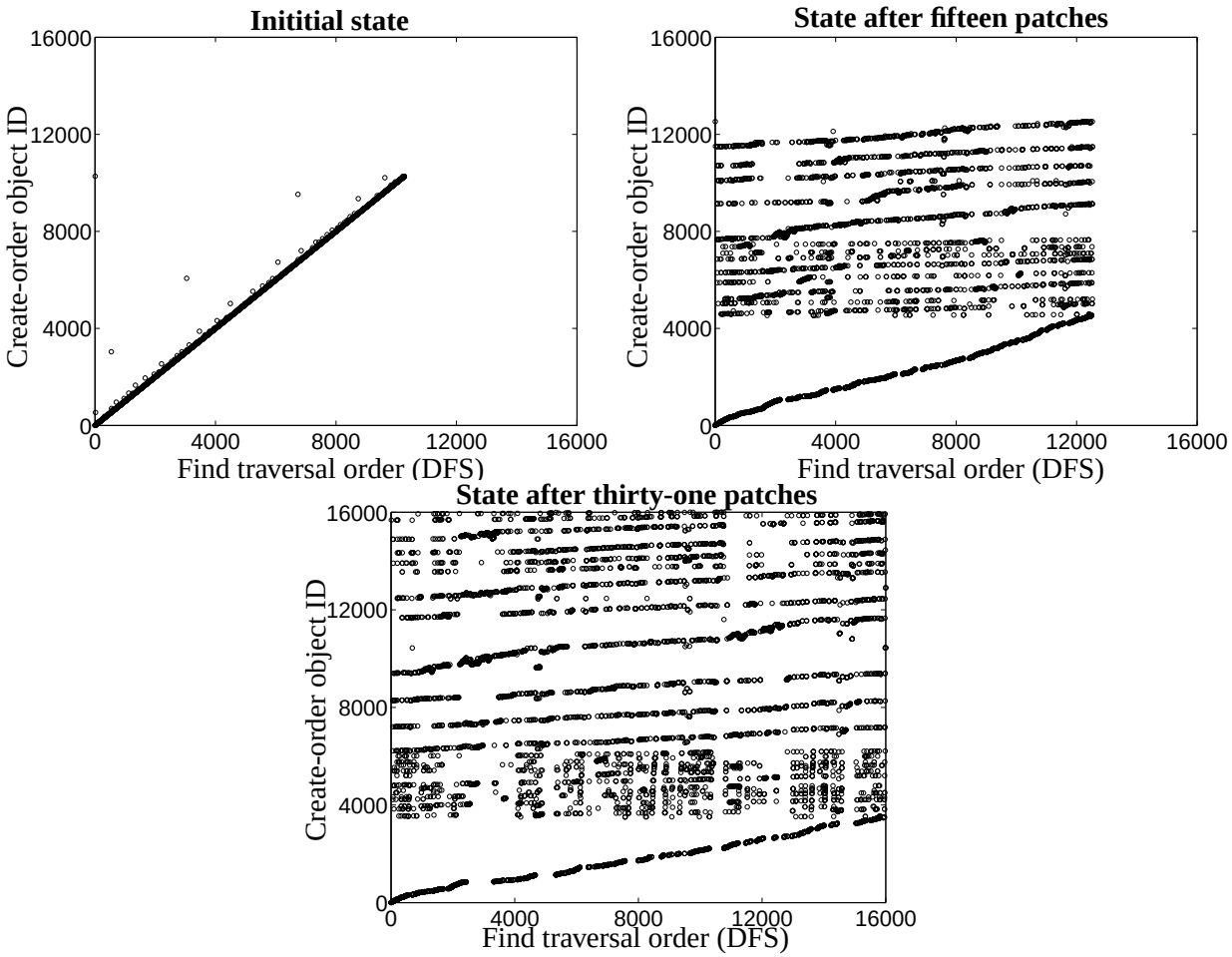

Figure 2: **Fragmentation over time in the object ID space when using the create-order assignment algorithm.**

IDs exactly match the order of traversal used by `find`. Thus, for depth-first searches, exactly the right objects will be prefetched. The middle graph shows the state of the checked out source tree after fifteen patches (from 2.4.0 to 2.4.15) have been retrieved from the CVS repository and applied. The resulting fragmentation is clearly visible—items accessed at similar times (i.e., that are close together on the x-axis) no longer have sequential object IDs. The state after thirty-one patch applications shows even more fragmentation.

An additional source of fragmentation can be RENAME operations that move files across directories, which creates a mismatch between the original allocation choices and the new namespace location. Fortunately, such operations are extremely rare in practice and should not be significant – analysis of the traces used in Section 5.2 indicate they comprise fewer than 0.001% of all operations.

Systems can cope with fragmentation proactively or reactively. Most file systems proactively segment the ID space (e.g., via allocation groups) to create sparseness and separate unrelated growth and churn. Related objects can be matched to the same segment of the ID space, and allocations/de-allocations within that segment will not affect others. Some file systems also reactively "defragment" by periodically sweeping through and changing the assignments to match the ideal. Such defragmentation tends to be difficult in large-scale, distributed storage systems (as contrasted with desktop file systems), but it does complement proactive techniques.

