# Data Propagation in a Distributed File System

Minkyong Kim*, Brian Noble*, Xichu Chen+, and Dawn Tilbury+
*Department of Electrical Engineering and Computer Science
+Department of Mechanical Engineering
University of Michigan
{minkyong,bnoble,xichuc,tilbury}@umich.edu

## Abstract

Distributed file systems benefit greatly from optimistic replication that allows clients to update any replica at any site. However, such systems face a new dilemma: Once data is updated at a replica site, when should it be shipped to others? In conventional file system workloads, most data is read by its writer, so it needs to be shipped only for administrative reasons. Unfortunately, shipping on demand substantially penalizes those who do share, while shipping aggressively places undue load on the system, limiting scalability.

This paper explores a mechanism to predict and support sharing in a wide area file system. This mechanism uses the observation that updates that invalidate cached objects elsewhere are likely to be shared, and they should be propagated. The mechanism is evaluated on its precision, recall, and F-measure over traces capturing live file system uses. It reduces data shipped by orders of magnitude compared to aggressive schemes, while reducing the performance penalty of on-demand shipment by nearly a factor of two.

## 1 Introduction

Wide-area file systems, particularly those meant for mobile clients [9, 12, 6, 17], have adopted *optimistic concurrency control*. In such systems, copies of files are replicated across many clients and servers, and any client can update any replica at any time. While this allows the possibility of conflicting updates, these systems assume that conflicts are so rare that the infrequent cost of merging versions is repaid by gains in performance and availability. This assumption has been borne out by empirical evidence [15].

When an optimistic update is made at a replica site, that site must eventually inform others of the new version's existence for consistency. However, sharing is rare in file system workloads. It is likely that a file written by a particular user will be read only by that user. Consequently, while the meta-data *describing* an update needs to be propagated, the data *comprising* that update often need not be. Systems that aggressively propagate the update contents do so at the expense of scalability.

This observation suggests an obvious optimization: avoid propagating update contents whenever possible. Instead of propagating aggressively, replica sites should move data only for administrative reasons—such as backup—or in instances of true sharing. This drastically reduces data movement compared to aggressive schemes, charging the cost of sharing to those who require it.

Unfortunately, these costs can mount quickly. In a wide-area deployment, there can be substantial latency to fetch a file from the replica on which it was last updated. Given the instabilities seen across paths with large bandwidth-delay products [14], this can have profound effects in cases of active sharing.

These concerns raise the question of when to propagate data objects. We will show that propagating aggressively wastes server resources, while propagating only on-demand unduly penalizes clients who share. In this paper, we explore mechanisms for deciding when to propagate an update, and their impact on the wide-area file system Fluid Replication [8].

Our propagation mechanisms are based on the assumption that past instances of sharing are likely to lead to future ones. When an update is made at one replica site, it may invalidate cached copies at others. If so, those other sites are likely to access the file again, so it should be propagated in advance. This is called the *invalidation heuristic*.

To understand the performance of the invalidation heuristic scheme and how it compares to the naive approaches, we used two sets of three-month traces of file references. The first three-month trace, from the University of Michigan, was collected by observing NFS traffic

to a file server hosting 86 home directories comprising over 35 GB of data. The second trace, from Harvard University, contains traffic to and from an NFS server that serves as the primary home directory for the EECS department.

Each scheme was given the trace as input, and measured along two dimensions. The first metric is *precision*, the degree to which unnecessary propagations are excluded. The second metric is *recall*, the degree to which necessary propagations are made before they are needed. The harmonic mean of these two is called *F-measure*.

We begin by describing Fluid Replication, the system providing the context for our work. We then describe our approaches to determining whether or not to propagate data, and the traces used to evaluate each scheme. The evaluation shows that the heuristic approach provides the best balance between precision and recall. This scheme reduces data shipped by orders of magnitude compared to aggressive schemes, while reducing the penalty of on-demand shipment by nearly a factor of two.

# 2 Fluid Replication

Fluid Replication is a wide-area file system supporting mobile clients. It follows a two-tier, optimistic replication model. The ultimate "truth" of a client's files is stored on its *home server*. When the client is in its home domain, it interacts with its home server as an AFS client does [7]. However, when the client roams outside of its domain, the latency to the home server imposes a substantial performance penalty. Representative workloads can increase by as much as a factor of two with additional network latency of only 30 milliseconds [8].

To absorb the cost of wide-area updates, a roaming client attach itself to a nearby *WayStation*, a second-tier replica of the client's file system. WayStations are presumed to be well-connected to the network at large, and given a high level of administrative care and scrutiny. All read and write requests are satisfied by the WayStation. If a read requests an uncached file, the WayStation must first fetch it from the home server.

The server records the list of WayStations that have cached each file. After accepting an update, the server informs each caching WayStation that its version is out of date. This message is asynchronous; it need not be sent before the update is accepted. Likewise, WayStations asynchronously inform servers of updates that they have accepted. Updates are batched for a short time to amortize update bursts [20]. With even a conservative delay of 15 seconds between accepting an update and informing other replica sites, conflict rates observed in real file system workloads are a modest 0.01% [8].

