Adaptive Consistency - Patterns of Sharing in a Networked World

Susan Spence
Erik Riedel
Magnus Karlsson

Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
{suspence,riedel,karlsson}@hpl.hp.com

Cover Page

Contact author: Susan Spence
email: suspence@hpl.hpl.com
address: Hewlett Packard
1501 Page Mill Road
Palo Alto, CA 94304
USA
tel: +1 650 857 5705

Reference Number: 589

topic area: Enterprise Data Management
category: research

Adaptive Consistency - Patterns of Sharing in a Networked World

Susan Spence Erik Riedel Magnus Karlsson
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
{suspence,riedel,karlsson}@hpl.hp.com

Abstract

The core function of any information system is access to information - i.e. reading and writing of stored data. Today, the sharing pattern for most data on the WWW is quite straightforward - creation and updating is done in a few locations, and the vast majority of users have read-only access. Users are quite tolerant of inconsistencies, both transient and longer-lasting. The difficulty in storage system design at Internet scale comes when sharing is richer and more demanding than this. Massive replication and caching, as done for popular web sites, is not appropriate when the data is accessed by only a small number of users and simple snapshot consistency is not sufficient when data is updated by multiple users in different locations. This study reports on the more demanding sharing behavior, as found, more typically, on a workgroup UNIX system, and extends it to the scale of a WWW workload. We find that the conventional wisdom on filesystems holds - very little data is shared, and most of that is shared in a read-only fashion. Our analysis focusses on the remaining behavior - those accesses to data that do involve write sharing - and classifies the behavior into several categories. We then propose a design for a "sharing cache" that predicts the sharing semantics of data, and manages consistency appropriately. We demonstrate that these predictions can take place automatically, by monitoring the behavior of applications at a low level, without intimate knowledge of what the applications are doing.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and the notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.
Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

1 Introduction

As users and companies depend increasingly on shared, networked information services, and as companies and customers become more international, computer systems need to keep pace. The core function of any information system is access to information - i.e. the sharing of stored data. This is one of the drivers that has made the WWW so wildly popular; the ability by a large audience to share data across organizational boundaries, geography, and time. A huge amount of data is now available for read-only access by a wide audience. The potential exists for sharing and collaboration at increasingly global scales, with multiple users updating the same data. This makes the ability to handle concurrent updates critical. In the future, we envisage enterprise information systems with the access characteristics of a filesystem, but at the scale of the WWW.

Past research work in characterizing file system workloads has focussed on the benefits of caching; first read caching [Baker91, Ousterhout85] and then write caching [Ruemmler93, Roselli00]. The use of large local caches to handle read locality has a first-order benefit in performance. Once re-reads are taken care of, write-behinds provide the next level of benefit. In processor cache design, both of these functions have been there from the start and, once these initial, single processor, functions are satisfied, the challenge for cache behavior becomes coherence across multiple processors[Hennessy96]. The same is true for storage access. With read and write caching taken care of, the next problem is one of *sharing* caches, since it interferes with both read caching for re-reads and write-back caching. The challenge is to ensure coherence across multiple machines in a cluster of nodes that share a single view of storage. Being able to identify and manage data sharing is important when users are in a relatively fast local network where minimizing remote access is preferable, but it is more pressing when users are distributed across more global-scale systems, where speed of light limitations will always place constraints on minimizing latency.

Past work has shown that sharing - particularly write sharing - among multiple users in a file system is rare and the results in the following sections will re-emphasize this point. Taking this as a given, the key challenge for a system that supports the sharing of data is to handle exactly those cases when there *is* sharing. This paper examines a detailed

trace of a local file system and considers what would have happened if the system had been distributed. We examine all occurences of sharing in detail, and attempt to characterize the application and user behaviors that lead to sharing - the *sharing patterns*. We also outline the design of a *sharing cache* that uses the knowledge of these sharing patterns and allows a storage system to automatically make decisions about how to manage consistency based on observed application behavior.

Section 2 outlines existing solutions to shared storage, and sketches the design of a *sharing cache*. Section 3 provides an overview of our trace, and discusses the overall sharing behavior we see. Section 4 details specific patterns of sharing. Section 5 examines variations and limitations of the analysis when accesses are known only at the block level. Section 6 considers how our analysis might scale to the much larger numbers of files and users in a web server workload. Section 7 revisits the sharing cache. Section 8 discusses related work. Finally, Section 9 concludes and outlines future work.

2 Alternatives for shared storage

There are a wide range of options for ensuring consistency of data directly in applications - from building on top of a database management system with strong consistency guarantees to numerous forms of middleware including object-oriented databases, distributed object systems, or direct implementation of communication between distributed sites. Use of such mechanisms allows an individual application to explicitly choose the semantics and performance requirements that are most appropriate for the behavior its users expect. The difficulty is that each individual application must be created or modified to use these mechanisms for distributed data management, which can require large amounts of coding and often wholesale (re-)design of software. This becomes a barrier to the deployment of highly-shared systems.

If a common, low-level interface for accessing and sharing data were available, then it could serve as the basis for consistency management, as long as it "learned" enough about the varying semantic requirements. Four scenarios below describe ways that data may be shared using a file system interface, some of which are illustrated in Figure 1.

2.1 Local file system

In most systems today, there is already a common interface to storage - applications are written against a file open, close, read, write interface at the file system level, which then translates into a set of lower-level block requests to a storage device driver. This commonality of interface makes it an attractive proposition to provide a shared storage service at one of these levels.

2.2 Shared SAN file system

In the case of a local area network, a shared SAN file system [SNIA01, IBM02] allows a number of hosts to share the same physical storage resources. As this sharing becomes more logical - i.e. where *data* can actually be shared, instead of simply raw storage capacity - the need for explicit management of data consistency becomes important [Amiri00, Burns00]. The latencies are not very large in such systems - and a number of optimizations can take advantage of this fact - but data sharing is a basic design concern. For access over greater distances, two different approaches exist today.