Method	/home/	/home/person/	/home/person/submit.ps	/home/person/submit.pdf
Child-closest	0x0400 0000	0x04a0 0000	0x04a0 0007	0x04a0 0008
Cousin-closest	0x0000 0040	0x0000 04a0	0x0000 04a7	0x0000 04a8

Table 1: **Namespace flattening example.** This table illustrates the two namespace flattening policies by showing object IDs that might be used for some files in a directory hierarchy. All numbers are in hexadecimal.

3.2.2 Namespace flattening algorithms

This section describes two namespace flattening algorithms for assigning object IDs and proactively avoiding fragmentation. Both segment the object ID into slots for each directory depth and avoid fragmentation by keeping each directory's contents within its slot. The first, child-closest, assigns the depths statically, explicitly placing subtrees (children) next to one another in the OID space. The second, cousin-closest, uses shifting to place directories on the same hierarchy level (cousins) close together in the OID space. Table 1 illustrates both, showing how some file names might translate to OIDs. The table and all examples in this section assume 32-bit OIDs (for presentation) shown in hexadecimal.

Child-Closest: The *child-closest* algorithm assigns a slot to each level of the directory hierarchy, moving from left to right, and numbers to each file or directory within a directory. For example, if `/home/` is assigned 0x4 and its subdirectory `person/` is assigned `0xa`, the id for `/home/person/submit.pdf` would start as `0x04a...`, assuming that each slot is four bits in size. The file number grows from right to left. If `submit.pdf` is assigned `0x8`, then the object ID of `/home/person/submit.pdf` would be `0x04a0 0008`. The object containing the contents of a directory is always the zeroth file in that directory, assigned file number `0x0` within the directory's number space.

The *child-closest* algorithm will perform particularly well for depth-first traversals (e.g., as exhibited by the `find` command) and workloads with similar access patterns. Specifically, with this algorithm, the closeness of the object ID of a directory and a file is a function of their *vertical* distance in the directory hierarchy, followed by their *horizontal* distance; the immediate descendants of a directory are assigned OIDs that are very close to that of the directory itself and other descendants are assigned OIDs that are decreasingly close. An additional feature of this policy is that it can represent an entire subtrees in a single range. For example, the range `0xae410000--0xae41FFFF` might describe the entire subtree rooted at `/home/person/cvs/project/src`.

Cousin-Closest: The *cousin-closest* algorithm also uses slots and assigns a number to each file and subdirectory within a directory. But, its object IDs grow from right to left by shifting the parent directory bits over one slot. For example, if `/home/` is assigned 0x4, `person/` is assigned `0xa`, and `submit.pdf` is assigned `0x8`, then the object ID of `/home/person/submit.pdf` would be `0x0000 04a8`. As with child-closest, the object containing the contents of the directory is the zeroth file in that directory.

The cousin-closest algorithm will perform particularly well for breadth-first traversals and workloads with similar access patterns. With this algorithm, the closeness of the object ID of a directory and a file is a function of their *horizontal* distance in the directory hierarchy, followed by the *vertical* distance. An additional feature of cousin-closest is that it tends to produce denser representations of directory hierarchies.

3.2.3 Excessively deep and wide hierarchies

Both namespace flattening policies use statically-sized slots for each directory and file number. This means that it is possible to have too much width (running out of numbers within a slot) or too much depth (running out of slots). To address overflow, we partition the object ID space into four distinct regions by using the two high-order bits. For concreteness, the descriptions below provide examples based on a 64-bit object ID space in which two bits are used to specify the region, leaving 62 bits within each region.

Primary region: The primary region uses one of the namespace flattening policies for non-overflow files and directories. In this example, lets assume three bits for each directory slot and eight bits for the file number within a directory. This primary region accommodates 18 levels of directories and 255 files within any directory.

Deep region: If a directory in the directory hierarchy is too deep (greater than 18 levels, in the example), an unallocated segment of the deep region is used. The deep region contains a prefix (e.g., 27 bits) and a slotted region (e.g., one file slot and 9 directory slots). Prefixes are allocated in sequential order, and a new one becomes the root of a new slotted region that grows downward. Locality is lost between the re-rooted directory and its parent, but locality with its subdirectories is maintained. Despite its name, the deep region is also used for excess subdirectories in a directory. In the example, only 8 directories can be numbered within a slot. The ninth and beyond are re-rooted to a deep region prefix, as described above.

Wide region (0x2): If a directory has too many files (more than 255, in this example), an unallocated segment in the wide region is used for the overflow from that directory. The wide region is divided into a prefix (e.g., 40 bits) and a large file segment (e.g., 22 bits). Using the sample numbers, four million files can be grouped under each of the trillion prefixes. As with the deep region, ID locality is lost between the original directory's object ID and the files rooted under the wide region. But, for very wide directories, this is unlikely to be a significant penalty.