Meta-data, in the form of version invalidation, is propagated quickly. This gives good consistency with modest overhead, since invalidations are small and easily processed. In contrast, updated files are large. Shipping them needlessly incurs substantial costs at the WayStation and the server, limiting scalability.

To prevent unnecessary propagations, WayStations can defer shipping updated files. If a client requests a file that is not at the server, it must be *backfetched* from the WayStation storing it. Propagating updated files too slowly increases client latency, while propagating them too aggressively increases server load. So we need a way to propagate updates that balances the two.

# 3 Simple Propagation Schemes

To put our propagation schemes in context, we first describe three naive attempts to decide when to ship updates. The first, *write-through*, aggressively ships all updates as they are made. The second, *on-request*, ships them only when they are needed elsewhere. The third, *periodic*, batches updates and ships them every $t$ minutes.

One way to propagate updates is to send them as soon as they are made. We call this the write-through scheme. Using this scheme, we get the upper bound for the amount of data propagated from WayStations to the server. Because all updates are at the server, no file is backfetched. The drawback is that many updates are propagated unnecessarily, increasing server load and network traffic.

At the other end of the spectrum, updates can be propagated only when they are needed elsewhere. This is called the on-request scheme. Using this scheme, we get the lower bound for the amount of data propagated, because files are shipped only when necessary. The drawback with this scheme is that clients may experience slow response time when requested files are at remote WayStations.

A simple improvement to the write-through scheme is to defer propagation, batching updates together and sending them periodically. This scheme is used in many systems, including our early Fluid Replication prototype [8]. Periodic schemes with long periods can ship substantially less data than the write-through scheme. This is because writes are bursty [20]—they tend to be clustered together in time. If an update is made but not shipped, a later update can overwrite it, which can lead to savings. Unfortunately, these periodic schemes are not very precise, because they propagate files indiscriminately; many of the propagated files are never accessed by other clients.

# 4 Making Propagation Decisions

The on-request scheme gives rise to backfetches, while the write-through scheme propagates files needlessly. The periodic schemes reduce this, but are either still too aggressive or not able to eliminate most backfetches. To address these problems, we introduce a new propagation scheme, called invalidation heuristic. We also discuss briefly two other schemes that we considered but rejected: Time to Next Use, and Adaptive Time Delay.

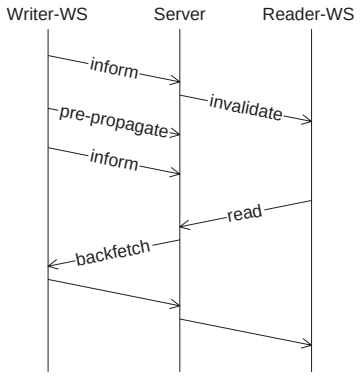
## 4.1 The Invalidation Heuristic Scheme

File accesses tend to have strong temporal locality. If files were shared in the recent past, they are likely to be shared in the near future. Based on this observation, updates should be propagated if the file is shared. When a WayStation informs the server of its updated files, the server invalidates copies at other WayStations. Updates that cause invalidations are propagated to the server immediately, because they are likely to be read again where they have been invalidated. We call this the *invalidation-heuristic* (IH) scheme. Note that this scheme requires no extra state at the server; the list of files to be propagated can be computed from the same state used to issue invalidations.

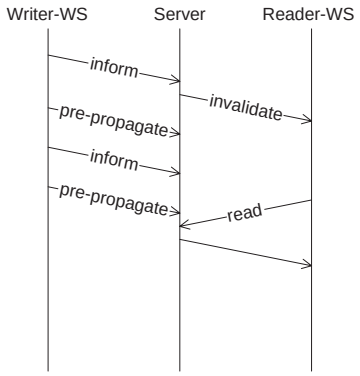
This simple approach has a drawback. Consider a scenario in which files are always written twice before other nodes access them. When the WayStation informs the server of its first write, the server invalidates all the other copies. At this point, the file is no longer shared. Therefore, only the first write of two is propagated, and every read requires backfetching. Figure 1(a) shows this scenario.

To solve this problem, we keep the same sharing status for a fixed number of bursty writes. This number is called *history* and determines how many writes of each burst are propagated to the server. The *burst* counter is reset when another node reads the file or writes to it. We considered *history* values from one to four in our evaluation. Figure 1(b) shows what happens in the above scenario when history is set to two. Compared to Figure 1(a) where history is one, the server need not backfetch the update, and the read request is served faster. Figure 2 summarizes the decision making process.

Another optimization is to consider spatial locality. If many files are backfetched from a particular WayStation, the server should prefetch other files that have been updated then, but not yet propagated. This mechanism is especially effective for producer-consumer relationships; one node has updated many files, and others read them later. More specifically, pending writes are prefetched if




(a) History = 1




(b) History = 2

This figure shows a reader accessing a file after a writer updates it twice. With history of one, a backfetch occurs since only the first update to the file is propagated. With history of two, a backfetch is avoided since the first two updates are propagated to the server.

