Fragment Reconstruction: Providing Global Cache Coherence in a Transactional Storage System

Atul Adya Miguel Castro Barbara Liskov Umesh Maheshwari Liuba Shrira

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Cooperative caching is a promising technique to avoid the increasingly formidable disk bottleneck problem in distributed storage systems; it reduces the number of disk accesses by servicing client cache misses from the caches of other clients. However, existing cooperative caching techniques do not provide adequate support for fine-grained sharing. In this paper, we describe a new storage system architecture, *split caching*, and a new cache coherence protocol, *fragment reconstruction*, that combine cooperative caching with efficient support for fine-grained sharing and transactions. We also present the results of performance studies that show that our scheme introduces little overhead over the basic cooperative caching mechanism and provides better performance when there is fine-grained sharing.

1 Introduction

This paper describes how to integrate two techniques that have been shown to improve performance in distributed client/server storage systems: *cooperative caching* [8, 11, 12, 20] and efficient support for *fine-grained sharing* [1, 4, 6, 9, 10, 15]. Cooperative caching combines client caches so that misses at one client can be serviced from caches of other clients. It takes advantage of fast local area networks to reduce access latency, replacing slow disk accesses by significantly faster fetches from the memory of other clients, and it reduces the load on the server processors and disks, thus improving scalability.

However, existing cooperative caching techniques use pages as the unit of consistency and do not provide adequate support for fine-grained sharing, although studies in databases [4] and DSM systems [6, 15, 16] have shown that such support is important to avoid performance problems caused by false sharing. This paper presents a new storage system architecture for client/server transactional storage systems, *split caching*, and a new cache consistency protocol, *fragment reconstruction*, that together preserve the benefits of both cooperative caching and efficient support for fine-grained sharing. The new architecture is designed for an environment where clients and servers are connected by a high-speed network, and machines trust each other and are willing to cooperate, e.g., a corporate intranet.

Split caching divides functionality between clients and servers: client caches are used to avoid disk reads, while server caches are used to avoid disk writes. Disk reads are avoided by using the combined memory of the clients as a cooperative cache. Disk writes are avoided by using the entire server main memory as an object cache, called the *m-cache* [13, 19], in which the server stores new versions of recently modified objects. When a transaction commits, a client sends its modified objects to the server, which stores them in the m-cache replacing any previously cached versions of those objects. These modifications are installed into their disk pages in the background when the m-cache fills up. Since the m-cache stores modifications more compactly than a page cache, it can absorb more modifications and delay installations for a longer time, which in turn reduces the number of writes. However, to install a modification, the system first needs the containing page. The disk reads needed to obtain containing pages are called *installation reads*. Split caching fetches the containing pages from the cooperative cache thus providing the benefits of the m-cache without the cost of the installation reads.

Fragment reconstruction ensures client cache consistency. It supports fine-grained concurrency control techniques such as adaptive call-back locking [4] and optimistic control [1]; such techniques improve performance because they avoid conflicts due to false sharing. When a transaction commits, copies of its modified objects in other client caches become obsolete, causing containing pages to become *fragments* with *holes* where the obsolete objects are. We use fragment reconstruction to bring fragments up to date by filling the holes with the latest object versions stored in the m-cache. This is done lazily when the client holding a fragment needs the missing objects, or when the fragment is needed to service another client's cache miss or to install modifications on disk. Laziness reduces communication cost when pages are modified repeatedly and it is safe (i.e., causes no loss in reliability) because the m-cache is recoverable from the transaction log.

To evaluate the effectiveness of our techniques, we implemented them in Thor [17] and ran a set of experiments using the multi-user OO7 benchmark [5]. Our results show that our approach preserves the benefits of both cooperative caching and support for fine-grained sharing: it adds almost no overhead to the basic cooperative caching mechanism, and it substantially improves performance when fine-grained sharing affects the pages that clients fetch from the cooperative cache. Furthermore, the results indicate that many disk reads can be avoided by fetches from the cooperative cache, thus substantially off-loading work from the server processors and disks.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the system model; Section 4 describes split caching and fragment reconstruction. Section 5 evaluates the effectiveness of the new approach. We close with a discussion of what we have accomplished.