Final region (0x3): If one runs out of prefixes in either the wide region or the deep region, the final region is used. Object IDs in the final region (2^{62}, in the example) are assigned via create-order.

To guide slot size choices and explore the extent of overflow expected in a reasonably sized filesystem, we studied a departmental lab server that houses the home directories and software development activities of about twenty graduate students, faculty, and staff. On this server, 95% of all directories contain fewer than eight directories and 47 files (99% have fewer than 127). Also, 95% of files and directories are fewer than 16 levels of directories away from the root directory. Thus, for this server, overflow would be rare for the example above.

4 Experimental Apparatus

This section describes the prototype implementation and trace-driven simulator used to evaluate the proposed techniques.

4.1 Prototype object-based storage system: Ursa Minor

We implemented both namespace flattening algorithms and server-driven metadata prefetching in an object-based storage system called Ursa Minor. A detailed description of Ursa Minor is available in [1], and it conforms to the basic architecture illustrated in Figure 1. Its centralized metadata server manages metadata and distributes capabilities for given object IDs. The storage devices, implemented as application-level software, store object data, and clients access this data. There is also an NFS server that acts as an object storage client on behalf of unmodified clients.

Server-driven metadata prefetching involved changes to the metadata server and the client metadata cache. Since the metadata server stores metadata in a B-tree indexed by object ID, it was easy to have it prefetch the remaining items in the B-tree page containing the metadata requested by the client. If the prefetch size (i.e., number of additional objects for which to prefetch metadata, set to 32 in our experiments) is greater than the number of metadata items contained in this page, the appropriate number of items from surrounding pages are also prefetched. Our current implementation does not support multi-object capabilities or metadata compression, so replies to metadata requests are just lists that contain both the requested and prefetched metadata. The client metadata cache was replaced with a segmented LRU cache [14] con-

taining both a primary and a prefetch segment. Demand-fetched entries are inserted into the primary cache and evicted using a LRU policy. Prefetched entries are inserted into the prefetch cache and moved to the front of the primary cache if used before being evicted.

We implemented the namespace flattening algorithms in the NFS server. The NFS server translates NFS requests for filehandle data into requests for object data that can be understood by the underlying object storage system. A one-to-one mapping between objects and filehandles is maintained by the NFS server, and object IDs are assigned at time of creation. Specifically, upon receiving a `CREATE` or `MKDIR` request, the NFS server allocates an object ID, creates the object, and constructs an NFS filehandle that includes the new object ID. Object IDs in the system are 128 bits in size, with the high 32 bits dedicated to specifying a partition number. Thus, the namespace flattening algorithms have 96 bits to work with. However, given the 64-bit object ID in the OSD specification, we limit the system to using 64 bits. We chose 10 bits for the file number and 5 bits for each of the 10 directory slots.[2]

Ursa Minor's metadata accesses can be classified into two categories: mandatory and non-mandatory. *Mandatory* accesses are accesses due to operations that must be propagated to the metadata server immediately so as to guarantee a consistent namespace view for all clients. Specifically, they are accesses due to operations that modify the namespace (e.g., CREATE, RENAME, and REMOVE). Since, in this system, the metadata server manages length information, APPEND and TRUNCATE operations also incur mandatory accesses. All other operations incur *non-mandatory* accesses (e.g., READ, WRITE, etc.). Namespace flattening and server-driven metadata prefetching can eliminate non-mandatory accesses, but cannot prevent mandatory accesses.

4.2 Trace-driven object storage simulator

We extended an object-based storage simulator to allow evaluation with large traces of real workloads and fewer implementation artifacts. The simulator takes as input an NFS trace and outputs the number of metadata cache accesses incurred by each client, which is an indication of the load placed on the metadata server and required client latency.

The simulator proceeds in two phases: reconstruction and simulation. The *reconstruction phase* scans a trace to recreate the state of the file system, as much as possible, at the time before the simulated portion of the trace. It uses the information yielded by traced operations (e.g., `CREATE`, `LOOKUP`, and `READDIR`) to reconstruct the namespace. At the end of this phase, object IDs are assigned to all items in the reconstructed namespace according to whichever policy is used.

The reconstructed namespace will be imperfect. Most importantly, only files and directories that were accessed are visible in the trace, and so unused parts of the original file system will be absent. This will tend to make the file system look smaller and more densely accessed. In addition, the creation order of files that exist before the trace began cannot be known. It can be predicted with the modification time available for most accessed files, given that most files are created and written in their entirity in the traced environment. We believe that, despite such limitations, the simulation results represent reasonable expectations for the traced environments.

