Proceedings 13th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). September 26 - 29, Atlanta, GA.

Replication policies for layered clustering of NFS servers

Raja R. Sambasivan, Andrew J. Klosterman, Gregory R. Ganger
Carnegie Mellon University

Abstract

Layered clustering offers cluster-like load balancing for unmodified NFS or CIFS servers. Read requests sent to a busy server can be offloaded to other servers holding replicas of the accessed files. This paper explores a key design question for this approach: which files should be replicated? We find that the popular policy of replicating read-only files offers little benefit. A policy that replicates read-only portions of read-mostly files, however, implicitly coordinates with client cache invalidations and thereby allows almost all read operations to be offloaded. In a read-heavy trace, 75% of all operations and 52% of all data transfers can be offloaded.

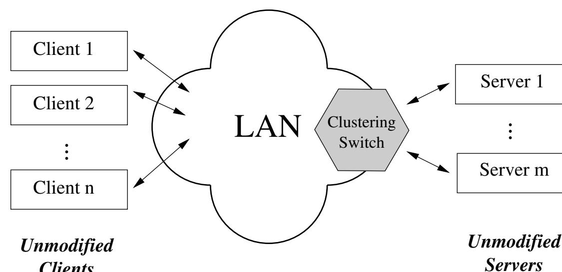

Figure 1. Layered clustering architecture. In the layered clustering architecture, clients, servers, and the client-server protocol are unmodified. Clients send requests to the machine that regularly exports the file the client wishes to access. The *clustering-switch* adds clustering functionality by transparently redirecting client requests to other servers.

1. Introduction

Cluster-based file services promise many benefits [1, 9, 19]. They offer incremental scalability of storage capacity and performance. They can spread data amongst themselves so as to balance the workload. They can keep redundant data for fault tolerance. They can provide high-end features with commodity components. For many years, the research community has known their superiority to the more monolithic file service architectures that persist. Unfortunately, though, their market penetration is minimal. Particularly in mid-sized environments, due to replacement costs and the inertia of existing server installations, it remains common to have a small set of stand-alone servers. Moreover, the architectures of popular distributed file systems (notably, NFSv3 and CIFS) make scalability and load balancing difficult and time-consuming.

An alternate architecture, which we call *layered clustering*, promises a large fraction of the benefits of cluster file services with minimal change to existing systems. Specifically, layered clustering leaves clients, servers, and the client-server protocol unchanged; it interposes *clustering switch* functionality between clients and servers, either in the network stack of each server or in a network component between clients and servers (see Figure 1). This clustering switch functionality can transparently redirect, duplicate, or otherwise modify RPC requests to achieve many of the benefits of a true cluster file service. Given the inertia of installed bases, layered clustering is a compelling option for mid-sized environments, and some companies (e.g., Rainfinity [15]) now offer clustering switches for file servers.

This paper develops and evaluates load balancing policies for clustering switches. We focus on the Cuckoo layered clustering model [12] in which file servers continue to host and "own" entire file systems while also servicing requests for data replicated from other servers. Thus, administrators can manage individual file servers as they always have, with no change in failure models or understanding of "what is where?". But, now read requests from busy servers can be offloaded to others, achieving a degree of cluster-like load balancing.

A request can be offloaded only if the data accessed has been replicated. A *replication policy* determines which objects (directories or files) to replicate in anticipation of future offloading opportunities. This paper describes and evaluates three such policies of increasing aggressiveness. The policies are evaluated by analysis of NFS traces from two real environments.

The simplest policy replicates read-only objects, avoiding all replica consistency issues. It allows 13%–20% of all operations to be offloaded, but almost all are metadata

requests—usually `Getattrs` for verifying client cache content freshness. This policy fails to enable offloading of most data transfer requests.

The best policy replicates read-only portions of read-mostly files in addition to read-only directories. This policy adds the ability to offload re-reads of files to which small changes (e.g., an append or metadata modification) have been made. Any such change will soon invalidate the entire file from the client cache, since freshness checks are done at whole-file granularity. Thus, client re-reads become server accesses, and this policy will allow them to be offloadable from the parent server. For some environments, including one of the two traces studied, such re-reads represent a significant portion of the entire server workload. Enabling offloads of such re-reads on this trace allows 75% of all operations and 52% of all data transfers to be offloaded. For both traces, this policy offloads almost all possible data transfers, given the read-only nature of Cuckoo offloading. The third policy additionally replicates high read:write ratio files, but offers little extra benefit.

The remainder of this paper is organized as follows. Section 2 overviews load balancing via layered clustering and discusses related work. Section 3 details the evaluation methodology. Section 4 describes the traces used in this study. Section 5 evaluates and compares the replication policies.

2. Background and related work

This section motivates our study of replication policies for layered clustering. It discusses approaches to load balancing, the Cuckoo model of layered clustering, and related work.

2.1. Load balancing and layered clustering

A balanced load benefits from the CPU, memory, disk, and network bandwidth capabilities of all servers, without one server becoming a bottleneck. This subsection describes three distributed file system architectures and how they provide for load balancing.

In traditional distributed file systems, such as NFS [5, 18] or CIFS [13], each server exports one or more file systems to clients. Clients mount exported file systems into their local namespace, allowing applications to use them transparently. Multiple servers can be used, with each one independently exporting file systems. Clients send requests for any given remote file to the server that exports the file system containing it. Thus, each server's load is dictated by the popularity of the portion of the namespace that it exports. Imbalance is common, and significant effort and downtime are needed to better balance loads. Specifically, portions of the namespace must be redistributed amongst the servers, and client mountpoints must be modified. Conventional wisdom says that such balancing is rarely performed.