2 Related Work

Our proposal builds on previous work on cooperative caching and support for fine-grained sharing. Cooperative caching has been studied in several contexts: distributed virtual memory [11], file systems [8, 7, 20], and a transactional database [12]. All studies show that cooperative caching can reduce access latency and improve system scalability in workloads with coarse-grained sharing. These studies are complementary to ours; we explain how to extend the benefits of cooperative caching to workloads with fine-grained sharing. We can use the techniques developed in [8, 11, 20] to perform page replacement in the cooperative cache.

Cache coherence work in client/server databases [4, 9] has addressed the performance problems caused by false sharing. The study by Carey et al. [4] describes a concurrency control and coherence protocol that supports fine-grained sharing efficiently. A coherence protocol using a cache of recent updates similar to the m-cache is studied in [9]. However, neither of these studies integrates fine-grained sharing support with cooperative caching.

DSM is similar to cooperative caching; it allows clients to fetch data from the memory of other clients. Some DSM systems [6, 15] provide efficient support for fine-grained sharing. However, these systems do not deal with accesses to large on-disk databases; they assume infinite client caches and they do not address the problems of efficiently updating on-disk data and reducing the latency of capacity misses. Furthermore, they do not support transactions.

The work closest to ours is the log-based coherence study by Feeley et al. [10], which extends DSM to support transactional updates to a persistent store. One key difference is that they associate mutexes with segments and require segments to be large to reduce the time overhead of acquiring mutexes. This coarse-grained concurrency control can cause severe performance degradation due to lock conflicts in workloads with fine grained sharing. Our fine-grained optimistic concurrency control algorithm avoids these problems. The log-based coherence protocol in [10] ensures cache consistency by propagating fine-grained updates to all cached copies of a segment eagerly when a transaction commits, but the authors acknowledge that eager propagation does not scale to a large number of clients. In contrast, our coherence protocol uses a scalable lazy invalidation policy. Furthermore, like other DSM systems, their system assumes infinite client caches.

The study presented in [18] proposes sending pages from clients to the server at commit time to avoid installation reads. Split caching allows the server to delay installations for a longer time, thus reducing their number while still avoiding most installation reads by fetching pages from the cooperative cache.

3 Base System Architecture

Our work is done in the context of Thor, a client/server object-oriented database system [17]. This section describes the system architecture before we extended it to support split caching and fragment reconstruction.

Servers provide persistent storage for objects and clients cache copies of these objects. Applications run at the clients and interact with the system by making calls on methods of cached objects. All method calls occur within atomic transactions. Clients communicate with servers only to fetch pages or to commit a transaction.

The servers have a disk for storing persistent objects, a stable transaction log, and volatile memory. The disk is organized as a collection of pages which are the units of disk access. The stable log holds commit information and object modifications for committed transactions. The server memory consists of a page cache and the m-cache. The page cache contains pages recently read from the server disk. The m-cache holds recently modified objects that have not yet been written back to their pages on disk.

3.1 Fetches

When a client C accesses an object x that is not present in its cache, it fetches the page p containing x from p's server. At this point, the server adds an entry to its *directory*, indicating that client C is now caching p; the directory keeps track of which pages are cached by each client.

3.2 Transactions and Cache Coherence

Transactions are serialized using optimistic concurrency control [1, 14]. The client keeps track of objects that are read and modified by its transaction; it sends this information, along with new copies of modified objects, to the servers when it tries to commit the transaction. The servers determine whether the commit is possible, using a two-phase commit protocol if the transaction used objects at multiple servers. If the transaction commits, the new copies of modified objects are appended to the log and also inserted in the m-cache, but they are not immediately installed in their containing disk pages.

Since objects are not locked before being used, a transaction commit can cause caches to contain obsolete objects. Servers will abort a transaction that used obsolete objects. However, to reduce the probability of aborts, servers notify clients when their objects become obsolete by sending them *invalidation messages*; a server uses its directory plus information about the committing transaction to determine what invalidation messages to send. Invalidation messages are small because they simply identify obsolete objects. Furthermore, they are sent in the background, batched and piggybacked on other messages.

When a client receives an invalidation message, it removes obsolete objects from its cache and aborts the current transaction if it used them. The client continues to retain pages containing invalidated objects; these pages are now *fragments* with *holes* in place of the invalidated objects. Performing invalidation on an object basis means that false sharing does not cause unnecessary aborts; keeping fragments in the client cache means that false sharing does not lead to unnecessary cache misses. Invalidation messages prevent some aborts, and accelerate those that must happen — thus wasting less work and offloading detection of aborts from servers to clients.