The *simulation phase* models client-metadata interactions for the reconstructed file system. It simulates a metadata cache (using segmented LRU) for each client. For each file accessed in the trace, a check is made to see if metadata for that file exists in the appropriate client's metadata cache. If so, the number of cache hits is incremented. If not, the number of metadata accesses is incremented and metadata for the surrounding N object IDs are prefetched into the client's metadata cache. Accesses to files not in the reconstructed namespace are ignored, as where they fit is unclear. Such files account for less than 2% of all files accessed

[2] Our implementation of namespace flattening algorithms uses 4 bits for the region number, even though only 2 bits are actually needed. This is an artifact of our current implementation.

in most of the traces analyzed, however, in one trace (EECS03), they account for 25%. Files created during simulation are added to the reconstructed namespace and assigned an object ID.

Like the object-based storage system, metadata accesses in the simulation phase are categorized as either mandatory or non-mandatory. Unlike in the prototype system, however, clients are granted the authority to manage the length of a file themselves for a certain amount of time without propagating updates to the metadata server. As a result, mandatory accesses only include those accesses that result from operations that modify the namespace (e.g., `CREATE`, `RENAME`, etc.). All other operations, including length updates, are modeled as non-mandatory accesses. This model of file length management more accurately reflects existing object storage systems [12, 24].

5 Evaluation

We evaluated server-driven metadata prefetching and namespace flattening algorithms in two ways. To quantify the benefits of both techniques on an actual system, we ran several benchmarks on the modified version of Ursa Minor (Section 5.1). To determine the effects of fragmentation on the object ID assignment algorithms, we also evaluated both techniques via trace replay of large, real, NFS traces using our object-based storage simulator (Section 5.2). For both evaluations, we report the reduction in accesses to the metadata server as compared to the case in which no prefetching is performed—this is a measure of both the decrease in end-to-end latency seen by clients and work saved at the metadata server.

5.1 Evaluation using Ursa Minor

We ran four benchmarks on Ursa Minor to determine the benefits of prefetching using the child-closest and cousin-closest namespace flattening algorithms compared to when no prefetching is performed. For comparison purposes, we also implemented the create-order assignment algorithm, which assigns a monotonically increasing object ID to objects when created. The benchmarks were run using a total of four machines. Two were run as storage devices—one stored data and the other metadata. The NFS Server and benchmark were co-located on the same machine and communicated via the `loopback` network interface. This setup emulates direct client access, albeit with additional software overhead. Finally, a single machine was dedicated to the metadata server. Each machine contained a 3.0 GHz Pentium 4 processor, a Intel Pro 1000 Network Card, and four 230 gigabyte Western Digital WD2500 SATA disk drives that ran on a 3ware 9000 series RAID controller in JBOD mode. Thirty-two items were prefetched on every access to the metadata server and the segmented LRU client metadata cache was configured to use a 2,000 entry demand cache and a 1,000 entry prefetch cache.

The four benchmarks used for evaluation are listed below. We used these because more popular benchmarks do not exhibit inter-file locality. For example, Postmark uses a random number generator to select which file to access next [15], and IoZone uses only a single file for its benchmarking [22]. For these synthetic workloads, no noteworthy benefits are seen from namespace flattening and metadata prefetching. Each of our custom benchmarks represents a specific, but common, use of a filesystem.

`Tar`: This benchmark consists of untarring the Linux 2.4 source tree. The potential benefit from prefetching is very limited because this benchmark's metadata interactions are almost all creates, which are mandatory accesses that cannot be eliminated.

`Build`: This benchmark consists of building the Linux 2.4 source. This involves reading the source files and creating object files. Due to the reads involved, the maximum potential benefit from prefetching is greater than in `Tar`. Still, the maximum benefit is limited by the large number of mandatory create operations for the object files.

`Patch/rebuild`: This benchmark consists of patching the Linux 2.4.31 source to Linux 2.4.32 and

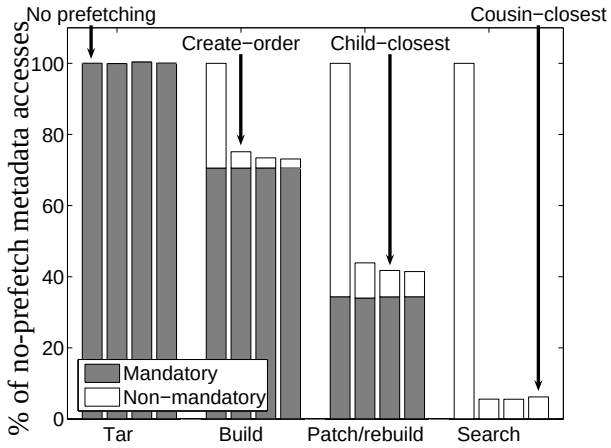

Figure 3: **Benefit obtained by using the various object ID assignment algorithms and server-driven metadata prefetching in Ursa Minor (lower is better).** These graphs show the decrease in metadata accesses when prefetching is performed while using the various object ID assignment algorithms as compared to the baseline (in which no prefetching is performed) for the benchmarks considered.

rebuilding. The maximum potential benefit from prefetching is greater than in `Build` because the re-build phase creates fewer files.