In cluster file systems, a set of servers collectively export file systems to clients. Files and metadata are spread across the servers, via, for instance, striping [4, 9] or mapping tables [1], and clients send requests to the appropriate server(s). Cluster file systems have a number of advantages over the traditional model, including effective load balancing in most designs.

Layered clustering can offer some of the benefits of full cluster file systems, without their requirement for client-side changes and traditional server replacement. By sliding an intermediary component (the clustering switch in Figure 1) into place, existing servers are made to work together with no other changes. Unmodified clients continue to use traditional protocols (e.g., NFS or CIFS), and servers continue to own and export independent file systems. Intermediary software (or networking hardware) transparently translates selected requests into other requests of the same protocol. Responses to these requests are translated (if necessary) and delivered back to the original client.

2.2. The Cuckoo model of layered clustering

Cuckoo [12] is a layered clustering model in which file servers export file systems just as in traditional configurations, and each *parent* server remains the one authoritative repository. So, all traditional management activities remain unchanged.[1] The clustering switch uses a replication policy to periodically replicate popular objects onto one or more *surrogate*[2] servers based on analysis of past activity. An offloading policy is used to determine when a parent server is busy and what requests to offload from that parent server to surrogates with replicas. Requests are offloaded to replicas by simply mapping the filehandles, rerouting the request content, and fixing a few server-specific fields. Previous work [12] details how this load shedding can be done in about two thousand lines of C code.

The practical value of this approach depends mainly on how effective it is at providing the desired load balancing features. For load balancing in this manner to be effective, three things must be true of the environment. First, a significant fraction of a server's workload must be offloadable. Second, the process of offloading must be efficient. Third, it should be possible to determine when a server is busy and

[1] There is one exception to this. Cuckoo-style layered clustering assumes that all updates to served objects are via the client-server protocol passing through the clustering switch. If there is a local access path, the layered clustering infrastructure may act on out-of-date information because it doesn't see those updates. Note that this does not affect back-up and other read-only access.

[2] Cuckoo hens lay their eggs in the nests of other species, leaving those birds to incubate and raise the chicks as surrogate parents [3].

hence when requests should be offloaded from that server. This paper focuses on the first requirement, since prior work (e.g., [2, 12, 21]) has established the second, and the third is relatively straightforward.

A replication policy determines which objects to replicate, seeking to maximize the amount of a server's workload that is offloadable while avoiding undo implementation complexity. As such, our replication policies allow only read operations, including directory lookups and attribute reads, to be offloaded. Such a policy eliminates implementation concerns related to ensuring consistency by using the parent server when there are any updates. Previous studies indicate that such operations make up a large percentage of the requests in real environments. For example, Gibson et al. [8] report breakdowns for substantial NFS and AFS environments in which read operations comprise 85–94% of all requests and 78–79% of all server CPU cycles. Other file system workload studies [16, 20] report similar dominance of read activity. For read offloading to help, however, it is also necessary for a substantial fraction of these read requests to go to data that are modified infrequently—otherwise, the replicas will rarely be up-to-date when needed.

The use of Cuckoo-style layered clustering for read offloading can not provide as much load balancing as true clustering. In particular, all updates to objects must be performed at their parent server, bounding the set of offloadable operations. Still, as our evaluations show, it can effectively provide significant load balancing in traditional server environments. Further, it does so without the sharp investment and administrative burdens of transitioning from traditional servers to clusters.

2.3. Related work

There has been a huge amount of work in distributed file systems. Here, we focus on particularly relevant categories of related work.

Layered clustering builds on the proxy concept [17], using interpositioning to add clustering to an existing client-server protocol. Several groups and companies have developed layered clustering systems. For example, Rainfinity [15] offers "file switches" that aggregate standard file servers into a single namespace. Anypoint [21], Mirage [2], Cuckoo [12], and Katsurashima et al.'s "NAS switch" [11] are research systems that aggregate an ensemble of NFS servers into a single virtual server by redirecting client requests to the appropriate server. However, despite the existence of these systems, we are aware of no previous comparative studies of replication policies for them.

Layered clustering is analogous to the web server clustering support offered in some network switches, which can improve load balancing in Internet server farms [14]. Doing this for file servers does require more effort, because of frequent updates and long-lived interactions (either via sessions or via server-specific filehandles), but should provide similar benefits.

AFS [10] provides several administrative features that would make layered clustering less necessary. Most notably, a set of AFS servers provide a uniform namespace broken up into directory subtrees called volumes. The client's view of the file system is independent of which servers serve which volumes, and transparent volume migration is supported. Further, read-only volumes can be replicated on multiple servers, and clients can send their requests to any replica. Read offloading via layered clustering goes beyond this by allowing servers to shed load for read-write volumes as well, but is of greater value to less sophisticated (yet much more popular) systems like NFS and CIFS.

Another approach, sometimes used in large installations, is to have unmodified clients interact with a set of front-end file servers that share a collection of back-end storage components. Caching and consistency issues could be substantial, but are usually avoided by having different servers use different portions of each storage component. This approach can provide load balancing for the storage components as well as fault tolerance via "fail over."

3. Evaluation methodology

We analyze traces of real NFS activity to evaluate replication policies for Cuckoo-style load balancing. This section describes the metrics used, the methodologies for evaluating benefits, and limitations.