At the block level, remote mirroring across disk arrays connected by a wide-area link [EMC00] provides replication of all write operations by applications at a primary node to a second backup node, but only one node is active at any given time. These systems can operate synchronously (ensuring complete consistency of the backup), asynchronously (delaying some operations), and variants of batch windows (sending periodic updates), however these

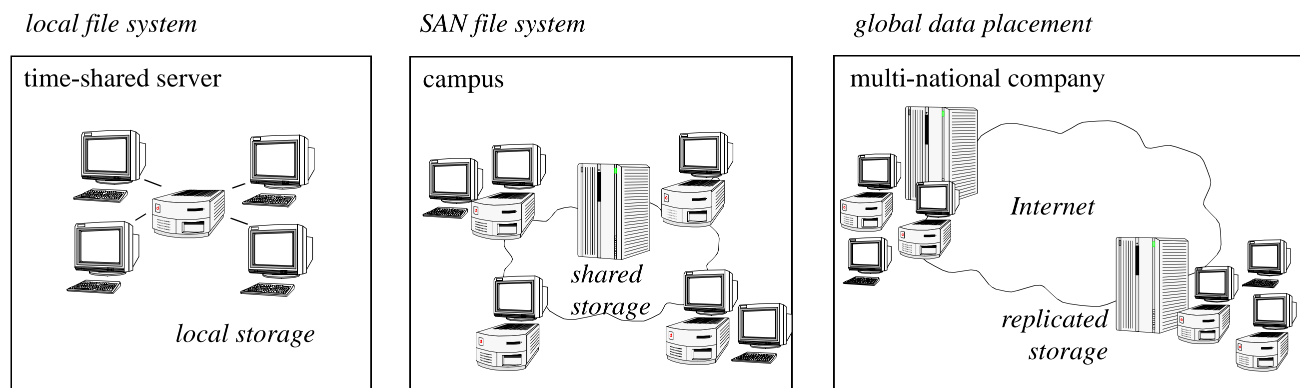


Figure 1. Shared storage systems. Three alternatives for shared storage today - a single server with directly-attached storage and multiple users, a number of servers sharing SAN-attached storage, and multiple distributed storage systems with global latencies and multiple replicas.

choices must usually be made all-or-nothing for the system, and do not differentiate between different classes of access.

A second option, with higher-level access, is a distributed file system - this layer has more information about access patterns and types of access, but still provides a limited set of interfaces for specifying different consistency semantics [Kistler92, Peterson97, Thekkath97, Ji00]. It must, by definition, be one-size-fits-all.

2.3 Web servers & content delivery networks

When sharing is largely read-only, with data distributed from a central location [Akamai02], consistency management is relatively straightforward. It becomes simply a trade-off between local performance and additional server resources. Replicas can be created aggressively, and more replicas mean better local performance. Consistency is rarely an issue as updates are few, relative to reads, and users tolerate inconsistency.

2.4 Shared global file system

Consistency becomes more of an issue as the scope of data being distributed is reduced, for example within a single corporation using an enterprise network [Liste01, Ricotta01]. As the scope narrows, the variety of access patterns increases.

Consider the increasingly common scenario of a multinational company. With offices around the world, the company needs to access its data using a variety of applications efficiently and reliably across these globally-distributed sites. Each location manages a local data center that provides data storage and processors to run applications. The centers might be connected with dedicated links, or may use connections over the commodity Internet. What are the implications of supporting global use of a range of applications which were originally intended for use only in a localized setting?

We believe that a successful system for global data management should operate at the lowest level possible in order to provide the maximum compatibility with existing applications, while at the same time taking into account as much higher-level knowledge as possible. The system should not expose additional semantics, but simply observe the behavior of applications and adapt the consistency and distribution schemes as appropriate for each subset of data and application behavior.

2.5 A sharing cache

We would like to be able to use our analysis of a filesystem trace to inform our design of a global-scale architecture for sharing data with similar access characteristics. Our final goal is to provide a general data sharing service - which we call *global data placement* - that is applicable to a range of applications that use a simple filesystem interface today. We envisage a potentially globally-distributed system, where users view their data as being stored in one very large, logical store. Users should be able to access the same data from multiple sites. We expect that the majority of data in the logical store will be unshared, while a small proportion will be shared by multiple users. The conventional wisdom on filesystems and our own numbers will bear this out.

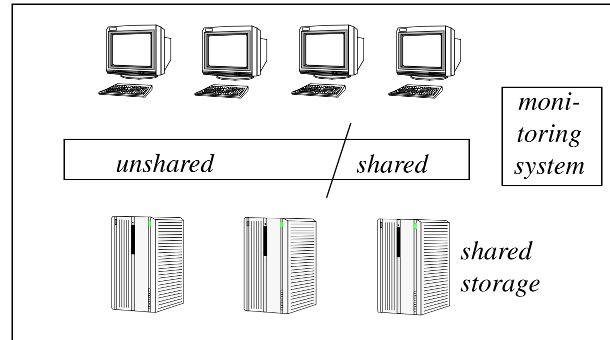


Figure 2. Sharing cache. A logical entity between hosts and storage; its management of shared and unshared data is informed by the monitoring system.

The logical store will be supported by a federation of networked storage devices. User accesses are made to a copy of the data that is created and cached locally for the user. During system operation there is a collection of recently accessed data resident in individual user caches, but only a small proportion of this data is shared and requires explicit consistency management: we view this collection of shared data logically as the “sharing cache”, as illustrated in Figure 2.

In order to identify the appropriate level at which to provide the shared abstraction - whether at the file-level or block-level interface - we must trade off the simplicity of the low-level management against the greater knowledge about application behavior available at the higher level. Our analysis looks at the information available to us in both file level and block level traces, identifies the tradeoffs between implementation at the level of the buffer cache in hosts and within arrays, and uses this to inform the design of the sharing cache.