Figure 1: Example of IH Scheme

the number of backfetches to the WayStation, $b$, exceeds the number of writes overwritten by the WayStation, $o$, by a certain limit, $L$:

$$b - o > L$$

When this condition is met for a WayStation, pending writes that happened within the last 30 minutes are propagated to the server. We call this modified scheme *invalidation-heuristic-with-spatial-locality* (IHSL).

## 4.2 Other Schemes Considered

Rather than using past sharing to predict future sharing, one can use the times of recent accesses to predict the next time that a read or write might occur. The *time-to-next-use* (TTNU) scheme decides whether to propagate or not based on the past history of file access patterns. The basic

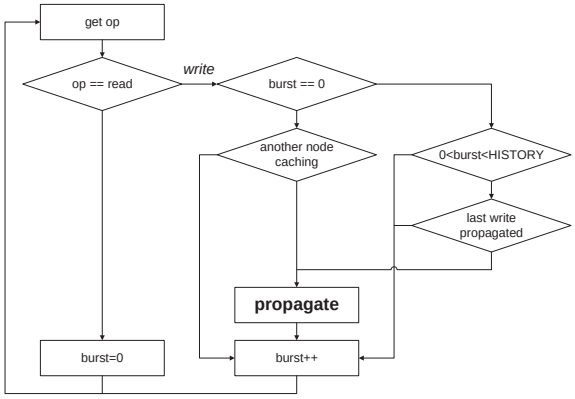



Figure 2: Decision Making in the IH Scheme

idea is that a modified file should be propagated if the expected time to the next read by another node is earlier than the expected time to the next write by the current node. The expected next access times are computed based on the past access pattern using exponentially weighted moving average filters.

Another scheme we considered is called the *adaptive-time-delay* (ATD) scheme. Static periodic schemes can be too aggressive, not effective, or both. This is because a single, static period is not capable of capturing the different access patterns between files and over time. ATD is based on the periodic scheme; it maintains a period estimate for each file, and adjusts this estimate based on the utility of past propagation decisions. By observing the benefit of prior propagation decisions, this feedback controller produces per-file estimates of need.

# 5 Evaluation

## 5.1 Trace Collection

To understand how propagation schemes work in a real environment, we used two sets of traces, one from the University of Michigan (UM) and another from Harvard University. The first set of traces was collected on an NFS server in our department from April to June of 2002. This machine serves the home directories for 86 users; they occupied 35.7 GB as of August 12, 2002. Using `tcpdump`, we collected NFS requests to the server and the corresponding responses. Then, we used `nfstrace` [2] to convert the traces to user level file system commands (reads and writes).

The second set of traces is from the EECS department at Harvard University [4]. They were taken from September to November of 2001. The NFS server is the primary file system for the department. Because traces are not available for all of these days, we use two consecutive weeks from each month to evaluate the propagation schemes.

| Month | Read | Write | Machine | File |
|---|---|---|---|---|
| **UM Trace** | | | | |
| 4 | 95,427 | 301,905 | 197 | 98,588 |
| 5 | 88,295 | 211,058 | 199 | 96,687 |
| 6 | 180,448 | 306,233 | 197 | 57,793 |
| **Harvard Trace** | | | | |
| 9 | 439,256 | 895,804 | 117 | 276,494 |
| 10 | 434,259 | 945,294 | 108 | 285,848 |
| 11 | 254,019 | 854,775 | 142 | 312,049 |

Table 1: Summary of Traces

Table 1 shows the number of reads and writes, the number of distinct machines that accessed the NFS server in question, and the number of files accessed during each month. The number of writes is equal to the number of propagations if we use the write-through scheme. Note that for the UM traces, the number of different machines that accessed the NFS server during each month is bigger than the number of user accounts which have home directories at the server. This is likely due to users using different machines from one session to another.

In applying these traces, we assume that each client machine is in a different location, and uses a different WayStation. This will tend to over-emphasize the costs of wide-area operations. A discrete event simulator replayed the traces, counting updates, reads, propagations, and backfetches. One can apply this simulator to different propagation policies, measuring their potential gain.