Metrics: We evaluate replication policies along three axes: offloadable operations, offloadable data transfers, and implementation complexity.

Offloadable operations refers to the percent of all operations sent to a parent server that can be offloaded to surrogates. Recall that all updates must be executed on the parent server, so they cannot be offloaded. Reads to replicated data and metadata can be offloaded. This metric is an indication of the amount of CPU work that can be saved at the parent server.

Offloadable data transfers refers to the percent of all data transferred from a parent server that can be offloaded to surrogates. This metric is an indication of the amount of disk work and network bandwidth that can be saved at the parent server.

Implementation complexity is a qualitative metric that refers to the intricacy of the replication policy and the work required to maintain replica consistency in the face of updates.

We determine the upper-bounds of offloadable operations and data transfers by assuming an offloading pol-

icy that offloads requests whenever possible, regardless of whether or not the parent server is busy. Assuming this "always offload" policy allows us to quantify the potential utility of each replication policy without interference from offloading policy artifacts.

Replication policy knowledge: Operations can only be offloaded if the data they access has been replicated to a surrogate. We envision periodic replication of data chosen by the replication policy, which means that the policy must make decisions with imperfect knowledge of the (future) request stream.

In evaluating replication policies, we consider two cases: oracular and history-based. The *oracular* replication results represent the best case for a replication policy in which decisions can be made knowing the future request stream. It eliminates ambiguity caused by imperfect predictions of which objects will meet a policy's criteria for replication. The *history-based* replication results represent one (simple) implementable approach to using history to predict future access patterns. Specifically, the behavior of period n is used as the prediction of period $n+1$.

Assumptions and limitations: Our replication policy comparisons focus on each policy's potential benefit to an overloaded parent server, ignoring issues that are independent of replication policy. For example, we do not consider the impact of offloading on surrogates. Also, we do not consider the effect of load placed on the system during replica creation. Such replica creation can be done during idle periods (e.g., overnight), which are common in the mid-sized environments targeted by layered clustering.

4. Traces used

We use week-long NFS traces from two real environments for our evaluations of replication policies. This section describes the two traces and some primary characteristics, highlighting aspects that help explain the evaluation results.

4.1. Environments traced

EECS: The EECS trace captures NFS traffic observed at a Network Appliance filer between Sunday, October 21st 2001 and Saturday, October 27th 2001. This filer serves home directories for the Electrical Engineering and Computer Science (EECS) Department at Harvard University. It sees an engineering workload of "research, software development, and course work"[6]. Typical client systems are UNIX or Windows NT workstations with at least 128 MB of RAM and locally installed system software and utilities. This environment is specifically noted *not* to contain e-mail or backup traffic. Detailed characterization of this environment can be found in [6].

Table 1. Operation breakdowns for the EECS and DEAS week-long trace periods.

	EECS		DEAS	
Total ops	28,742,622		198,646,288	
Upper-bound offloadable	67.2%		80.4%	
Operations seen to long-lived and created objects				
`Getattr`	19.9%	7.0%	25.7%	1.4%
`Lookup`	39.5%	0.7%	4.3%	0.1%
`Read`	7.0%	4.3%	49.7%	0.2%
`Readdir`	0.8%	0.1%	0.7%	0.0%
`Write`	2.5%	13.7%	14.7%	1.2%
Other	3.1%	1.4%	1.5%	0.5%
Total	72.8%	27.2%	96.6%	3.4%

DEAS: The DEAS trace captures NFS traffic observed at another Network Appliance filer at Harvard University between Sunday, February 2nd 2003 and Saturday, February 8th 2003. This filer serves the home directories of the Department of Engineering and Applied Sciences (DEAS). It sees a heterogenous workload of research and development *combined with* e-mail and a small amount of WWW traffic. Hence, the workload seen in the DEAS environment can be best described as a combination of that seen in the EECS environment and e-mail traffic. The volume of traffic seen by the DEAS filer over any given period exceeds that seen to the EECS filer by an order of magnitude. Detailed characterization of this environment (over different time periods) can be found in [7].

The e-mail traffic seen in DEAS affects the filer's workload greatly. The e-mail inboxes stored by the DEAS filer are always read sequentially from beginning to end. These sequential scans generate many `Read` operations, accounting for half of all operations seen to this server. These same sequential `Reads` also induce a large amount of data to be read from the DEAS filer. Since `Writes` to these e-mail inboxes are always appends, the ratio of `Reads` to `Writes` is very high. For all the policies we analyzed, these effects cause layered clustering to be more effective for DEAS than for EECS.

4.2. Trace characteristics

Because of their different activity levels and workloads, the DEAS and EECS traces exhibit different characteristics in terms of data transferred, distribution of NFS operations, and number of NFS operations seen. This section describes these different characteristics.

4.2.1. Operation breakdown. Table 1 shows the breakdown of NFS operations in the EECS and DEAS traces. The operations are binned into two categories: operations seen to

Table 2. Breakdown of data transferred during the EECS and DEAS week-long trace periods.

		Long-lived files		Created files		
Trace	*Total data*	*Data written*	*Data read*	*Populating data*	*Data written*	*Data read*
EECS 10/21/01–10/27/01	92.4 GB	5.5%	14.2%	1.4%	57.1%	21.8%
DEAS 02/02/03–02/08/03	321.1 GB	30.6%	54.4%	0.1%	12.8%	2.1%

long-lived objects and operations seen to newly-created objects. *Long-lived* objects are objects that exist at the beginning of the week-long trace period, whereas *created* objects are objects that are created during the week-long trace period.