Our aim is for the system to predict whether data will be shared or unshared, and to tailor management of that data accordingly. Predictions about sharing are based on a history of interactions between application processes and a library of knowledge about the consistency requirements of those applications. When different users share data, the multiple physical copies of the same data distributed to each user’s host should be maintained with sufficient consistency to ensure that the users still see one logical store. The degree of consistency actually required depends on the applications being used.

The goal of our study is to determine whether an automated system can do prediction of sharing with sufficient accuracy to support a usable, globally-distributed storage system. Based on the results of our analysis, we look at the design of the sharing cache in more detail in Section 7.

3 Workload analysis

Although the premise for our analysis is that a trace of the activity of a group of users on a time-sharing server with a local storage system is representative of the type of usage and sharing amongst a similar group of campus-wide or globally distributed users, this is probably a pessimistic assumption. Users of a more widely distributed system, with inherently higher latencies, are likely to share data less aggressively than in the faster, local case. Nevertheless, considering a locally-shared workload in the context of a widely-distributed system tests the "worst case" scenario. This section presents details of the original trace and how we interpret it to simulate a more distributed context.

The basis for our evaluation is a 10-day trace of all file system accesses done by a medium-sized workgroup of researchers, using a 4-way HP-UX time-sharing server attached to several disk arrays and a total of 500 GB of storage space. A range of applications are run on the server, including compilations, simulations, email, web browsing and source code control. Table 1 provides an overview of this trace.

3.1 Trace collection

The trace of accesses to the filesystem was taken at two different levels. The higher-level trace captures the file requests, such as open, read, write and stat, that are generated by applications and seen by the filesystem. This trace was collected by instrumenting the kernel to log all file system calls at the syscall interface, i.e. above the buffer cache.

	12-hour	10-day
hours	12	240
pre-cache requests	5,796,200	86,385,100
post-cache requests	1,370,270	20,623,300
data moved	23 GB	129 GB
active users	23	32
user accounts	207	207
active files	111,000	969,000
total files	4.0 million	4.0 million
file systems	24	24

Table 1. *Overview of file system trace.* The 10-day trace covers a period in late 2000 from a Thursday to the following Saturday. The 12-hour subset covers 8am to 8pm on the first day.

The HP-UX filesystem translates these operations into block-level operations, such as R data, W inode, R indirect and W data, and passes them down to the buffer cache. If the data to service an operation is already resident in the buffer cache, the operation will be serviced there. However, an operation on data that is not already resident in the buffer cache, or needs to be synchronously written, results in a cache-miss. The lower-level trace captures only those block-level operations that require data to be retrieved from the storage system by instrumenting within the storage device driver.

We refer to the higher-level trace as *pre-cache* and the lower-level trace as *post-cache*. Note that between these two traces, there are actually two mappings being made, which complicates our analysis. First, the interface changes from file to block accesses, but we do not have sufficient information to map files to the block they actually use on disk. Second, the block-level trace is post-cache, so it hides any accesses that hit in the buffer cache.

In order to compare the traces, most of our analysis focuses only on accesses to inodes, which represents the translation between files and disk blocks. This means that we do not have exact information about the size of the data being accessed, so our analyses report cache sizes as number of files. With a more detailed block-level trace taken above the buffer cache (not available to the authors), additional study could be done using the explicit mapping of filenames and inode numbers to the actual data blocks of the file.

The disparities between the two traces make our analysis somewhat more difficult. Section 5 discusses how these two views might affect our results, and how it would impact a final design.

3.2 Trace analysis

The trace upon which we have actually performed our analyses is a subset of the original trace. We have preprocessed both the pre-cache and post-cache traces to remove records that are irrelevant or misleading. Our focus is on processes run by individual users, rather than on system processes, since we are interested in analysing sharing between users. Thus, from both traces, we have removed all requests issued by backup processes, the netnews server, the ClearCase vobadm, and the processes collecting the traces themselves. While the behavior of these various system-level applications is interesting, we wish to focus on "real" end-user application behavior.

From the pre-cache traces we have also removed all requests to unknown filesystems, those with zero-length filenames or with otherwise invalid filedescriptors.

3.3 Where is the data stored?

The original trace has all users located at one central site. We initially considered an interpretation of the trace with users randomly distributed across three global sites, e.g. modelling a company with three design centers around the world. However, we subsequently decided to increase the number of sites and maximise sharing between sites by considering each user to be assigned to a separate site, giving us up to 32 sites involved. Using three sites just for illustration in Figure 3, if each circle represents the footprint of data accessed by one user, there is likely to be a large proportion of the data which is only used by that individual user, a smaller proportion of the data, as indicated by the overlaps, that is shared with each other site and an even smaller proportion of the data that is shared by multiple sites. Our concern is with the shared data represented by those overlaps.

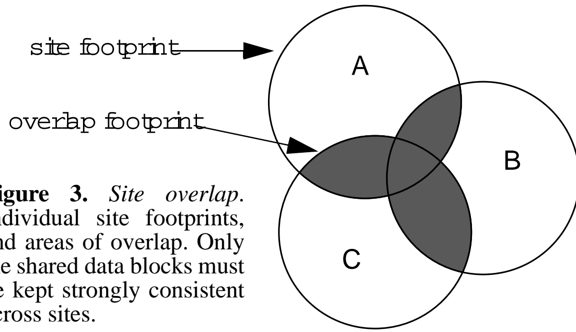


Figure 3. *Site overlap.* Individual site footprints, and areas of overlap. Only the shared data blocks must be kept strongly consistent across sites.

3.4 How much data is shared?

To confirm whether our assumptions on the proportions of shared data are correct, we have performed a number of analyses on data sharing between users. We initially focus on the size of the data footprints involved. Our traces do not tell us how large the used files are on disk. However, from the pre-cache trace we can determine how many files are used at each site and how many files are shared between sites, as illustrated in Figure 5 and Figure 6. To give some intuition about the number of files in those overlaps between two or more users, Figure 4 graphs the number of files that are shared against the number of users that share them. Over the 12 hour period this graph represents, 65 files are shared by only two users, 4 files are shared by four users, 2 files are shared by five users and in all other cases, only one file is being shared.