Using these file system traces, we set out to answer the following questions:

- What percentage of writes are read by others?
- How much does a client suffer from sharing when the on-request scheme is used?
- What is distribution of sharing over machines?
- For each scheme, how many files are propagated compared to write-through?
- For each scheme, how many files are backfetched compared to on-request?
- How effective is each scheme in terms of precision and recall?
- Which scheme produces the best overall performance?

| Month | File | Backfetch | Backfetch to write |
|---|---|---|---|
| **UM Trace** | | | |
| 4 | 1,223 | 2,438 | 0.81% |
| 5 | 463 | 1,142 | 0.54% |
| 6 | 738 | 1,648 | 0.54% |
| **Harvard Trace** | | | |
| 9 | 12,273 | 34,595 | 3.86% |
| 10 | 6,808 | 29,419 | 3.11% |
| 11 | 3,477 | 23,361 | 2.73% |

Table 2: Backfetch

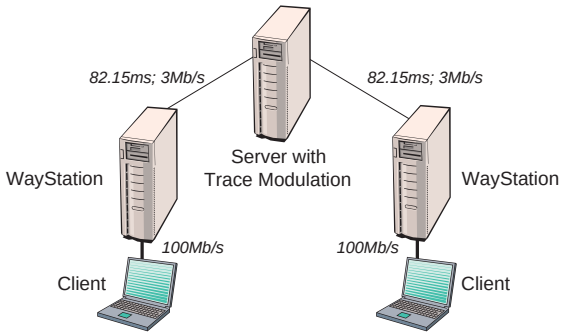



Figure 3: Experiment Setup

## 5.2 Aggressive Propagations

Table 2 shows backfetches when the on-request scheme is used. The second column shows the number of distinct files that are backfetched; the last column shows the ratio of backfetches to writes. There are two things to note about this data. First, the number of distinct backfetched files is smaller than the number of backfetches. From this, we can infer that the files that are shared once are likely to be shared again. Second, less than 4% of writes are read by others. So propagating all updates introduces unnecessary network traffic between the server and WayStations. In short, the write-through scheme is too aggressive.

## 5.3 Passive Propagations

Our goal for propagation schemes is to help those clients who suffer under passive propagations due to sharing. Before exploring new schemes, we need to consider whether clients actually suffer or not under the on-request scheme. To answer this, we chose one hour with substantial sharing from the Harvard traces and one from the UM traces, and measured the time a client spends to fetch updated files from other nodes.

The network condition between the server and WayStations are slow (latency of 82.15 ms and bandwidth of 3 Mb/s), while that between WayStations and clients are fast (100 Mb/s). The WayStations are connected to the server via a *trace modulated* network. Trace modulation performs emulation of a slower network over a LAN [16]. To get the latency and bandwidth parameters, we ran `ping` from a machine in our department to a machine at Hewlett Packard with different packet sizes. The average round trip time with packet size of 20 MB was 217.9 ms; the time with packet of eight bytes was 164.3 ms. We used half of 164.3 ms, which is 82.15 ms, as the one-way latency. This gives us bandwidth of 3 Mb/s using the equation $delay = latency + size/bandwidth$ [3]. Figure 3 shows the experiment setup.

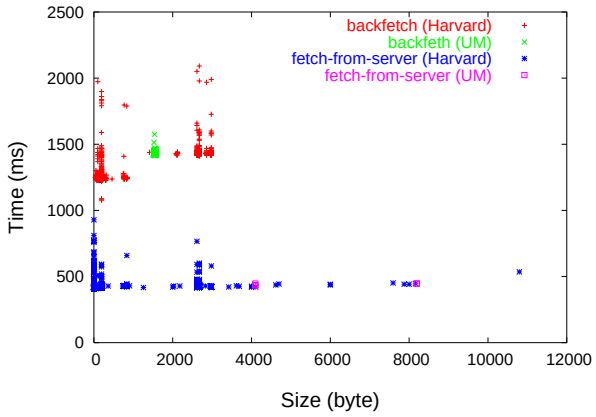
| | Count | Size(MB) | Time(s) |
|---|---|---|---|
| **UM Trace** | | | |
| Read-hit | 0 | 0 | 0 |
| Fetch-from-server | 5 | 0.0287 | 2.2 |
| Backfetch | 138 | 0.2132 | 197.8 |
| Total | 143 | 0.2419 | 200.0 |
| Backfetch-to-total | | | 98.9% |
| **Harvard Trace** | | | |
| Read-hit | 487 | 0.8761 | 0.077 |
| Fetch-from-server | 1607 | 1.0762 | 696.5 |
| Backfetch | 1808 | 30.2044 | 2382.1 |
| Total | 3092 | 32.1567 | 3078.7 |
| Backfetch-to-total | | | 77.4% |

Table 3: Total Fetch and Backfetch Time

Using the traces, we measured the time that one client spent fetching files. Updated file data is never sent to the server unless other nodes request it, causing backfetches. The goal of this experiment is to find the percentage of time spent backfetching files from other WayStations.

Table 3 shows the time with cold cache; no files are cached at the WayStation. *Read-hit* shows the time to fetch files cached locally. *Fetch-from-server* shows the time to fetch files at the server. *Backfetch* shows the time to fetch files that require backfetching from the writer, because the server does not have the up-to-date copy. Backfetch time is thus the time to ship a file from a remote WayStation to the server, plus the time to ship the file to the local WayStation, and finally to the client. In the UM trace, the client spent 98.9% of total fetch time waiting for the files that needed to be backfetched. It spent 77.4% in the Harvard trace. So the on-request scheme penalizes highly the clients who share.