The *upper-bound on offloadable operations* field of Table 1 lists the maximum percent of all operations that can be offloaded from the EECS or DEAS filers during these trace periods. This value assumes a scenario where replication is performed at the beginning of the week and the replication period is one week (nearly the same percentage is observed with daily replication). In this scenario, only long-lived objects will be considered for replication and so the upper-bound on offloadable operations is the sum of the read operations (i.e., `Getattrs`, `Lookups`, `Reads`, and `Readdirs`) seen to long-lived objects in Table 1. This value is 67.2% in the EECS trace and 80.4% in the DEAS trace. It represents the largest percent of offloadable operations that any replication policy can hope to achieve assuming Cuckoo-style layered clustering.

In the EECS trace, the majority of operations to created objects are `Writes`. Conversely, the majority of operations to long-lived objects are metadata operations such as `Getattrs` and `Lookups`. `Writes` dominate operations to created objects for two reasons. First, files in NFS are empty when initially created and must be populated by a set of *Populating `Writes`*[3]. Second, many created objects are temporary files and logs that see a large number of `Writes` and few or no `Reads`.

Unlike the EECS trace, where only 72.8% of operations access long-lived objects (versus created objects), 96.6% of all operations access long-lived objects in the DEAS trace. This difference is due to `Reads` seen in the DEAS trace to the long-lived e-mail inboxes.

Finally, we note that offloadable operations in the EECS trace are composed mostly of metadata operations (i.e., `Getattrs` and `Lookups`). In the DEAS trace, on the other hand, most offloadable operations are data accesses (`Reads`).

4.2.2. Data transferred breakdown. Table 2 shows the breakdown of data transferred within `Read` and `Write` operations in the traces. Both the contributions to total data transferred from/to long-lived and from/to created files are shown. Also shown is the contribution to total data transferred by populating `Writes` that are used to fill newly created files with data.

Due to the amount of read traffic seen to the e-mail inboxes, the DEAS trace presents a much greater opportunity for data transfer offloading than the EECS trace. The *upper-bound on offloadable data transfers* is the maximum percent of all data transfers that can be offloaded and is represented by the *data read from long-lived files* column of Table 2. This value is 14.2% in the EECS trace and 54.4% in the DEAS trace. Data read from created files is not offloadable since these objects could not have been replicated in advance.

5. Analysis of replication policies

This section describes and analyzes three replication policies, each more aggressive than the previous. The first only replicates read-only objects. The second additionally replicates read-only sections of files, even if they are not entirely read-only. The third additionally replicates read/write data, if the read:write ratio is above a threshold. The goal, of course, is to maximize offloadable operations and data transfers without excessive implementation complexity.

A replication policy identifies objects that should be replicated. We describe each policy in two parts: ranking scheme and criteria for replication. The *ranking scheme* imposes a partial order on objects being considered for replication. The *criteria for replication* specifies the minimum rank and other attributes an object must possess in order to be replicated.

5.1 Policy one: Replicate read-only objects

Ranking scheme: Rank objects in descending order by number of total operations seen.

Criteria for replication: Replicate the N highest-ranked objects that are read-only.

Policy one is simple and requires minimal work for consistency. Objects are ranked by the total number operations that they see. The N highest-ranked read-only objects are replicated. Any write operation to a replicated object is performed at the parent server and also invalidates all replicas of that object.

[3] Populating `Writes` are contiguous `Write` operations seen immediately after a `Create`. They populate a file with data.

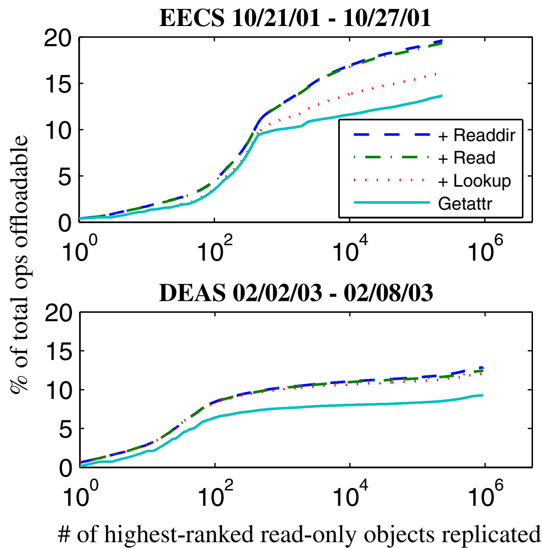

Figure 2. Policy one: offloadable operations.

Our evaluation of this policy finds that its simplicity prevents it from providing much benefit in terms of offloadable operations and data transfers. Though this policy does not show much promise, analyzing it yields several insights regarding what constitutes a good replication policy. We use these insights to formulate the next replication policy.

5.1.1. Oracular replication: offloadable operations. The graphs in Figure 2 show the percent of all operations that can be offloaded by replicating the N highest-ranked read-only objects. When all such objects are replicated, the parent server can offload 19.6% of all operations in the EECS trace and 12.9% in the DEAS trace.

Comparing these values to the upper-bounds on offloadable operations (67.2% and 80.4%, respectively, from Table 1), we see that policy one leaves much to be desired. It fails to achieve the additional 47.6% (EECS) or 67.5% (DEAS) of all operations that could be offloaded.