Note that the total number of files accessed in the trace during this time is 111,000, so only a tiny fraction of the files are shared at all.

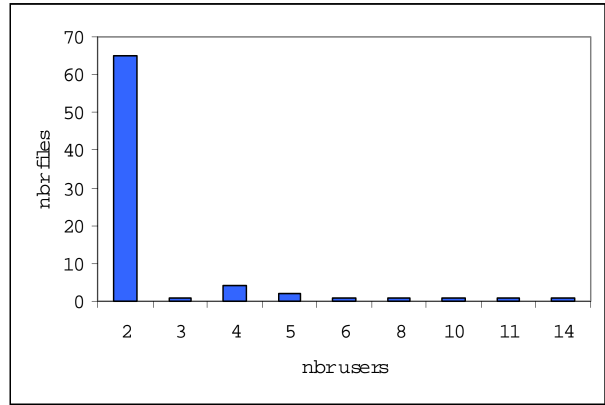


Figure 4. *Files shared.* The number of files shared by n users, where file sharing occurs over 12 hours.

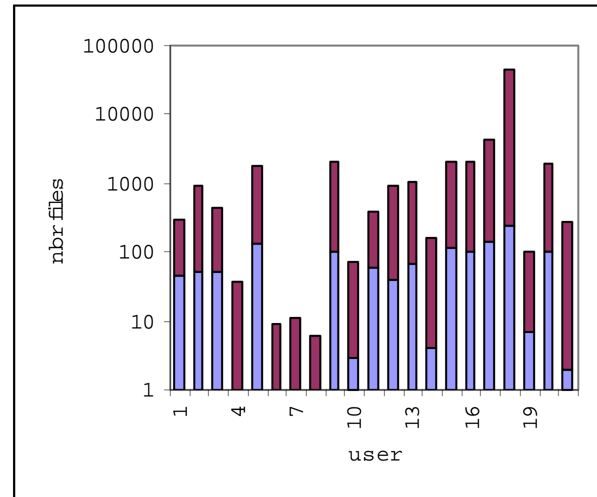


Figure 5. *Data footprint for 12 hours.* Number of files accessed and shared by each user.

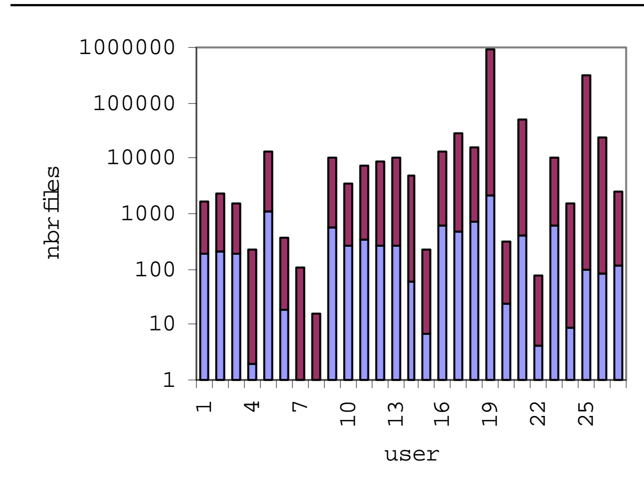


Figure 6. *Data footprint over 10 days.* Number of files accessed and shared by each user.

3.5 How often is data shared?

We can determine more about the sharing behaviour of users at different sites by studying how often they share data and the amount of time between accesses to the same file by different users. This is quantified by our *inter-reference* analysis. An inter-reference is defined as a write access made by one user on a file, followed by a read or write access on the same file by a different user.

Table 3 shows the number of inter-references that occur in the trace, compared with the total number of requests overall. The number of inter-references illustrates how often accesses are made by different users to the same data; the data in the overlap footprints as introduced in the previous section.

	12-hour	*10-day*
inter-references	3,763	40,648
requests	5,796,200	86,385,100
active files	111,000	969,000
total files	4.0 million	4.0 million

Table 2. *Basic interreference information*

3.6 Stability of sharing behaviour

The footprints of shared data remain stable over days, as shown in Table 3. Of the files that are found to be shared, 149 files are shared only within a single day, while only five files are shared on all ten days. Most importantly for our analysis:

- the total number shared files is low (194) and
- the vast majority of inter-refs are to the busiest files.

The stability of the shared footprints is important because it gives us an idea how well an automated system might be able to predict which data is subject to sharing and consistency requirements.

4 Sharing patterns

The next stage of our analysis focusses on sharing in more detail. Looking at the inter-reference times, we narrow our focus to the most critical inter-reference behavior - those

# of days	*filecount*	*interrefs*
1	149	226
2	14	99
3	5	31
4	5	74
5	1	18
6	4	144
7	4	364
8	3	585
9	4	432
10	5	2228
total	194	4201

Table 3. Stability of file sharing over 10 days.

with the *shortest time* between a write at one site and a subsequent access at a second site. We then consider the detailed sharing behavior among the interacting applications. Finally, we categorize all the sharing behaviors into a small set of common consistency types that determine how access to the shared data should be handled.

4.1 How much time to maintain consistency?

Focussing on one hour of the trace with typical activity (10-11am on the first day), we chart a cumulative distribution of all the inter-references that occur in that hour. As shown in the first graph of Figure 7, this illustrates, for every inter-reference, the time taken between the first write and the first subsequent read or write to the same file. Focussing on those critical inter-references that occur within two seconds, as illustrated in the second graph of Figure 7, we see that only 8-10% of the interreferences occur within this time.