`Search`: This benchmark consists of a depth-first search for a non-existent word in the Linux 2.4 source tree using the command `find . -type f | xargs grep <NONEXISTENT>`. Since its workload is read-only, this benchmark involves no mandatory accesses to the metadata server. The reduction in metadata accesses is limited only by the number of items prefetched and the efficacy of the object ID assignment algorithm used. As such, compared to the other benchmarks, `Search` exhibits the greatest potential for reduction in metadata server accesses.

Figure 3 shows the results of the benchmarks. The results show that, in `Search` where the potential for benefit is greatest, server-driven metadata prefetching and use of any of the three possible object ID assignment algorithms eliminates 94% of all metadata accesses. In `Patch/rebuild`, 89% of non-mandatory accesses and 58% of all metadata accesses are eliminated by the child-closest and cousin-closest algorithms; slightly fewer accesses are eliminated by the create-order algorithm. Because 71% of all accesses are mandatory in `Build`, the potential for benefit is limited. However, prefetching using the namespace flattening algorithms eliminates 27% of all accesses in this benchmark and 75% of all non-mandatory accesses. Create-order eliminates 9% fewer non-mandatory accesses than the namespace flattening algorithms, but this yields only a 2% difference in total metadata access prevented. Since all accesses in `Tar` are mandatory, prefetching yields no benefit in this benchmark.

The create-order, child-closest, and cousin-closest algorithms all perform similarly in these benchmarks. The create-order algorithm performs well because none of the benchmarks exhibit enough concurrency or create enough additional files to fragment the object ID space during their brief runtimes. Even so, some difference is visible—the create-order algorithm performs sightly worse than the child-closest or cousin-closest algorithms in `Build` and `Patch`.

5.2 Evaluation using the trace-driven object storage simulator

In addition to evaluating server-driven metadata prefetching and namespace flattening algorithms in the modified object-storage system, we also performed a trace-based evaluation of these techniques using our object-based storage simulator. For this evaluation, the simulator was configured to use the same prefetch

size and cache configuration as that used for the experiments run on Ursa Minor (32 items were prefetched on every cache miss, and the segmented LRU client metadata cache was configured to use a 2,000 entry demand cache and a 1,000 entry prefetch cache). Since the creation order of files cannot be known for files that were created before the start of the trace period, the create-order algorithm was approximated using the earliest modification time seen for each file in the trace. Finally, for comparison purposes, we also implemented a pseudo-random object ID assignment algorithm. We expected this algorithm to perform worse than any of the other assignment algorithms and, when significant cache pressure exists, worse than when no prefetching is performed.

Section 5.2.1, describes the traces used for this evaluation. Section 5.2.2 discusses the aggregate results obtained. Section 5.2.3 discusses the effects of individual client workloads on the aggregate benefit achieved.

5.2.1 Traces used

Three NFS traces from Harvard University were used in this trace-based evaluation. We describe each trace and its constituent workload below.

EECS03: The EECS03 trace captures NFS traffic observed at a Network Appliance filer between February 8^{th}–9^{th}, 2003. This filer serves home directories for the Electrical Engineering and Computer Science Department. It sees an engineering workload of research, software development, course work, and WWW traffic. Detailed characterization of this environment can be found in [7].

DEAS03: The DEAS03 trace captures NFS traffic observed at another Network Appliance filer between February 8^{th}–9^{th}, 2003. This filer serves the home directories of the Department of Engineering and Applied Sciences. It sees a heterogenous workload of research and development *combined with* e-mail and a small amount of WWW traffic. The workload seen in the DEAS03 environment can be best described as a combination of that seen in the EECS03 environment and e-mail traffic. Detailed characterization of this environment can be found in [6] and [7].

CAMPUS: The CAMPUS trace captures a subset of the NFS traffic observed by the CAMPUS storage system between October 15^{th}–28^{th}, 2001. The CAMPUS storage system provides storage for the e-mail, web, and computing activities of 10,000 students, staff, and faculty and is comprised of fourteen 53 GB storage disk arrays. The subset of activity captured in the CAMPUS trace includes only the traffic between one of the disk arrays (home02) and the general e-mail and login servers. NFS traffic generated by serving web pages, or by students working on CS assignments is not included. However, despite the these exclusions, the CAMPUS trace contains more operations per day (on average) than either the EECS03 or DEAS03 trace. Detailed characterization of this environment can be found in [5] and [6].

Due to differences in the number of operations seen in each trace, the size restrictions of the database used by the simulator to store the reconstructed namespace, and raw time required for processing, we were unable to use the the same time periods for each trace. For the EECS03 trace, we reconstructed the server namespace using the February 8^{th}, 2003 trace and performed simulation over the February 9^{th}, 2003 trace. Reconstruction and simulation for the DEAS03 trace was performed over the same dates as the EECS03 trace. For CAMPUS, we reconstructed using the October 15^{th} to October 21^{st}, 2001 trace and simulated using the October 22^{nd} to 28^{th} trace.