The majority of offloadable operations in both traces are metadata operations, specifically `Getattrs` and `Lookups`. Offloadable data operations (i.e, `Reads`) make up less than 4% of all operations in the EECS trace and less than 1% of all operations in the DEAS trace. `Getattrs` and `Lookups` are frequently issued by NFS clients in order to determine whether a locally cached object has become inconsistent. The large contribution of such metadata operations to the offloadable operations total suggests that policy one tends to replicate objects that are already effectively cached by NFS clients. The fact that very few data operations are seen to these replicas supports this observation.

Another issue with policy one is its ranking scheme, which prioritizes objects based on the number of operations. Not surprisingly, the objects that see the most operations are usually the ones that are cached effectively at the clients. Thus, when the number of objects replicated (N) is decreased, an even larger percentage of offloadable operations are `Getattrs` and `Lookups` issued by clients to check cache consistency.

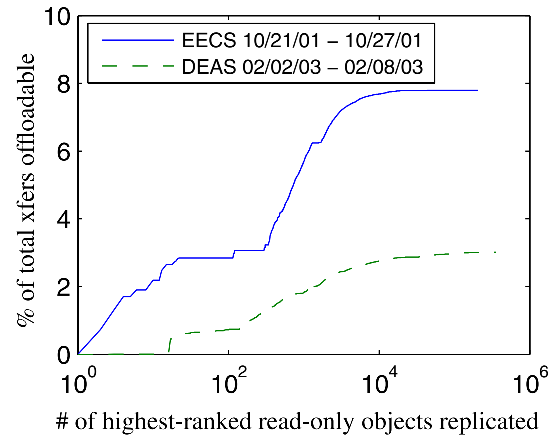

Figure 3. Policy one: offloadable transfers.

5.1.2. Oracular evaluation: offloadable data transfers. Figure 3 shows the percent of data transferred from objects that can be offloaded as a result of replicating the first N highest-ranked read-only objects. When all such objects are replicated, the parent server can offload 7.8% of all data transfers in the EECS trace and 3.0% in the DEAS trace. Comparing these values to the upper-bounds on offloadable data transfers (14.2% and 54.4%, from Table 2), we again see that policy one leaves much room for improvement: another 6.4% of all data transfers for EECS and another 51.4% for DEAS.

Since policy one tends to replicate objects that are very effectively cached by NFS clients, NFS clients are never forced to refresh their cached copy of these objects because they never change. Hence, any given client will usually only have to read data from an object replicated by this policy once. This overlap between client-cached objects and objects replicated by policy one prevents offloading of a sufficient percentage of data transfers to approach the upper-bound. Large amounts of read sharing (i.e., many clients reading the same data) would compensate for this shortcoming, but read sharing is not common in these traces (or most NFS environments). From this, we conclude that a replication policy that aims to offload a large percentage of data transfers should aim to replicate objects that cannot be effectively cached by clients.

5.1.3. Summary and analysis. Replication policy one stands out because it is simple and has minimal consistency re-

quirements. But, its simplicity comes at a price. We have shown that this policy has a tendency to replicate objects that are effectively cached by clients. As a result, few data transfers can be offloaded and most offloadable operations are metadata accesses.

Analysis of this replication policy yields important insights. Most important is that different objects should be replicated depending on whether the goal of a particular policy is to reduce a data transfer (disk or network) or a CPU bottleneck.

A replication policy that aims to alleviate a data transfer bottleneck should seek to replicate objects that cannot be cached effectively by NFS clients. This is because the act of caching is in itself a replication policy that attempts to minimize data transfers. As well, the ranking scheme should focus on data transfers rather than operations. If reducing CPU load is the primary concern, it is enough to replicate the objects that see the most operations.

Finally, analysis of policy one suggests that updates to metadata should not invalidate entire replicas; instead metadata should be kept consistent. This prevents a small metadata update from invalidating replicas that possess large read-only data sections.

The next section presents a replication policy that utilizes the insights gained from analysis of policy one. Specifically, a type of object that cannot be cached effectively by clients is identified and targeted.

5.2 Policy two: Replicate read-only directories and read-only sections of read-mostly files

Ranking scheme: Rank files in descending order by the amount of data read. Rank directories in descending order by total number of operations seen.

Criteria for replication: Replicate the read-only sections of the N highest-ranked files with read-only percentage (ROP) $\geq K$. A file's ROP (Equation 1) quantifies the percentage of a file's data that is not written. Also, replicate the M highest-ranked read-only directories.

$$ROP = \frac{File\,Size - Size\,of\,RW\,Sections}{File\,Size} \times 100 \qquad (1)$$

Policy two is slightly more complex than policy one in that some effort is required to keep the metadata of replicated objects consistent. Policy two extends policy one by additionally replicating read-only sections of some read-mostly files. Like policy one, writes to a replicated section of an object are performed at the parent server for the object and invalidate corresponding replicas. But metadata updates and writes to non-replicated sections cause the metadata of partially replicated objects to be updated (e.g., with new length and mtime information). This small increase in implementation complexity yields a large gain in offloadable operations and data transfers.