4.2 False sharing

While we may observe the same inode being used repeatedly by multiple processes in the course of our sharing analysis, this does not necessarily indicate that any data sharing is actually taking place. The same inode may be freed when no longer in use for one file, and subsequently

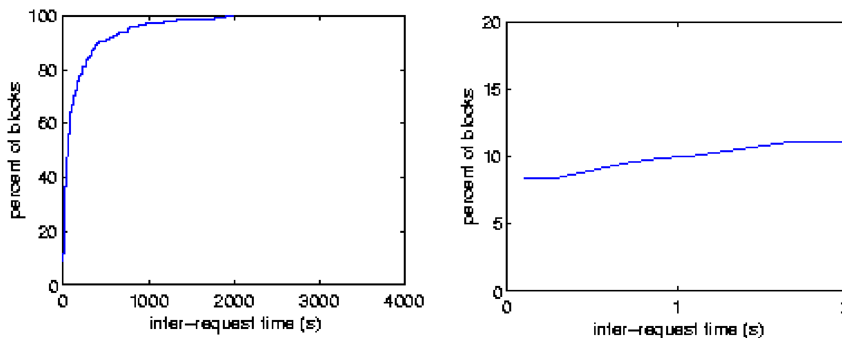


Figure 7. *Inter-reference times*. Time between references to shared files, as a percentage of all the shared file accesses.

reused for another file. False sharing is defined as different users using the same inode for different files. Being able to distinguish false sharing from real sharing enables us to gain more useful information about the data being shared and may enable us to avoid maintaining consistency unnecessarily.

Table 4 shows how, based on our analysis of all requests, we can identify the inter-references, focus on those that occur within a period of time that is likely to cause consistency problems at a global scale (120 seconds) and distinguish between the real and false sharing in those significant inter-references. The amount of real sharing, in comparison to the total number of requests issued, is a small fraction.

time	*requests*	*interrefs*	*<120s*	*false*	*real*
10day	86,385,100	40648	14611	6140	4736
12hr	5,796,200	3763	2518	1245	604
1hr	827,894	556	384	188	99

Table 4. Inter-reference and file-sharing information over 10 days

Figure 8 illustrates twelve hours of false and real sharing, as proportions of all inter-references under 120 seconds in each hour. (Those labelled unmatched are the inter-references for which we were unable to determine the name of the file involved from our traces.) The amount of real sharing is a small proportion of the significant inter-references observed in each hour.

4.3 Application sharing behavior

We can identify when sharing takes place at the low level of the post-cache trace too. As illustrated in Figure 9, we can observe the writes issued by different processes as both file-level requests and block-level requests. We can also observe when those processes share the same data in close succession. What is actually happening is that the rmail process illustrated on the left issues writes to append a message to a user's mail spool file. Within 30 seconds, the zmail process reads the same mail spool file, detects the update and makes a new internal copy of the file. What we cannot determine, just from observing the requests, is whether the second process accessing the file must see a consistent copy.

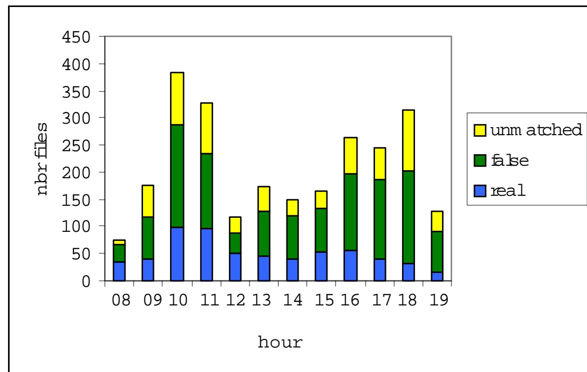


Figure 8. *False sharing*. The amount of real sharing, false sharing, and un-identified records in 12 hours of the trace.

4.4 Sharing behavior - all applications

To begin to quantify the different types of consistency mechanisms required, we have created a categorization of inter-references by the applications involved in them and a categorization of the type of consistency required by those applications.

The consistency types include:

- *publish* - a write issued by one application is followed by subsequent accesses by other applications, but ordering of these requests is not important.
- *append* - a write is append-only, and its ordering w.r.t. subsequent writes is not important.
- *database* - writes by different applications are interleaved and their ordering must be maintained
- *lock file* - consistency is managed through a lock file, the lock file itself sees contention, but the underlying data does not.

The consistency types are used to classify a range of application behaviors as shown in Table 5. All of the critical inter-references are considered over 12 hours and all 10 days of the trace, i.e. those with real file sharing that occur in less than 120 seconds. We are able to classify over 97% of the inter-references successfully.

4.5 Prediction in the sharing cache

Given the complete past history of a system, as captured in the filesystem trace, we have been able to identify sharing and even to classify the sharing we have observed into different consistency types. However, if we do wish to design a system that relies on prediction to handle sharing correctly, when we do not yet know when sharing will occur, we need appropriate criteria to classify *all updates* with consistency types, even where one of those types is "*no consistency required*". Based on a library of application knowledge and on what we have "seen before", we must make assumptions on every "first write". Table 6 illustrates how different assumptions change the number of inter-references left unclassified, depending on the criteria used. It is clear that making classifications based on one or even both applications alone is not sufficient: too many inter-references are left unclassified in these cases. Classification is far more successful if filenames are also used. Many applications do use standard filenaming conventions and, in

application	12hour	10 days	consistency type
utmp	41	242	database
wtmp	7	61	append
devlog	75	445	none
devnull	140	846	publish
history	5	12	append
httpd	1	53	publish
smbd	0	9	publish
superlock	19	354	lockfile
maildelivery	2	20	append
maildeliveryread	127	1017	publish
maildeliveryedit	2	42	publish
mailreadsort	4	4	publish
mailreadedit	18	22	publish
mailreadsend	0	0	publish
maildeliverysend	0	0	publish
mailsend	0	0	publish
mailread	3	26	publish
mailftp	0	0	publish
mailnetd	2	15	publish
encodeftp	4	4	publish
shellftp	0	0	publish
mailtwm	91	479	publish
maildeliverywww	29	259	append
mailreadwww	1	21	publish
mailsendwww	0	0	publish
editwww	0	0	publish
printing	27	150	publish
unclassified	6	120	-
total	604	4201	-

Table 5. *Detailed inter-reference types*. A more detailed count of all the inter-reference events in the trace, broken down by application function and consistency type. Measured across 12 hours of the first day and all ten days of the trace.

day	filename only	1st app only	both apps	both apps +filename
1	23	327	278	47
2	15	262	292	58
3	1	30	29	5
4	3	19	34	11
5	44	191	148	36
6	29	250	267	77
7	10	284	276	54
8	24	161	194	68
9	15	251	211	27
10	3	52	22	3

Table 6. *Prediction of consistency types*. What info is used to predict an access pattern. Lower numbers are better.

combination with application information, classification is more successful.