When a transaction aborts, its client restores the cached copies of modified objects to the state they had before the transaction started; this is possible because a client makes a copy of an object the first time it is modified by a transaction. If the copy has been evicted due to cache management, the modified object becomes a hole and its page becomes a fragment.

3.3 Installation

When the m-cache fills up, the server installs some of the cached modifications in their disk pages. The server reads these pages into memory (these are *installation reads*), installs the modified objects in their pages, and writes the pages to disk. It then removes these modified objects from the m-cache, freeing up space for future transactions. If the server crashes, the m-cache is reconstructed at recovery by scanning the log.

Since installation is lazy, pages on disk can contain obsolete versions of some of their objects. Therefore before sending a page to a client in response to a fetch request, the system retrieves new versions of the page's objects from the m-cache and installs them in the page.

The m-cache architecture improves the efficiency of disk writes for fine-grained updates [13, 19]. It avoids installation reads at commit time and stores modifications in a compact form, since only the modified objects are stored. Nevertheless, installation reads can consume a significant portion of the available disk bandwidth [13, 19].

4 The Split Caching Architecture

The split caching architecture differs from what was described above in two important ways. First, it uses server memory only for the m-cache; there is no server page cache. Second, it implements cooperative caching. The cooperative cache is used both to service fetches and to avoid installation reads; it decreases fetch latency and makes the system more scalable by decreasing the load on the server disk.

Fragment reconstruction ensures cache consistency while allowing clients and servers to fetch fragments from the cooperative cache. To allow a fetch or an installation to use a copy of a page obtained from a client *C*, the system must ensure that *C*'s copy can be brought up-to-date using information in the m-cache. To determine whether information in the m-cache is sufficient for this purpose, we augment the servers' directory information to record a *status* for each cached page. A page is *complete* at client *C* if *C* has the latest versions of all its objects, i.e., the page is not a fragment; it is *reconstructible* if the m-cache contains new versions for all holes in *C*'s copy of the page (so that applying them to this copy will make it complete); otherwise, it is *unreconstructible*. Only complete and reconstructible pages are used to service fetches and avoid installation reads.

4.1 Fetches

When a client *A* requests a page *p* from the server, the server reads *p* from disk only if no client cache contains a complete or reconstructible copy of *p*. If such a client *B* exists, the server forwards the request to *B* along with any updates in the m-cache for *p* and updates the directory to record *p* as complete at *A* and *B*. (It selects a client *B* containing a complete copy of *p* if possible, because in this case no updates from the m-cache are sent to *B*.) *B* merges the updates into its fragment and sends the page to client *A*. This situation is illustrated in Figure 1. *B* includes only the latest *committed* versions of objects in the page; if its current transaction has modified the page, it uses the copies it made when the objects were first modified to restore the page to its committed state. (These copies were discussed in Section 3.2.)

If client *B* does not have page *p* or does not have the committed state for one of *p*'s objects, it informs the server that it cannot service the fetch. The server marks *p* as absent or unreconstructible for *B*, and obtains it from another client if possible. To improve performance, clients inform the server when they evict pages or copies of modified objects by piggybacking that information on messages sent to the server.

We also provide a second kind of fetch request, which

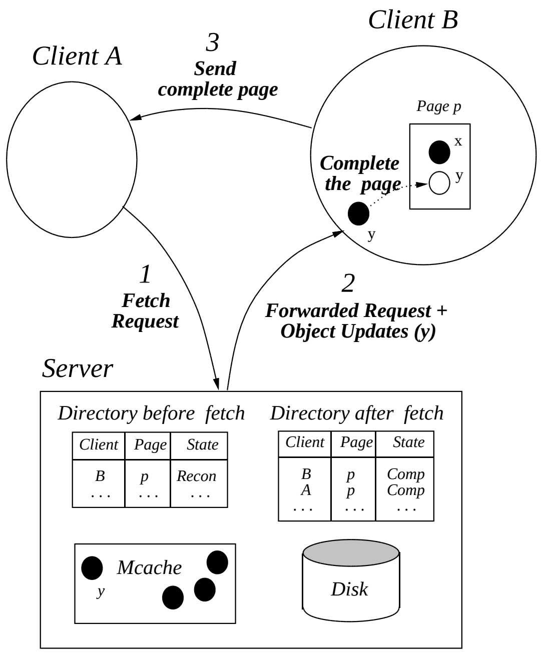


Figure 1: Fetch from cooperative cache.