5.2.2 Overall results

The graphs in Figure 4 show the aggregate metadata accesses incurred by all clients in each trace when prefetching using the various algorithms. The left-most graph shows the number of total metadata accesses

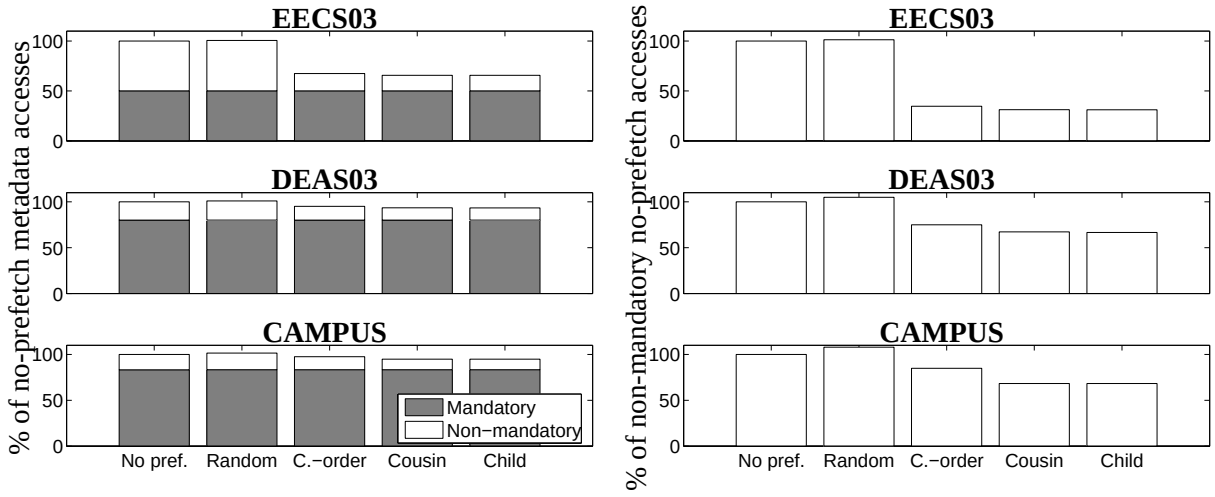

Figure 4: **Percentage of metdata accesses required by each object ID assignment algorithm as compared to the case where no prefetching is performed (lower is better).** In the leftmost graph, metadata accesses are categorized as either mandatory and non-manatory. Since mandatory accesses cannot be elminated, the rightmost graph non-manatory accesses only.

required as a percent of that required when prefetching is disabled. In this graph, the gray bar shows the contribution by mandatory accesses whereas the white bar shows the contribution by non-mandatory accesses. Since mandatory accesses cannot be eliminated via prefetching, the right-most graph shows the results obtained when mandatory accesses are excluded. The aggregate results shown in the graphs yield two major results. First, use of server-driven metadata prefetching and namespace flattening results in a large reduction in metadata accesses in the EECS03 trace, but the benefit these techniques offer is limited in both the DEAS03 trace and the CAMPUS trace. Second, the create-order algorithm does perform worse than the child-closest and cousin-closest algorithms in all of the traces, but the difference is most evident in cases where the benefit derivable from prefetching is limited; in such cases the choice of what items to prefetch becomes paramount and the inter-file relationships exposed by the child-closest and cousin-closest algorithms serve to realize the small potential for benefit.

Of all the traces used for our study, the EECS03 trace is probably the most representative of an academic/research workload, as both DEAS03 and CAMPUS are both dominated by e-mail traffic. As such, it is worthy of note that the EECS03 trace stands to gain the most benefit from both techniques. Almost 50% of all metadata accesses in this trace are non-mandatory and can be eliminated. This potential is realized most by the child-closest and cousin-closest algorithms, which both eliminate 68% of them. Problems due to fragmentation do not affect create-order much in this trace, as it performs only slightly worse than the two namespace flattening algorithms.

The potential benefit from prefetching is more limited in the DEAS03 and CAMPUS trace than in the EECS03 trace. However, the effects of fragmentation on the create-order algorithm are much more visible on these two traces. In DEAS03, 80% of all metadata accesses are mandatory; this predominance of mandatory accesses results from the large number of temporary files (each of which incur mandatory `CREATE` and `REMOVE` operations) seen in this trace. Of the 20% of accesses that are non-mandatory, the child-closest and cousin-closest algorithms eliminate 33%, while create-order eliminates 25%.