4.6 Prediction using first writes

To implement a system relying on prediction of sharing, we need criteria to decide what we will attempt to keep consistent. The classification of inter-references we have observed in our trace provides us with knowledge about files that are typically shared and applications that typically interact to share data. Using this knowledge of past accesses, we can make predictions about what sharing we will see in the future, based on what we can determine about the “first write” that we observe in a potential inter-reference. As Table 7 demonstrates, we are able to make the best predictions based on a combination of the filename and the application involved in a “first write”.

Let us consider all the writes that occur in one hour of the trace to illustrate this point. There are 270,000 write requests made during the hour. Making predictions of sharing, based on filename and application involved in the

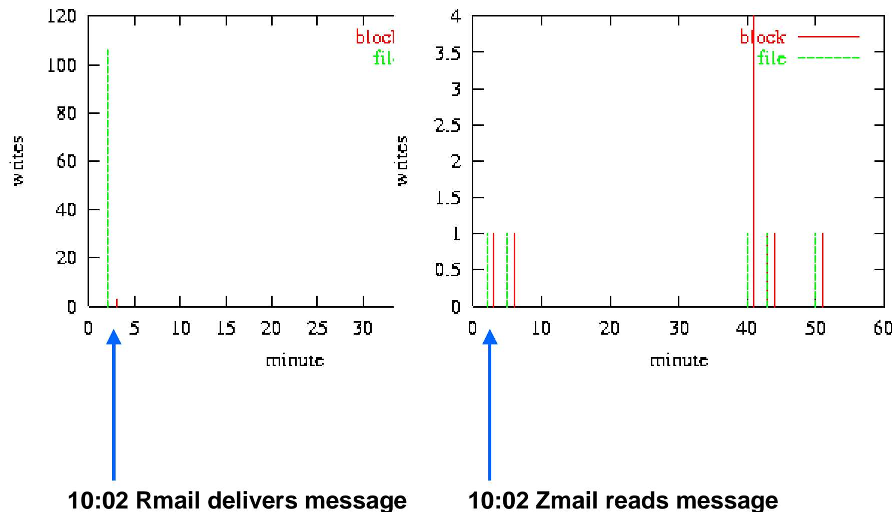
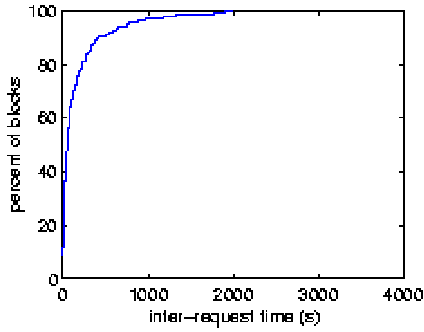


Figure 9. *Sharing example - combined apps.* The detailed behavior of an rmail and zmail program interacting on shared data.



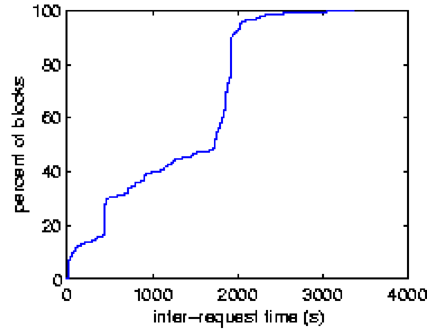


Figure 10. *Inter-reference timing*. Details of pre-cache and post-cache block inter-references.

Pre-cache
Total requests: 827,894
Nbr inter-refs: 556

Post-cache
Total requests: 371656
Nbr inter-refs: 14304

	1 hour	12 hours
write requests	269,387	1,252,330
unshared	243,677	805,584
potential sharing	25,710	446,743
actual inter-refs	556	3,763

Table 7. Predictions based on first write for one hour and for 12 hours of the trace.

"first write", we identify 26,000 write requests with a potential for sharing. This implies that we should apply appropriate consistency management to 10% of the write requests issued in that hour and assume that the remaining 90% are unshared. In fact, based on what we actually know about inter-references in that hour, it turns out that only 556 of those write requests are actually involved in inter-references. The same analysis over 12 hours indicates that we should apply appropriate consistency maintenance to 35% of the writes. The difference between the actual number of inter-references and the number of writes for which we have predicted sharing is the price paid for having to be conservative in our predictions, to ensure that we do manage the actual sharing successfully. The cost of maintaining consistency in this system is still significantly less than for one that tries to maintain consistency on every update.

5 Details - files versus blocks

A significant difference between doing our sharing analysis with the pre-cache trace and with the post-cache trace, is that we can view sharing of files in the pre-cache trace, while we see sharing of blocks in the post-cache trace. This affects our interpretation of the sharing observed at each level, our ability to do comparisons between the traces and the amount of information we can deduce about sharing at each level. In this section, we look in more detail at the benefits and drawbacks of the respective interfaces.

5.1 Effect of caching

As discussed in our introduction to the precache and post-cache traces in Section 3.2, some requests are satisfied by the buffer cache, while others require data to be brought in from disk. To determine the effect of the buffer cache on our sharing analysis, we compare the number of inter-references observed in the pre-cache trace with those in the post-cache trace. The first graph in Figure 10 shows the pre-cache inter-references for the same hour as in Figure 7, while the second graph shows the post-cache inter-references observed for that same hour.

The number of requests observed in the pre-cache trace should be interpreted as the number of requests made on inodes; the number of inter-references as the number of shared inodes.