indicates that the client just needs to fill in the holes in a page. In such a case, the client need not receive the whole page and the special fetch allows us to avoid a disk read or extra network communication. If the page is reconstructible, the server sends the updates in the m-cache to the client and marks the page as complete; otherwise, it treats the request as a page-fetch request.

4.2 Transaction Commits

Commits are handled as described in Section 3.2 with one addition: when a transaction committed by a client *A* modifies an object on page *p*, the server modifies the directory to mark all complete copies of *p* at clients other than *A* as reconstructible.

4.3 Installation

Client caches are used to avoid installation reads, in a manner similar to what happens for fetches. If a client *B* has a complete or reconstructible copy of page *p*, the server sends it an *installation fetch* message that includes updates for *p*. *B* completes *p*, i.e., fills the holes with the received updates, and sends it back to the server, which marks *B* as having a complete copy of *p*. If no client *B* has a complete or reconstructible copy, the page is read from disk and the changes are installed in it in the usual way.

Then the server writes the page to disk, and removes its updates from the m-cache. Additionally, it modifies the directory to mark all reconstructible copies of *p* as unreconstructible. This is necessary because after discarding these modified objects from the m-cache, the server can no longer bring those copies up-to-date using the m-cache.

Figure 2 illustrates using page *p* from client *B*'s cache to avoid an installation read. It also shows client *C*'s entry being marked unreconstructible after the installation has been performed.

4.4 Discussion

Our scheme preserves correctness because only complete and reconstructible pages are used in fetches from the cooperative cache; this is equivalent to fetching pages from disk in the earlier system. Unreconstructible pages are never used: Servers cause pages to become unreconstructible only by removing objects from the m-cache; they record this information in directories and never request unreconstructible pages from clients. Clients cause pages to become unreconstructible only via evicting copies of modified objects; they keep track of this information and refuse to satisfy client and installation fetch requests with unreconstructible (or missing) pages.

Sending modifications to the client as part of fetching the page from its cache is similar to update propagation [3] but is not exactly the same. The server sends the m-cache updates relatively rarely, e.g., the updates may be sent after many updates to the same page have occurred. Furthermore, the server does not update all cached copies of the page but only the copy it uses to satisfy the client or installation fetch.

It is possible to reduce the number of modified objects sent from the m-cache to the clients. This can be done by augmenting the status information in the server directories to record the latest transaction whose modifications are reflected in the client's cache, and sending only the modifications made by transactions that committed later. We rejected this optimization because our current scheme is simpler and requires less storage at servers.

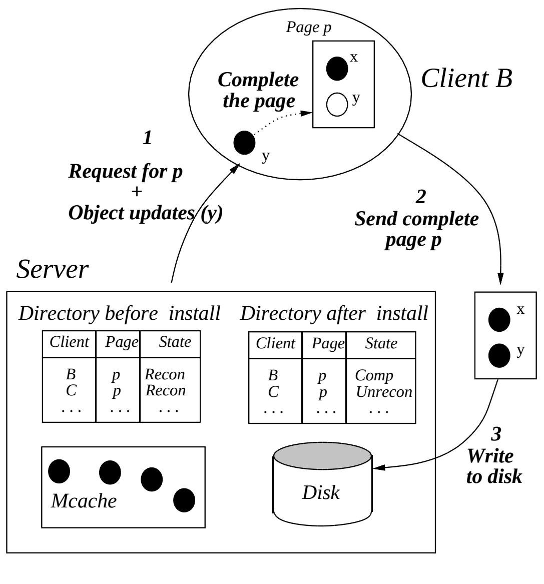


Figure 2: Installation fetch from cooperative cache.

5 Performance Evaluation

This section evaluates the costs and benefits of fragment reconstruction and split caching. We do not attempt to show the benefits of cooperative caching, since that has been shown by others [8, 11, 12, 20].