With a warm cache, no file will be fetched directly from the server; the files that required fetching from the server with a cold cache will be at the local WayStation with a




This figure shows the fetch and backfetch time measured at the client for individual files under latency of 82.15 ms and bandwidth of 3 Mb/s.

Figure 4: Fetch and Backfetch Time for Individual File

warm cache. So the total time will be less while the backfetch time remains the same. Therefore, we can predict that the ratio of backfetch to total time will be even higher with a warm cache than with a cold cache.

Figure 4 shows the fetch and backfetch times measured at the client for individual files. Backfetching takes about three times longer than fetching from the server, for files of the same size.

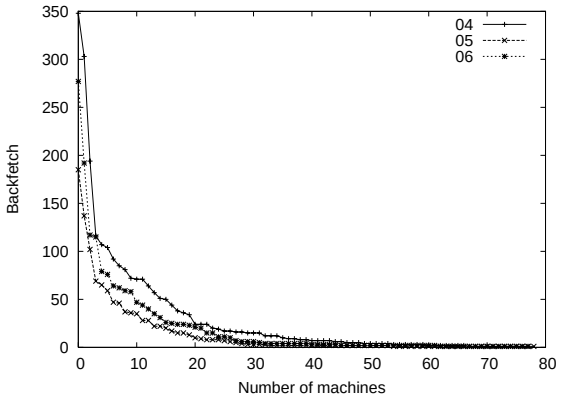
## 5.4 Burstiness

Figure 5 shows the distribution of backfetches over machines. The x-axis shows the number of machines; the y-axis shows the number of backfetches each machine experienced. The machines that did not experience any backfetches are excluded from these graphs.

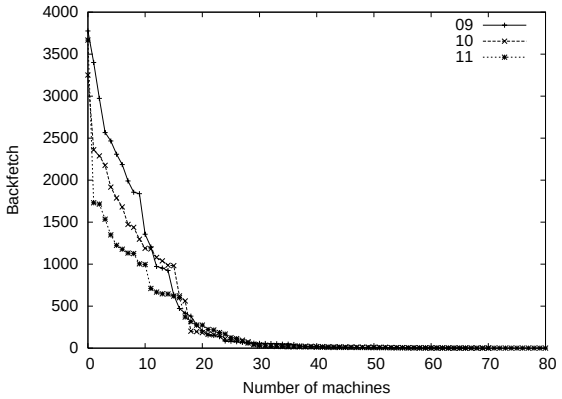
The backfetches are not distributed evenly over all machines. They are highly concentrated among a few machines. For the UM trace, the top three machines are responsible for more than one third of all backfetches: 34.7%, 37.1% and 35.6% for the months of April, May, and June, respectively. For the Harvard trace, the top three are responsible for more than one fourth: 29.3%, 26.9%, and 30.4% for the months of September, October, and November, respectively. Thus, most machines experience no or few backfetches, while some machines experience many.

## 5.5 Files Propagated and Backfetched

We built a simulator to compare the different propagation schemes. The simulator takes traces as input and returns a summary of data transferred between the server and WayStations.




(a) UM



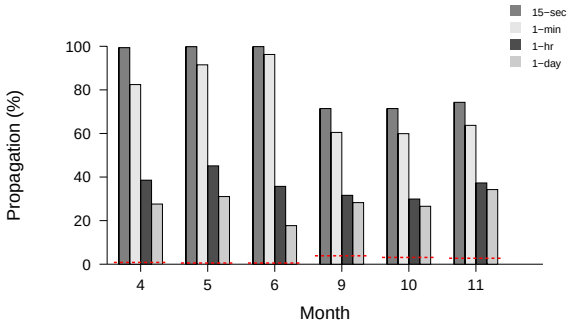

(b) Harvard

This figure shows the distributions of backfetches over machines. The x-axis shows the number of machines; the y-axis shows the number of backfetches each machine experienced.

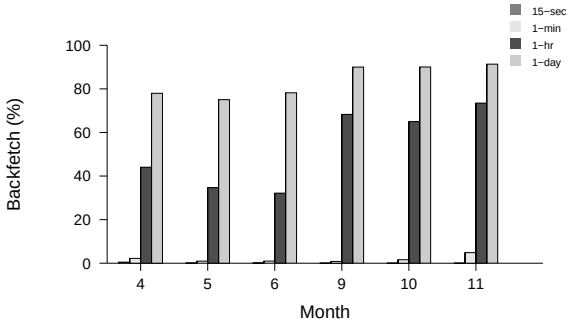
Figure 5: Backfetch Distribution

To evaluate the family of periodic schemes, we subjected several of them to our traces. We tried periods of 15 seconds, 1 minute, 1 hour, and 1 day. Figure 6 shows the percentage of propagations normalized over the write-through scheme, and the percentage of backfetches normalized over the on-request scheme. Note that the propagations include both propagations done before the data is actually needed elsewhere (pre-propagations) and propagations caused by backfetches. The x-axis shows the month that the trace was taken; 4 to 6 denote the UM traces and 9 to 11 represent the Harvard traces. The horizontal dotted lines in Figure 6(a) represent the values for the on-request scheme that serve as the lower bounds.