The number of requests observed in the post-cache trace should be interpreted as the number of requests made on blocks; the number of inter-references as the number of shared blocks.

Obviously, it is difficult to make a direct comparison between the two traces here. The number of file requests translates into a much smaller number of block requests post-cache. However, because there may be multiple blocks shared within the one file, there are more inter-references on blocks than there are on files. However, it is encouraging, when considering the time available for maintaining consistency, that there are few really critical inter-references at either level. Only 62 inter-references (11% of all those in the hour) occur in less than 2 seconds in the pre-cache trace for the hour, while only 26 (0.18%) occur in less than 2 seconds in the post-cache trace.

5.2 Inodes vs. files

To take into account the effects of false sharing, we can revisit the stability of file sharing that was illustrated in Table 3. Our initial presentation of inter-references from

the pre-cache trace really considers the sharing of inodes. The difference between the number of inodes used over multiple days and the number of files used over multiple days is illustrated in Table 8. Overall, more filenames are shared than inodes, illustrating inode reuse.

	INODES		*FILENAMES*	
# of days	*count*	*interrefs*	*count*	*interrefs*
1	149	226	258	293
2	14	99	6	16
3	5	31	3	11
4	5	74	1	4
5	1	18	1	18
6	4	144	3	62
7	4	364	1	354
8	3	585	4	646
9	4	432	4	311
10	5	2228	6	2486
total	194	4201	287	4201

Table 8. Stability of file sharing, compared with inode sharing.

When interpreting sharing based upon filenames, we must take into account that the same filename with a different inode represents two different files. When interpreting sharing based upon inode information, we must take into account that the same inode can represent different files, and often does, since we have observed aggressive inode reuse in our traces.

6 Scaling up - WWW workload

The preceding analysis focussed on the traces for a range of filesystem applications for a relatively small server of 25 users. This section gives a brief impression of the implications of extending our analysis to a much larger workload - at the scale of a corporate web server with over a million users. The long-term intent, though not addressed in this paper, is to see if the sharing behavior of the file system workload can be projected onto the scale of the web server workload.

6.1 Scaling sharing

To get some feel for how we might map the large-scale sharing of the filesystem workload to the large-scale distribution of the WWW workload, we have generated tables for each workload representing the distribution of numbers of users vs. number of file accesses over the 10 days.

Table 9 shows the dominance of high numbers of accesses by small numbers of users in the filesystem workload, while Table 10 shows a much more even distribution of accesses across numbers of users for the WWW workload.

7 Sharing cache - design

Given the results of our analysis, we envisage the sharing cache working in the following way.

The system will come with an existing but extendible library of knowledge about names of files that are known to be typically shared and lists of applications that are known to do data sharing with other applications. This library will be created based on analyses similar to that done in section 4. Thus, it will cover established applications typically used in a filesystem, such as email readers, web browsers and text editors, and well-known files such as user-accessed system files.

The store will have metadata associated with each file, indicating whether the file is shared or unshared. An initial categorization of stored files, using library knowledge, can categorize only based on filenames. However, our analysis of sharing predictions based on filenames indicate that this is a reasonable start.

Our analysis of the use of filenames and “first write” applications as predictors of sharing shows encouraging results. Thus, a list of files and the applications that are predictors of sharing will be kept at each host's cache.If a file is created with a name corresponding to that of a known shared file, it will be put in the host's sharing cache. When an existing file is accessed by a user, the data will be brought from the store into the host's cache. If the data is already marked as shared, it will be brought directly into the shared cache; otherwise, into the unshared cache. If data in the unshared cache is accessed by an application that is predicted to share, the data will be moved to the sharing cache (by changing pointers as appropriate).

The sharing cache will have a range of consistency mechanisms available to it, to support the range of known consistency types. Data created or updated in the sharing cache will be propagated to disk and to any replicas, with the mechanism appropriate for its consistency type. Data

#users / #files	*small*	*medium*	*large*
small	415,016	40,495	0
medium	7,330	7,076	117
large	140	222	52

Table 9. Inter-reference and file-sharing for file system trace over 10 days

#users / #files	*small*	*medium*	*large*
small	65,135	0	0
medium	345	2,856	0
large	1	38	117

Table 10. Inter-reference and file-sharing for web trace over 10 days

updated in the unshared cache will be propagated to disk synchronously or asynchronously in the normal way.

Complimentary to the library of sharing knowledge, there will be a monitoring system which identifies as yet unrecognised sharing interactions between applications when they occur. Our analysis has demonstrated that we can identify inter-references from a historical trace. The initial update to a newly-identified shared file will be propagated in a best-effort manner. The sharing applications will be added to the library forthwith to ensure that any subsequent interactions are managed in the sharing cache.

8 Related Work

For consistency at the storage level, the bulk of previous work is in the context of distributed file systems [Kistler92, Peterson97, Thekkath97, Bolosky00, Ji00]. These use a variety of methods for achieving consistency, many of which are applicable to the applications presented here. However, they each allow only a single model of consistency, while we believe that the system should offer multiple models, and adjust to an application's requirements with minimal changes to access semantics, allowing a much wider range of applications to make use of these facilities.

Akamai has built a successful company by distributing web content using servers at strategic points in the Internet and intelligent distribution algorithms [Karger97]. These systems work well due to the relatively weak semantics and low update rates of web content, leaving room for systems that encompass a wider range of data types and semantics. We have explored exactly these richer semantics, and motivated how they will become increasingly important as larger groups of users begin to more "actively" share data.

At the block level of storage access, there may be lessons for the design of a sharing cache in work on distributed shared memory systems [Mosberger93, Amza99], as well as in mechanisms developed explicitly for storage systems such as various forms of optimistic concurrency [Adya95, Amiri00] or specialized semantics [Burns00, Yu00] applicable to a subset of workloads.

Various forms of concurrency control have been studied for many years in the context of database systems [Gray92], object-oriented databases [Butterworth91, Lamb91], and in distributed object systems [Birman93, Liskov96, Little96].