Like DEAS03, benefit is limited in the CAMPUS trace due to temporary file accesses. During peak hours, 50% of all files referenced in CAMPUS trace are temporary lock files used to coordinate access to the inboxes. Additionally, many CAMPUS users use e-mail applications that create many temporary files for e-mail compositions [5]. Non-mandatory accesses comprise 17% of all metadata accesses in CAMPUS; 31% of these accesses are eliminated by the child-closest and cousin-closest algorithms, while only 15% is

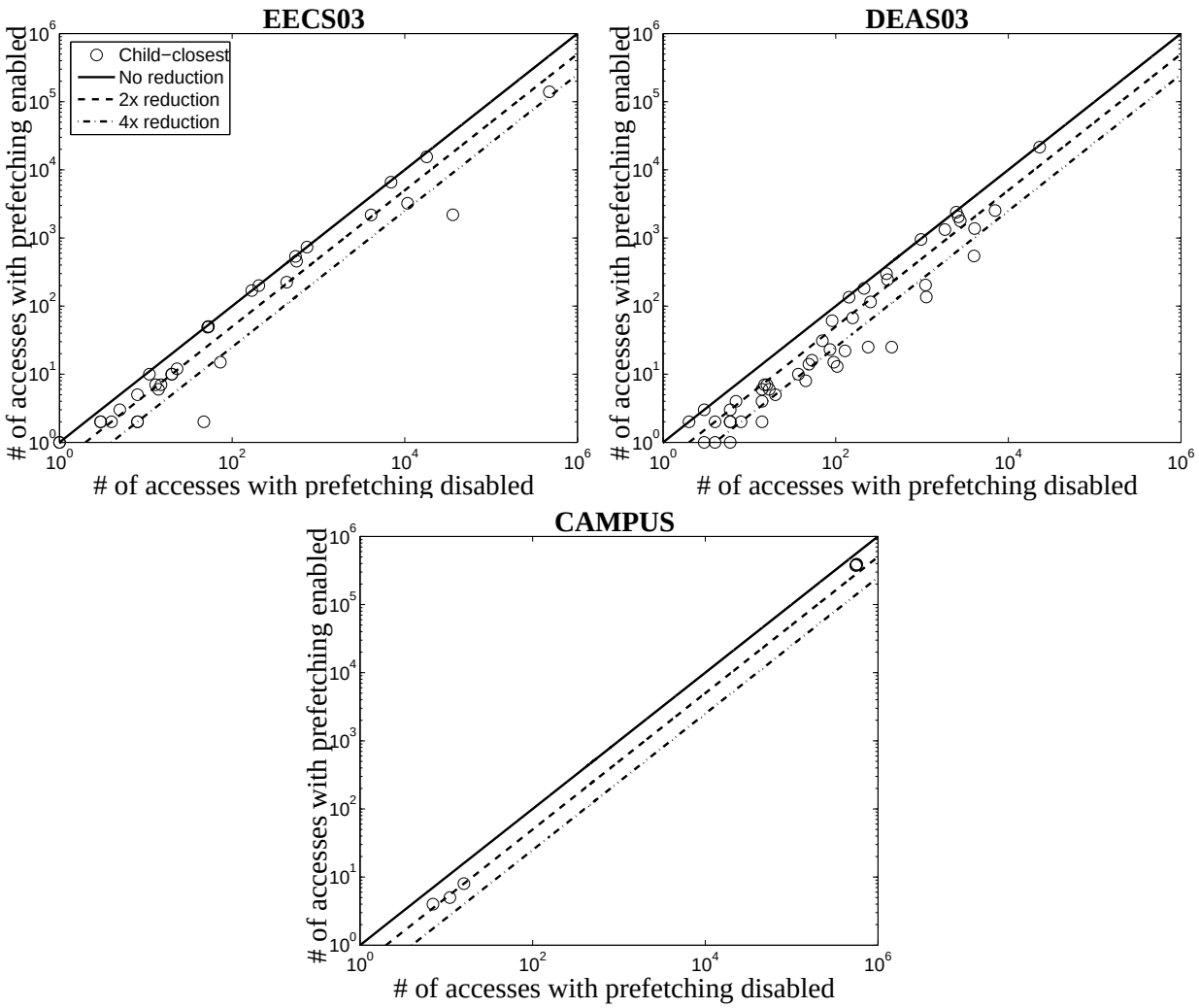

Figure 5: **Factor reduction in number of metadata accesses by each client.** Each log-log scale scatter-plot shows the number of metadata accesses with prefetching disabled on the x-axis and the number of metadata accesses with prefetching enabled on the y-axis.

eliminated by the create-order algorithm.

This trace-based evaluation shows that, though use of server-driven metadata prefetching and namespace flattening algorithms do yield noticeable benefits on academic/research workloads (e.g., they yield a 68% reduction in non-mandatory metadata accesses in the EECS03 trace), the exact choice of namespace flattening algorithm is not critical. Two possible conclusions can be inferred from the identical performance of the child-closest and cousin-closest algorithms . First, prefetching metadata of files in the same directory level might be as useful as prefetching the metadata of descendants. Alternatively, the access patterns seen in the traces might be such that it is only useful to prefetch metadata for items in the same directory as the demand-fetched files.