The key performance goals are reductions in client cache miss latency and in the latency to install modified objects. Therefore, we start (Section 5.1) by presenting a simple analytic model of the latency of different types of client and installation fetches. Section 5.2 uses micro-benchmarks run in our prototype and published performance numbers for fast networks and disks to estimate values for the variables used in the model. These values are used to compute an estimated latency for the different types of fetches and show that our techniques introduce only a small overhead over the basic cooperative caching mechanism.

The average fetch latency is determined not only by the latency of each type of fetch but also by the number of fetches of each type. Therefore, we ran the multi-user OO7 benchmark[2] in a version of our prototype instrumented to count the number of fetches of each type. The counts obtained were fed to the analytic model and used to predict average fetch latency. These results are presented in Section 5.3. They show that our techniques retain the benefits of cooperative caching in workloads with coarse-grained sharing and can significantly improve performance in workloads with fine-grained sharing; they reduce average client fetch latency by up to a factor of 3 and the number of disk reads to service fetches by up to a factor of 3.5. Furthermore, installation fetches from the cooperative cache reduce the number of installation reads by up to a factor of 52 and the average installation fetch latency by up to a factor of 11. The significant reductions in the number of disk reads show that our techniques improve the scalability of the system when disk I/O is the performance bottleneck.

We chose the experimental methodology just described instead of directly measuring elapsed times because our machines are connected by a slow Ethernet network; elapsed times measured in our environment would be dominated by the network overheads. Furthermore, cooperative caching was designed for a large client population but we have only a small number of client machines. We simulate several client machines using a single machine, which prevents us from collecting meaningful elapsed times. Our experimental methodology allows us to circumvent these problems while still obtaining sound results.

5.1 Analytic Model

This section presents a simple analytic model to estimate the cost of fetches and installation of modified objects in the database. Using the notation shown in Figure 3, the fetch and installation costs for various cases in our scheme are:

$D(x)$	Disk time taken to read x bytes
$N(x)$	Network latency for message with x bytes (including processing overheads)
$P(j)$	Processor time taken to perform job j
p	Size of a page
m	Average size of a message containing modified objects
r	Size of a request message (without data)

Figure 3: Parameters of the Model.

1. Modified objects fetched from m-cache:

$$F(\textit{m-cache}) = N(r) + N(m)$$

2. Page fetch from server disk:
$$F(\textit{disk}) = N(r) + D(p) + N(p)$$

3. Page fetch from a client with a complete page:
$$F(\textit{complete}) = N(r) + N(r) + P(\textit{unswiz}) + N(p)$$

4. Page fetch from a client with a reconstructible page:
$$F(\textit{reconstructible}) = N(r) + N(m) + P(\textit{recon}) + P(\textit{unswiz}) + N(p)$$

5. Installation fetch from server disk: $I(\textit{disk}) = D(p)$

6. Installation fetch from client with a complete page:
$$I(\textit{complete}) = N(r) + P(\textit{unswiz}) + N(p)$$

7. Installation fetch from client with a reconstructible page:
$$I(\textit{reconstructible}) = N(m) + P(\textit{recon}) + P(\textit{unswiz}) + N(p)$$

Here, *P(recon)* is the cost to reconstruct a fragment by filling its holes. *P(unswiz)* is the cost of unswizzling. Like many other object-oriented databases, Thor uses pointer swizzling [17] to make code at clients run faster: persistent references stored in cached copies of objects are replaced by virtual memory pointers before they are used. Swizzled references in a page must be unswizzled into persistent references before the page can be shipped to another client.

Our cache coherence mechanism introduces extra processing overhead at the server to maintain the status of pages cached by clients; we ignore these costs because they are negligible compared to the total fetch or installation costs.

5.2 Latency of Different Fetch Types

This section presents results of microbenchmarks that measure the processing costs defined in our analytic model in Section 5.1. The results show that the overhead introduced by our cache consistency mechanism on the fetch path is small. The experiments were run on an unloaded DEC Alpha 3000/400, 133 MHz, workstation running DEC/OSF1; we used a page size of 8 KB.