The periodic-15-second scheme propagates only



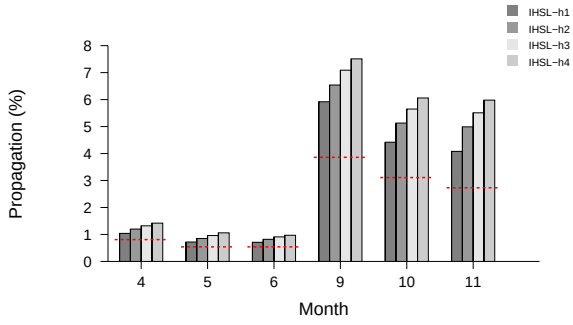

(a) Propagations



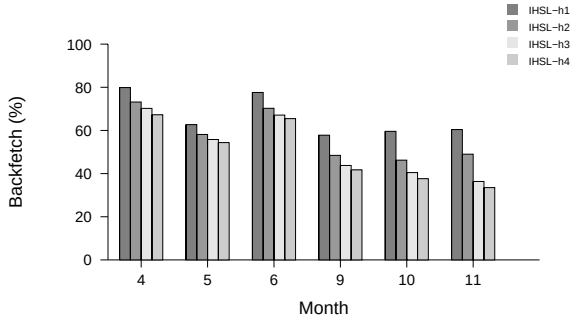

(b) Backfetches

This figure shows the percentage of propagations normalized over the write-through scheme, and the percentage of backfetches normalized over the on-request scheme. Updates are propagated every 15 seconds, 1 minute, 1 hour, and once a day. The dotted lines denote lower bounds.

Figure 6: Performance of Periodic Scheme




(a) Propagations




(b) Backfetches

This figure shows the percentage of propagations to the write-through scheme and the percentage of backfetches to the on-request scheme. Performance of the IHSL scheme with history of 1, 2, 3 and 4 is shown. The horizontal dotted lines denote lower bounds.

Figure 7: Performance of IHSL Scheme

slightly less than the write-through scheme for the UM trace; it propagated 99.7% of the write-through scheme on average. It does better for the Harvard trace, propagating 72.4%. The periodic-1-day scheme sends much less data, but causes almost as many backfetches as the on-request scheme. Specifically, this scheme propagates 25.5% and 29.7% on average for UM and Harvard traces respectively, but these are still much more than the on-request scheme, which propagates 0.6% and 3.2%.

Figure 7 shows total propagations and backfetches during each month for the invalidation-heuristic-with-spatial-locality (IHSL) scheme with history values ranging from one to four. For the UM trace, IHSL propagates less than 1.5% of the data shipped by the write-through scheme. For the Harvard trace, it propagates less than 8%. This scheme propagates less data than the 1-day periodic scheme, but leaves fewer backfetches to clients.

The on-request scheme propagates the minimum amount of data—two orders of magnitude less than the write-through scheme—but penalizes clients with substantial sharing. In contrast, IHSL propagates less than twice that of the on-request scheme for all months except for November, but reduces backfetches by 43.5% on average. For November, the IHSL scheme with history of three and four propagated a little more than twice that of the on-request scheme.

All IHSL schemes with different history values outperformed the simple schemes (write-through, on-request and periodic), but it is hard to compare them to each other because schemes with fewer propagations often cause more backfetches. How can we tell which scheme is best? In the following section, we introduce a new metric to compare our propagation schemes.

### 5.6 Effectiveness

Our goal is to minimize both server load and client response time, but improving one often penalizes the other. For example, a scheme that propagates aggressively may reduce client response time, but increase server load. So it is difficult to tell which propagation scheme is best.

To measure effectiveness, we introduce two metrics, *precision* and *recall*, which are often used in information retrieval. In the context of information retrieval, precision is the fraction of the relevant documents that has been retrieved, and recall is the fraction of the retrieved documents that is relevant [1].

In our system, precision refers to the fraction of accessed file versions which has been pre-propagated, and recall refers to the fraction of pre-propagated files which has been accessed. Note that we use the term *version* to refer to a file at a specific time. We excluded all the accesses that did not require propagations. In other words, we do not include reads to files that are never updated or reads to files that are only updated locally.

Let Set A be the accessed file versions that are modified elsewhere, and Set B be the file versions that are pre-propagated to the server. Figure 8 illustrates these basic sets. Precision ($p$) and recall ($r$) are defined as follows:

$$p = \frac{|A \cap B|}{|B|} \quad (1)$$

$$r = \frac{|A \cap B|}{|A|} \quad (2)$$

One way to combine precision and recall is the harmonic mean of two, called the *F-measure*. F-measure was introduced by van Rijsbergen [18], and is an established metric for information retrieval. The degree to which two sets do not match is defined as *E-measure*. It is the shaded area in Figure 8. With normalization, it is written as:

$$E = \frac{|A \cup B - A \cap B|}{|A| + |B|} \quad (3)$$

In term of $p$ and $r$, $E$ is written as:

$$E = 1 - \frac{2rp}{r + p} \quad (4)$$

Then, F-measure is defined as $1 - E$. It denotes area of $A \cap B$ in Figure 8.

$$F = \frac{2rp}{r + p} \quad (5)$$

Figure 9 shows the precision-recall for six months. It does not include the on-request scheme because precision is undefined; nothing is propagated in advance for the on-request scheme.