The insights gained from analysis of policy one are reflected in policy two's ranking scheme and criteria for replication. Most importantly, different ranking schemes and criteria are used for directories and for files. Since most directory accesses are metadata operations, policy two ranks directories by number of operations. Conversely, since files are the sole targets of data transfers, policy two implicitly assigns a high rank to files that are not cached effectively by clients. The criteria for replication for directories and files aim to replicate the subset of the highest-ranked objects that are easiest to keep consistent: read-only directories and read-mostly files. Read-mostly files are not completely read-only (read-only files are very effectively cached by clients), but rather files whose data consists primarily of read-only sections and a few read-write sections.

Read-mostly files are excellent candidates for replication in a NFS layered-clustering system that aims to maximize offloadable data transfers. These files cannot be cached effectively by clients, since the `Getattr` and `Lookup` operations used by clients to maintain cache consistency reveal only the last modified time of the *entire* file. A small write to a file forces clients that have parts of the file cached to discard all cached portions of it, regardless of whether or not those cached portions were actually made inconsistent by the write. Read-mostly files suffer most from this NFS caching limitation because clients are forced to re-fetch the large read-only sections whenever the small read-write sections are modified. As a result, replication of the read-only sections of read-mostly files should maximize the number of offloadable data transfers from a parent server.

For our evaluation of policy two, we assume that all read-only directories are replicated. This is because we have already evaluated policy two's ranking scheme and criteria for directories in our analysis of policy one. Our evaluation of policy two reveals that, when assuming oracular replication, it is able to offload a percentage of data transfers that approaches the upper-bound in both traces. With regard to offloadable operations, this policy performs better than policy one in both traces, but it performs especially well in the DEAS trace where it is able to offload a percentage of all operations that approaches the upper-bound. The performance of history-based replication in the DEAS trace is similar to that of oracular replication, but it fares less well in the EECS trace.

5.2.1. Oracular replication: offloadable operations. Figure 4 shows the percent of offloadable operations in both traces as a function of minimum ROP and the number of read-only sections of files replicated. The graphs in the top row show the contribution to offloadable operations by `Read` operations. The graphs in the bottom row show the contribution by `Getattr`, `Lookup`, and `Readdir` operations.

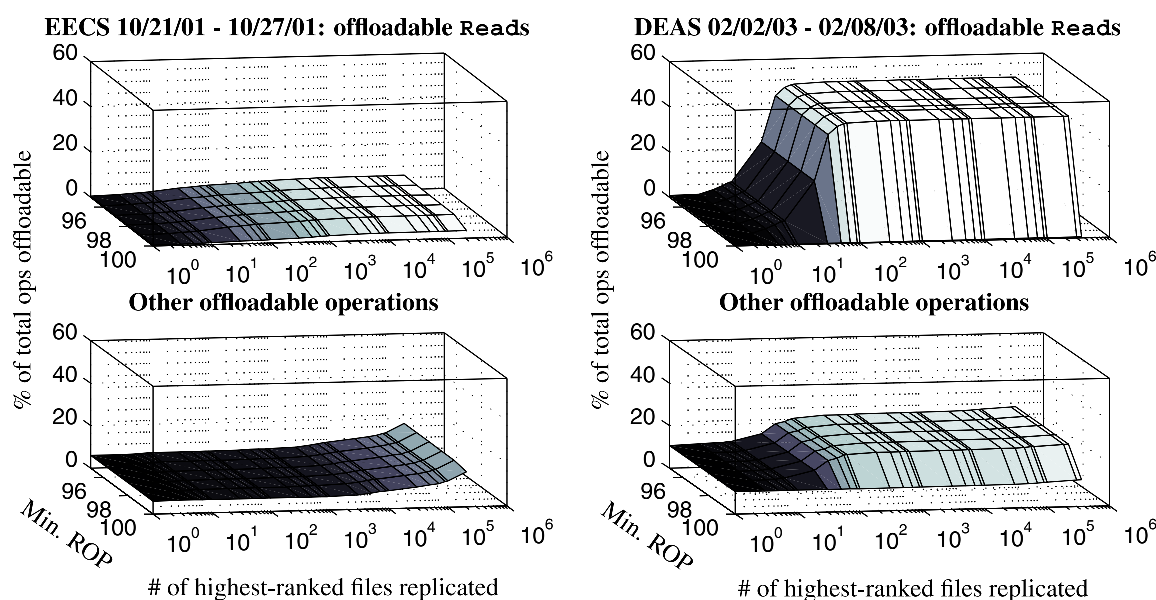

Figure 4. Policy two: offloadable operations.

The graphs show that there is minimal benefit in considering for replication the read-only sections of files with ROP $< 99\%$. Such files contribute negligible amounts to both offloadable metadata and data operations.

Replication of the read-only sections of just the one thousand highest-ranked files with ROP $\geq 99\%$ enables offloading 12.6% of all operations in the EECS trace and 72.6% of all operations in the DEAS trace. Replicating read-only sections of all files with ROP $\geq 99\%$ increases these values significantly (from 12.6% to 23.9%) in EECS and slightly (from 72.6% to 74.5%) in DEAS. Policy two fails to capture the additional 43.3% of all operations that could be offloaded in the EECS trace. However, only 5.9% more operations could be offloaded in the DEAS trace. In general, policy two performs better in the DEAS trace because a large fraction of the operations in DEAS are offloadable `Reads` which policy two is very adept at capturing.