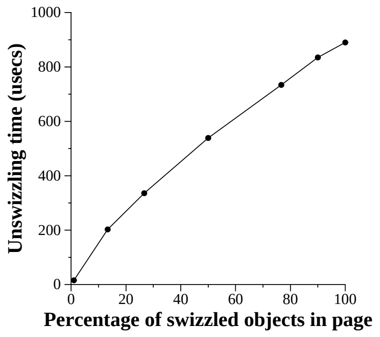


Figure 4: Elapsed time unswizzling a page, $P(unswiz)$

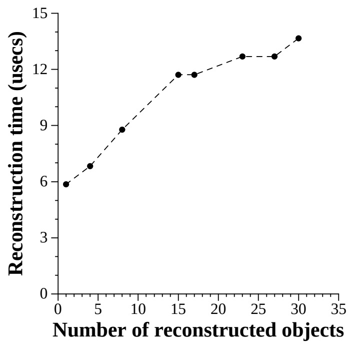


Figure 5: Elapsed time reconstructing a page, $P(recon)$.

Figures 4 and 5 presents the unswizzling and reconstruction costs in our prototype as the number of objects to be unswizzled or reconstructed is increased. Reconstruction times are low because they only involve setting up an I/O vector for a *readv* routine to read the new versions of modified objects into the cache. Unswizzling costs are high because unswizzling is memory intensive. However, in the experiments described in the next section, we observed an average percentage of swizzled objects per-page of 25%; the unswizzling cost for this percentage is about 320 μs. Furthermore, the unswizzling cost is not directly related to support for fine-grained sharing; this cost will be incurred by any cooperative caching system that uses pointer swizzling. On the other hand, unswizzling does increase the cost of fetching pages (whether complete or reconstructible) from client caches. Our algorithm also incurs a cost for restoring objects modified by the current transaction to their committed state; we ignore this cost because restoring rarely occurred in our experiments.

Figures 6 and 7 present the predicted fetch and installation fetch latencies using the analytic model, the previous graphs, and the worst case observed values for number of objects unswizzled or reconstructed that we observed in the experiments described in Section 5.3. The network times are for a fast implementation of TCP over ATM, U-Net [22], and the disk times are for a modern disk, a Barracuda 4 [21].

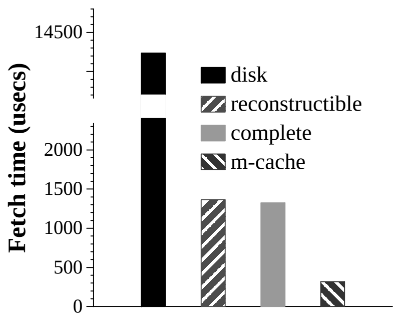


Figure 6: Predicted fetch latencies.

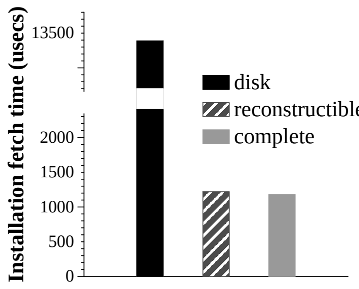


Figure 7: Predicted installation fetch latencies.

Note that the y-axis is broken to represent the true disk times.

Fetches from disk have the same latency with or without our techniques, as do fetches of complete pages from other client caches. Split caching and fragment reconstruction introduce two new fetch types: fetches of modified objects from the m-cache and fetches of reconstructible pages from other client caches. The first type of fetch is the fastest, and can therefore reduce fetch overheads relative to other systems when there is true fine-grained sharing. Fetching reconstructible pages from other client caches is only slightly more expensive than fetching complete pages because reconstruction messages were smaller than 512 bytes in our experiments and the processing cost of reconstructing a page is lower than 14μs.

We ignored contention for the disk and network in our model. Our scheme does not increase disk contention relative to other systems. Since reconstruction messages and replies to fetches from the m-cache are small, we also expect contention on the network to be similar in our prototype and in other cooperative caching systems.

Our scheme slows down clients that service fetch requests. These clients need to send and receive messages, reconstruct pages and unswizzle pages. The only cost that is related to support for fine-grained sharing is the reconstruction cost, and it is low.