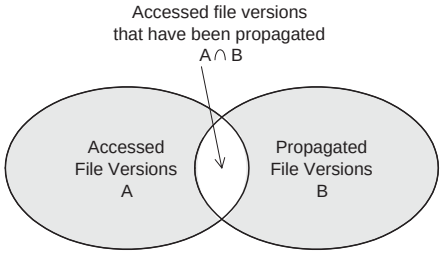



Figure 8: Precision-Recall Diagram

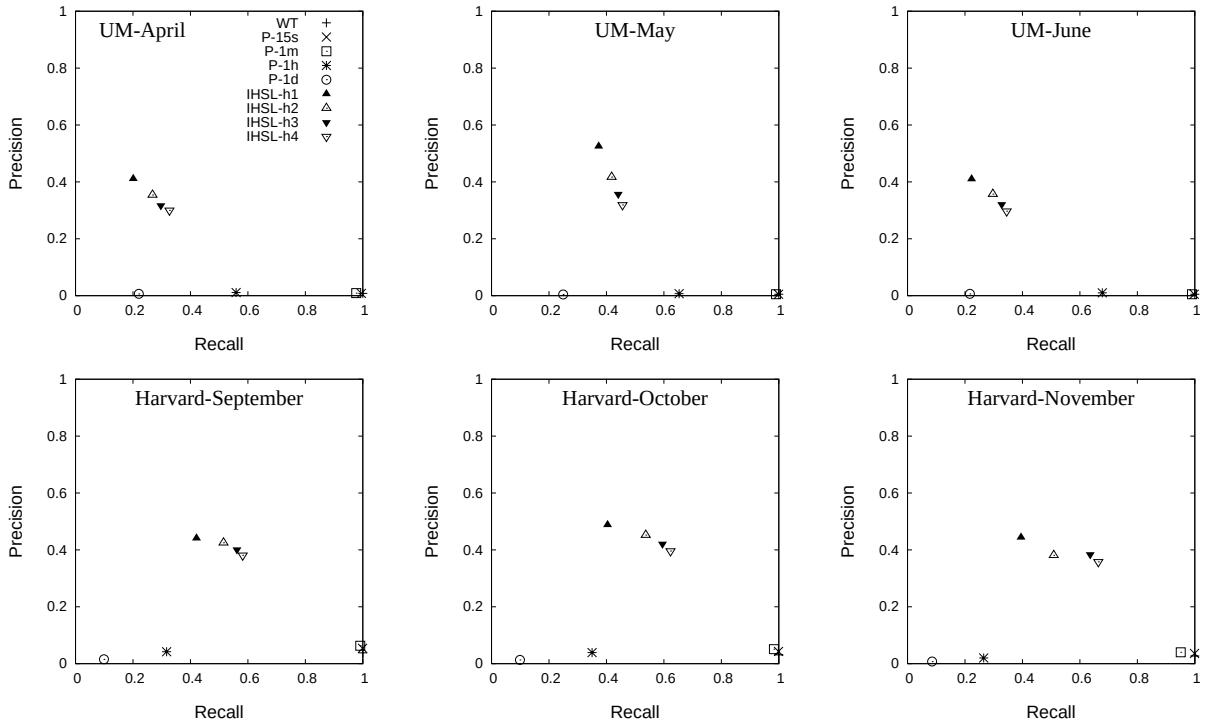
As expected, the write-through scheme has recall equal to one, but precision close to zero. For the periodic scheme, as the period increases from 15 seconds to 1 day, recall decreases; precision is very low for all periods. The periodic scheme with 15-second period depicts results similar to the write-through scheme. The periodic scheme with 1-day period has both low precision and low recall, meaning it propagates many files without reducing the number of backfetches compared to the on-request scheme.

For the IHSL scheme, we tried history values ranging from one to four. Increasing history improves recall but reduces precision; it is more liberal in deciding which files to ship. IHSL provides better overall performance by detecting when sharing no longer holds.

Figure 10 shows the F-measure. For all months, the relative performance of propagation schemes remains about the same. Different history values in the IHSL scheme did not make much difference in performance, but histories of two and three worked slightly better than one and four. In short, our experiments show that the IHSL scheme with history value of two or three provides the best balance between recall and precision.