5.2.2. Oracular replication: offloadable data transfers. Figure 5 shows the percent of data transfers in each trace that can be offloaded as a function of both the minimum ROP value and number of read-only sections of files replicated. The graphs show that replication of the read-only sections of just the one thousand highest-ranked files with ROP $\geq$ 99% enables this policy to offload a percent of data in both the EECS trace (12.6%) and the DEAS trace (51.6%) that approaches the upper-bound. Only 1.6% (EECS) or 2.8% (DEAS) more data transfers could be offloaded. When read-only sections of all files with ROP $\geq 99\%$ are considered for replication, these differences decrease to less than 2% for both traces.

The data transfer graphs highlight the benefit of replicating read-mostly files. The graphs show that read-only files (files with ROP == 100%) account for only a small percentage of offloadable data transfers regardless of the number of files considered for replication. When read-mostly files (files with ROP just less than 100%) are considered in addition to the read-only files, the graphs show a dramatic increase in offloadable data transfers, quickly approaching the upper-bound as more files are considered. This highlights an important insight: most of the data read from long-

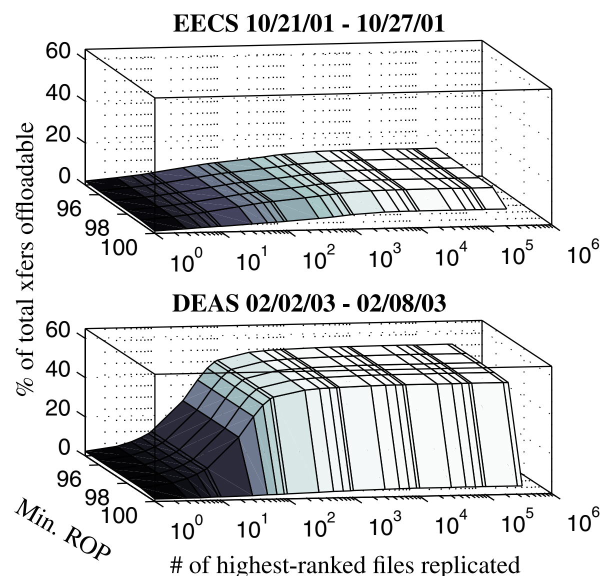

Figure 5. Policy two: offloadable transfers.

Table 3. Policy two: oracular replication vs. history-based replication (EECS).

	Monday		Tuesday		Wednesday		Thursday		Friday		Saturday	
	Oracular	History	Oracular	History	Oracular	History	Oracular	History	Oracular	History	Oracular	History
`Getattrs`	3.3%	3.4%	5.3%	3.5%	7.7%	7.4%	6.5%	5.4%	7.3%	5.2%	7.0%	6.4%
`Lookups`	1.6%	2.6%	9.8%	1.7%	6.8%	6.2%	10.3%	5.4%	1.5%	3.7%	7.4%	0.9%
`Readdirs`	0.2%	0.1%	0.5%	0.2%	0.3%	0.2%	0.5%	0.3%	0.2%	0.2%	0.2%	0.1%
`Reads`	6.0%	3.1%	6.1%	3.9%	4.9%	3.7%	5.1%	3.0%	4.0%	3.2%	6.4%	2.5%
Data transfers	10.2%	5.4%	8.6%	5.9%	25.3%	17.7%	21.4%	14.0%	9.9%	7.7%	14.0%	5.5%

Table 4. Policy two: oracular replication vs. history-based replication (DEAS).

	Monday		Tuesday		Wednesday		Thursday		Friday		Saturday	
	Oracular	History	Oracular	History	Oracular	History	Oracular	History	Oracular	History	Oracular	History
`Getattrs`	18.6%	17.0%	19.6%	19.8%	20.5%	19.6%	21.1%	16.5%	20.8%	16.7%	33.3%	31.7%
`Lookups`	1.2%	1.1%	1.3%	1.2%	1.6%	1.3%	5.9%	1.3%	5.8%	1.3%	2.2%	2.1%
`Readdirs`	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	1.5%	0.0%	1.3%	0.4%	0.0%	0.0%
`Reads`	53.1%	48.7%	53.0%	52.5%	51.0%	51.0%	45.8%	45.2%	42.9%	42.6%	41.2%	41.1%
Data transfers	63.5%	45.1%	63.3%	61.0%	71.0%	68.0%	70.3%	67.5%	74.3%	71.9%	65.5%	64.0%

lived files come from the read-only sections of read-mostly files rather than from the read-write sections.

5.2.3. History-based replication: offloadable operations and data transfers. Our evaluation of policy two, so far, has assumed perfect knowledge of the future request stream when choosing which objects should be replicated. This section looks at policy two when it must base replication decisions on analysis of past traces. Specifically, we consider the situation in which policy two makes replication decisions for day $n+1$ based on the files that meet the criteria for replication on day n. For this analysis, we assume that the read-only sections of the one thousand highest-ranked files with ROP $\geq$ 99% are replicated and that all read-only directories are also replicated. This evaluation offers an indication of how effectively policy two can be implemented in a real system.

Tables 3 and 4 compare the performance of policy two when using oracular replication and history-based replication for each day of the traces except Sunday (day one). The performance of history-based replication is very similar to that of oracular replication in the DEAS trace. For the EECS trace, however, history-based replication yields only mediocre results.

In four cases, history-based replication offloads more `Lookups` or `Getattrs` than oracular replication. This is an artifact of the ranking scheme and criteria used for directories by policy two. History-based replication replicates directories known to be read-only on day n and allows offloading on day $n+1$ until the directory is written, whereas oracular replication can only replicate directories that are read-only on day $n+1$. Hence, history-based replication can potentially offload more operations than oracular replication for directories that are read-only on day n and read-write on day $n+1$.