5.2.3 Individual client performance

The graphs in Figure 5 show the reduction in non-mandatory metadata accesses seen by each individual client in the traces. Since no differences were observable between the child-closest and cousin-closest

algorithms, only the accesses incurred by the first are shown. The graphs show that the individual workloads generated by various clients significantly impact the benefit obtainable from server-driven metadata prefetching and namespace flattening. For example, though the aggregate reduction in metadata accesses is limited in the DEAS03 trace, many clients see between a factor of 2 to a factor of 4 reduction in accesses. The aggregate benefit is limited because of one client that generates 43% of all metadata accesses and sees no benefit from prefetching. Conversely in the EECS03 trace, most clients do not see much benefit from prefetching. However, a single client, which accounts for 86% of all metadata accesses, sees a factor of 4 reduction. Only seven clients are present in the CAMPUS trace; three of these clients account for less than 1% of all accesses combined, whereas the other four account for 25% each. The four clients that account for the majority of accesses do not see much benefit from prefetching.

In summary, even though the aggregate reduction in metadata accesses may be limited for some workloads, hence limiting metadata server scalability, individual client latencies may still see large benefits.

6 Conclusion

Server-driven metadata prefetching and namespace flattening mitigate the small file efficiency problems of object-based storage systems. Rather than having clients interact with the metadata server for each file, the server provides metadata for multiple files each time. This reduces both metadata server load and client access latency. Namespace flattening translates namespace locality into object ID similarity, providing hints for prefetching and other policies, enhancing metadata index locality, and allowing compact range representations. Combined, these techniques should help object-based storage to satisfy a larger range of workload types, rather than being just for high-bandwidth large file workloads.

Acknowledgements

We would like to thank Craig Soules for his many insights, comments and reviews. We also thank Dan Ellard and Margo Seltzer for sharing the NFS traces.

References

[1] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 59–72. USENIX Association, 2005.

[2] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, Dave Andersen, Mike Burrows Timothy Mann, and Chandramohan A. Thekkath. Block-level security for network-attached disks. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 159–174. USENIX Association, 2003.

[3] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, **14**(1):41–79. ACM, February 1996.

[4] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. *Summer USENIX Technical Conference* (Boston, MA), pages 171–182, 6–10 June 1994.

[5] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003), pages 203–216. USENIX Association, 2003.

[6] Daniel Ellard, Jonathan Ledlie, and Margo Seltzer. *The utility of file names*. Technical report TR-05-03. Harvard University, March 2003.

[7] Daniel Ellard and Margo Seltzer. New NFS tracing tools and techniques for system analysis. *Systems Administration Conference* (San Diego, CA), pages 73–85. Usenix Association, 26–31 October 2003.

[8] EMC Corp. EMC Centera: content addressed storage system. http://www.emc.com/products/systems/-centera.jsp?openfolder=platform.

[9] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: the future building block for storage systems. *IEEE Symposium on Mass Storage Systems* (Sardinia, Italy, 19–24 June 2005), pages 101–105. IEEE, 2005.

[10] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. *USENIX Annual Technical Conference* (Anaheim, CA), pages 1–17, January 1997.

[11] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU–CS–03–178. Carnegie Mellon University, August 2003.

[12] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[13] James Griffoen and Randy Appleton. Reducing file system latency using a predictive approach. *Summer USENIX Technical Conference* (Boston, MA), pages 197–207, 6–10 June 1994.

[14] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk performance. *IEEE Computer*, **27**(3):38–46, March 1994.

[15] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[16] Kent Koeninger. CXFS: A clustered SAN filesystem from SGI. http://www.sgi.com/Products/PDF/-2691.pdf.

[17] Thomas M. Kroeger and Darrell D. E. Long. The case for efficient file access pattern modeling. *Hot Topics in Operating Systems* (Rio Rico, Arizona, 29–30 March 1999), pages 14–19, 1999.

[18] Lustre. http://www.lustre.org/.

[19] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[20] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based Storage. *Communications Magazine*, **41**(8):84–90. IEEE, August 2003.

[21] namesys. ReiserFS. http://www.namesys.com/.

[22] William Norcott and Don Capps. Iozone filesystem benchmark program, 2002.

[23] Information technology – SCSI Object-Based Storage Device Commands (OSD), Ralph O. Weber, editor. ANSI Technical Committee T10, July 2004. ftp://ftp.t10.org/t10/drafts/osd/osd-r10.pdf.

[24] Panasas, Inc. http://www.panasas.com/.

[25] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5):79–95, 1995.

[26] Permeon solutions, Permabit Inc. http://www.permabit.com/.

[27] Keith A. Smith and Margo I. Seltzer. File system aging-increasing the relevance of file system benchmarks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Seattle, WA, 15–17 June 1997). Published as *ACM SIGMETRICS Performance Evaluation Review*, **25**(1):203–213. ACM Press, 1997.

[28] Tivoli. Tivoli SANergy: Helping you reach your full SAN potential. http://www.tivoli.com/products/-documents/datasheets/sanergy_ds.pdf.

[29] Feng Wang, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long. OBFS: a file system for object-based storage devices. *NASA Goddard/IEEE Conference on Mass Storage System and Technologies* (Adelphi, MD, 13–16 April 2004). IEEE, 2004.

[30] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. *ACM International Conference on Supercomputing* (Pittsburgh, PA, 06–12 November 2004). IEEE Computer Society, 2004.