At the other extreme of complexity for programmers, the TACT middleware system requires the end application to specify it's consistency requirements along multiple dimensions of consideration. This has the benefit that the resulting semantics are *exactly* what the application requires, and therefore any optimizations are guaranteed to be correct, which can lead to a better-tuned system (less "unnecessary" consistency maintenance work). However, it has the downside of being quite invasive to program and specify. Our results have shown that by making educated "guesses" an automated system without application modification can make quite good predictions, and very infrequently make "mistakes". A future study could directly compare the overhead of an automatic sharing cache against a TACT-based application, but that is beyond the scope of this paper.

The other file system level systems for allowing weak consistency in the previous work are Coda [Kistler92] and Bayou [Peterson97]. Both of these systems have ways to fix up "mistakes". Either by using application-specific resolvers (Coda), or building in rollback semantics for each request (Bayou). The latter again requires application changes, the former requires external resolvers to be written. Coda assumes that things will go right - and then fixes the problems later on. In our sharing cache, the system attempts to predict when things might go right or wrong, and uses strong consistency when that is appropriate. This should reduce the absolute number of "mistakes" relative to an optimistic system such as Coda. Bayou prevents consistency problems with -like programming semantics, so it is not subject to "mistakes", but requires much of the programming overhead that TACT requires. TACT is a more detailed model, but may also be more intuitive and allow a greater range of optimizations than Bayou.

9 Conclusions and future work

In order to target the design of a "sharing cache" to a global enterprise information system that does not yet exist, we have analysed the relevant properties of two existing workloads. We have analysed the sharing characteristics found a filesystem trace and considered the scale of operation of a WWW workload. We can identify sharing in the filesystem trace, at both pre-cache and post-cache levels, but we have shown that real file sharing can only be identified pre-cache. Shared data is a small proportion of all accessed data, and only a small proportion of that shared data has the short inter-reference times that challenge consistency maintenance particularly at the latencies of global scale systems. A classification of observed sharing into consistency types provides the knowledge for managing sharing using prediction based on the filename and the application updating it. We have shown that using such predictions on all writes keeps the proportion of updates for which consistency much be maintained to a manageable , based on our trace.

Future work includes refinement of our classification scheme, development of our "sharing cache" design, and quantification of the impact of our approach on the performance seen by end users.

References

[Adya95] A. Adya, R. Gruber, B. Liskov and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *SIGMOD*, May 1995.

[Amiri00] K. Amiri, G. Gibson and R. Golding. Highly concurrent shared storage. *Intl. Conference on Distributed Computing Systems*, April 2000.

[Akamai02] www.akamai.com, January 2002.

[Amza99] C. Amza, A. Cox, S. Dwarkadas, L. Jin, K. Rajamani and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proc. of the IEEE* 87(3), March 1999.

[Baker91] Baker, M. et al. Measurements of a distributed file system. *SOSP*, October 1991.

[Birman93] K. Birman. The process group approach to reliable distributed computing. *CACM* 36 (12), 1993.

[Bolosky00] W. Bolosky, J. Douceur, D. Ely, M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *SIGMETICS*, June 2000.

[Burns00] R. Burns, R. Rees and D. Long. Consistency and locking for distributing updates to web servers using a file system. *Workshop on Performance and Architecture of Web Servers*, June 2000.

[Butterworth91] P. Butterworth, A. Otis and J. Stein. The Gemstone Object Database Management System. *CACM* 34 (10), 1991.

[EMC00] EMC Corporation. Symmetrix Remote Data Facility (SRDF). *Product Description Guide*, June 2000.

[Gray92] J. Gray and A. Reuter. *Transaction Processing:* Concepts and Techniques, September 1992.

[Hennessy96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1996.

[IBM02] IBM Corporation. IBM Storage Tank - A Distributed Storage System, White Paper, January 2002.

[Ji00] M. Ji, E. Felten, R.Wang and J. Singh. Archipelago: an island-based file system for highly available and scalable Internet services. *USENIX Windows Symposium*, August 2000.

[Karger97] D. Karger, E. Lehman, F. Leighton, M. Levin, D. Lewin and R. Panigraphy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web, *STOC*, May 1997.

[Kistler92] J. Kistler and M. Satyanarayanan. Disconnected operation in the Code file system. *ACM Trans. on Computer Systems* 10(1), 1992.

[Lamb91] C. Lamb, G. Landis, J. Orenstein and D. Weinreb. The ObjectStore database system. *CACM* 34 (10), 1991.

[Liskov96] B. Liskov, et al. Safe and efficient sharing of persistent objects in Thor. *SIGMOD*, June 1996.

[Liste01] M. Liste. Content Delivery Networks (CDNs) - A Reference Guide. *Cisco World* magazine, March 2001.

[Little96] M. Little and S. Shrivastava. Using application specific knowledge for configuring object replicas. *Third International Conference on Configurable Distributed Systems*, May 1996.

[Mosberger93] D. Mosberger. Memory consistency models. *Technical Report 93/11*, University of Arizona, 1993.

[Ousterhout85] Ousterhout, J.K. et al. A Trace-driven analysis of the UNIX 4.2 BSD file system, *SOSP*, December 1985.

[Peterson97] K. Petersen, M. Spreitzer, D. Terry, M. Theimer and A. Demers. Flexible update propagation for weakly consistent replication. *SOSP*, October 1997.

[Ricotta01] J. Ricotta. Managed Enterprise CDNs. *Networld + Interop*, May 2001.

[Roselli00]D. Roselli, J. Lorch and T. Anderson. A Comparison of Filesystem Workloads. *USENIX 2000.*

[Ruemmler93] C. Ruemmler and J. Wilkes. UNIX disk access patterns. *USENIX Winter 1993.*

[SNIA01] SNIA Technical Council. Shared Storage Model - A framework for describing storage architectures, June 2001.

[Thekkath97] C. Thekkath, T. Mann and E. Lee. Frangipani: a scalable distributed file system. *SOSP*, October 1997.

[Yu00] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. *OSDI,* October 2000.