Performance of any type of replication (oracular or history-based) is limited on the EECS trace, because its metadata heavy workload is not very conducive to offloading. Though we believe that a better history-based prediction scheme would yield results closer to oracular for the EECS trace, the key insight here is that when layered clustering offers substantial benefits (as with the DEAS trace), a simple history-based replication scheme is sufficient to capture these benefits.

5.2.4. Summary and analysis. Policy two requires a small amount of effort to keep metadata consistent on replicas. However, by replicating a small number of read-only sections of read-mostly files, policy two is able to offload a percentage of data transfers that approaches the upper-bounds for both traces. With regard to offloadable operations, policy two outperforms policy one both the EECS and DEAS traces and approaches the upper-bound in the DEAS trace.

Finally, our analysis of history-based replication shows that, when many offloadable data transfers exist (as in DEAS), history-based replication exposes them effectively.

5.3 Policy three: Replicate read-only directories and read-only/read-write sections of files

We analyzed a third policy, which extends policy two to replicate read-write sections of files if the read-write sections exhibit a high ratio of data read to data written. It is

more complicated than policy two in that it requires both data and metadata sections of replicas to be kept consistent. Due to the additional data replicated, we expected that this policy would be capable of offloading more operations and data transfers, but we found that policy three performs no better than policy two on both traces.

The extra ability of policy three to replicate read-write sections did not yield a benefit because, in both traces, the majority of offloadable operations and data transfers for long-lived files come from their read-only sections. Additionally, very few of the read-write sections of long-lived files exhibit a high read-write ratio. Simply put, policy three offers more expressive power than is required for the EECS and DEAS traces.

6. Summary

Layered clustering is a promising method for balancing load across unmodified NFS or CIFS servers, when using the right replication policy. Although simple replication of read-only objects offers minimal potential benefit, replication of read-only portions of read-mostly objects can offer significant opportunities for offloading of all requests (24%–75%) and all data transfers (14%–52%). The key insight is that this replication policy captures objects that are invalidated in client caches, allowing re-reads to be offloaded. By doing so, this replication policy provides a good balance between offloadable work enabled and effort required to maintain consistency.

Acknowledgements

We thank Dan Ellard and Margo Seltzer for sharing the NFS traces. We thank Mike Mesnier for thoughtful comments. 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, and Sun) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, and by the Army Research Office, under agreement number DAAD19–02–1–0389.

References

[1] T. E. Anderson, et al. Serverless network file systems. *ACM Transactions on Computer Systems*, **14**(1):41–79. ACM, February 1996.

[2] S. Baker and J. H. Hartman. *The Mirage NFS router*. Technical Report TR02–04. Department of Computer Science, The University of Arizona, November 2002.

[3] M. Brooke and T. R. Birkhead. *The Cambridge encyclopedia of ornithology*. Cambridge University Press, 1991.

[4] L.-F. Cabrera and D. D. E. Long. Swift: using distributed disk striping to provide high I/O data rates. *Computing Systems*, **4**(4):405–436, Fall 1991.

[5] B. Callaghan, et al. *RFC 1813 - NFS version 3 protocol specification*. RFC–1813. Network Working Group, June 1995.

[6] D. Ellard, et al. Passive NFS tracing of email and research workloads. Conference on File and Storage Technologies. USENIX Association, 31 March–02 April 2003.

[7] D. Ellard and M. Seltzer. New NFS tracing tools and techniques for system analysis. Systems Administration Conference. Usenix Association, 26–31 October 2003.

[8] G. A. Gibson, et al. File server scaling with network-attached secure disks. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. Published as *Performance Evaluation Review*, **25**(1):272–284. ACM, June 1997.

[9] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. ACM Symposium on Operating System Principles. ACM, 5–8 December 1993.

[10] J. H. Howard, et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, **6**(1):51–81. ACM, February 1988.

[11] W. Katsurashima, et al. NAS switch: a novel CIFS server virtualization. IEEE Symposium on Mass Storage Systems. IEEE, 7–10 April 2003.

[12] A. J. Klosterman and G. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[13] P. J. Leach. *A Common Internet File System (CIFS/1.0) Protocol (Working Draft)*. Technical report. Internet Engineering Task Force, December 1997.

[14] V. S. Pai, et al. Locality-aware request distribution in cluster-based network servers. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **33**(11):205–216. ACM, 3–7 October 1998.

[15] Rainfinity. www.rainfinity.com.

[16] D. Roselli, et al. A comparison of file system workloads. USENIX Annual Technical Conference. USENIX Association, 18–23 June 2000.

[17] M. Shapiro. Structure and encapsulation in distributed systems: the proxy principle. International Conference on Distributed Computing Systems. IEEE, Catalog number 86CH22293-9, May 1986.

[18] Sun Microsystems, Inc. *NFS: network file system protocol specification*. RFC–1094. Network Working Group, March 1989.

[19] C. A. Thekkath, et al. Frangipani: a scalable distributed file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **31**(5):224–237. ACM, 1997.

[20] W. Vogels. File system usage in Windows NT 4.0. ACM Symposium on Operating System Principles. Published as *Operating System Review*, **33**(5):93–109. ACM, December 1999.

[21] K. G. Yocum, et al. Anypoint: extensible transport switching on the edge. USENIX Symposium on Internet Technologies and Systems. USENIX Association, 26–28 March 2003.