## 6 Related Work

All of our data-propagation schemes depend on predicting future file access patterns. This is similar to prefetching algorithms developed to reduce file access latencies. But, most previous work tries to predict file accesses from a single node while our work focuses on predicting future sharing of files between different nodes. Griffioen [5] developed a prefetching algorithm based on a probability graph where each node represents a file and a directed arc represents access order. Each arc is weighted by the number of times that the target is accessed after the source. Lei [13] used *access trees* to capture the re-




This figure shows the precision-recall graphs. For the periodic schemes, as the period increases recall decreases; precision is very low for all periods. For the IHSL scheme, increasing history improves recall but reduces precision.

Figure 9: Precision and Recall

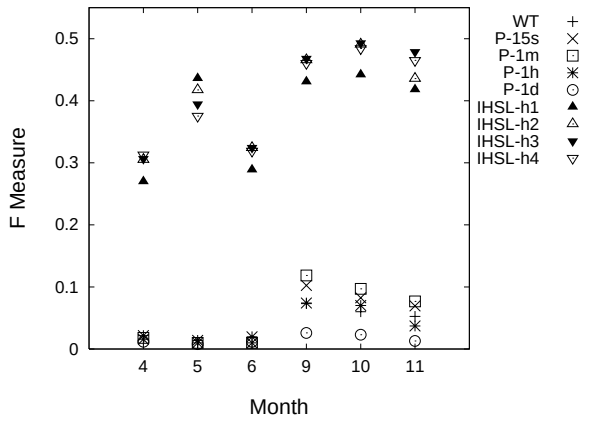



Figure 10: F-Measure

lationships among files. Access trees are maintained for each program; when a program is re-executed, the current access tree is compared against the saved access trees. Kroeger [10] suggested a last-successor model, which predicts that each file access will be followed by the same file that followed the last time. Kuenning [11] used *semantic distance* between files to choose which files should be hoarded; hoarded files are used later when mobile users are disconnected.

Distributed databases also face the data migration problem, but have very different access patterns—a lower degree of locality but a higher incidence of sharing. Mariposa applies an economic model for data migration [19]. Each query has a budget allocated to it, and it tries to minimize expenditures. Each replica site can either process a query or ship tuples; both actions accrue revenue but cost resources. By attempting to maximize revenue given fixed resources, Mariposa allocates data efficiently.

# 7 Conclusion

Wide-area file systems benefit greatly from optimistic concurrency. The low incidence of sharing allows such systems to simplify their consistency mechanisms, with substantial performance benefits. However, one must then decide when to propagate updated data from one replica

site to others. Simple schemes either ship many unneeded files, or suffer from long backfetch delays, or both.

In this paper, we have presented the IHSL scheme to intelligently decide whether or not to propagate an update. It depends on the past predicting the future. It ships updates whenever they cause invalidations elsewhere. This simple heuristic ships two orders of magnitude less data than aggressive schemes, while reducing the penalty of on-demand shipment by nearly a factor of two.

# References


[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[2] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the Winter 1992 USENIX Technical Conference*, pages 333–343, Berkeley, CA, January 1992.

[3] J. Bolot. Characterizing end-to-end packet delay and loss behavior in the Internet. *Journal of High Speed Networks*, 2(3):305–23, 1993.

[4] D. Ellard, J. Ledlie, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings fo the Second Annual USENIX File and Storage Technologies Conference*, pages 203–216, San Francisco, CA, March 2003.

[5] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *Parallel and Distributed Computing Systems*, pages 165–170, September 1995.

[6] J. S. Heidemann, J. T. W. Page, R. G. Guy, and G. J. Popek. Primarily disconnected operation: experience with Ficus. In *Proceedings of the Second Workshop on the Management of Replicated Data*, November 1992.

[7] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayana, R. Sidebothamn, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feburary 1988.

[8] M. Kim, L. P. Cox, and B. D. Noble. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*, Monterey, CA, January 2002.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, Feburary 1992.

[10] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. In *Workshop on Hot Topics in Operating Systems*, pages 14–19, 1999.

[11] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 264–275, Saint-Malo, France, October 1997.

[12] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Technical Conference*, pages 95–106, New Orleans, LA, January 1995.

[13] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of USENIX Annual Technical Conference*, pages 275–288, Anaheim, CA, January 1997.

[14] S. H. Low, F. Paganini, J. Wang, S. Adlakha, and J. C. Doyle. Dynamics of TCP/RED and a scalable control. In *Proceedings of IEEE INFOCOM'02*, pages 239–248, New York, NY, June 2002.

[15] B. D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *Proceedings of 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 138–149, Nashville, TN, May 1994.

[16] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proceedings of ACM SIGCOMM '97*, pages 51–61, Cannes, France, September 1997.

[17] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Software — Practice and Experience*, 28(2):155–180, February 1998.

[18] C. V. Rijsbergen. *Information Retrieval*. Butterworth, London, 1979.

[19] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in Mariposa. In *Proceedings of*

*3rd International Conference on Parallel and Distributed Information Systems*, pages 58–67, Austin, TX, September 1994.

[20] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island Resort, SC, December 1999.