Therefore, we conclude that split caching and fragment reconstruction do not introduce any significant overhead

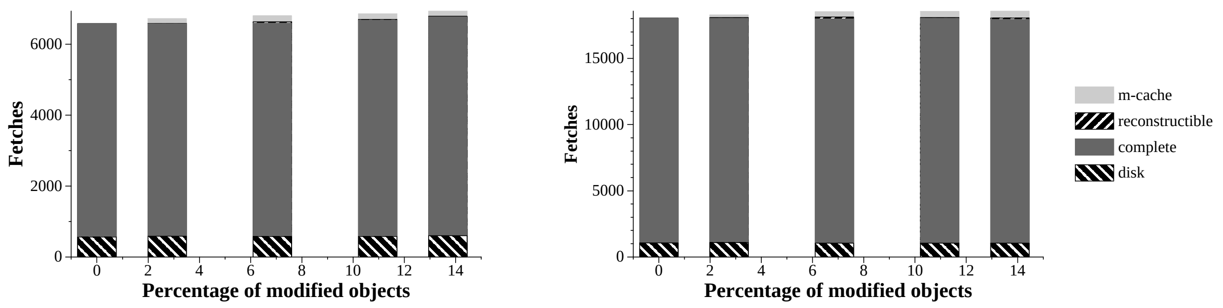


Figure 8: Fetch breakdown for coarse-grained sharing with 20% and 50% shared accesses (6 MB m-cache).

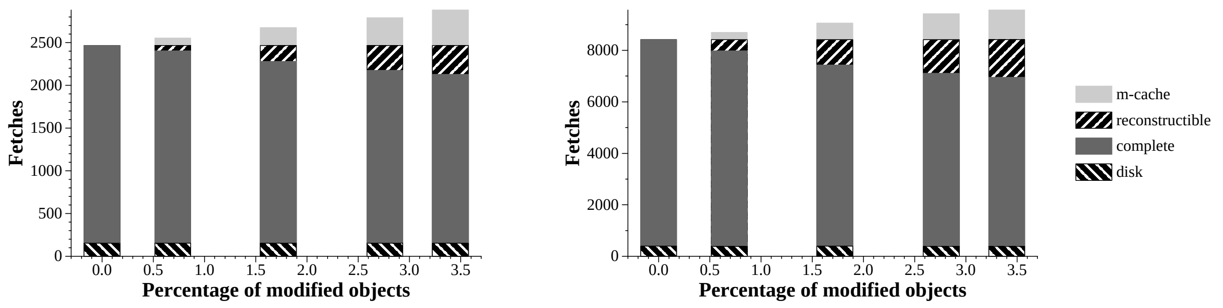


Figure 9: Fetch breakdown for fine-grained sharing with 20% and 50% shared accesses (6 MB m-cache).

on the fetch path relative to the basic cooperative caching mechanism. The predicted installation fetch latencies in Figure 7 show that installation fetches from client caches can significantly speed up installations.

5.3 OO7 Benchmark

This section describes experiments that ran the multi-user OO7 benchmark[2, 5] in our prototype to determine the distribution of fetches by type. We chose the multi-user OO7 benchmark to generate the workloads, because it is a standard benchmark and it allows us to control the percentages of shared and write accesses.

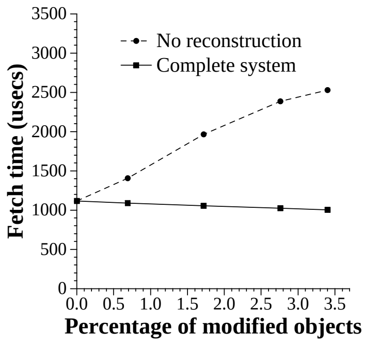
The OO7 database contains one *private module* per client and a *shared module*. Each private module has a tree of *assembly* objects and 500 *composite parts*. The internal nodes of this tree have 3 children, and its leaves point to 3 composite parts chosen randomly from among the module's composite parts. The shared module scales with the number of clients. It has a root assembly object with one subtree per client. Each subtree has 100 composite parts. The subtrees are identical to the trees in the private module, except that they are one level shorter. The leaves of a subtree point to 3 composite parts chosen randomly from among the composite parts corresponding to the subtree. Each composite part contains a graph of 20 *atomic parts* linked by *connection* objects; each atomic part has 3 outgoing connections. The total size of the database is 85 MB.

Clients traverse the database concurrently. Each traversal consists of 2000 transactions. Transactions randomly pick the client's private module or the shared module, choose a random path down the module's assembly tree, and execute an operation on one of the composite parts referenced by the leaf assembly. This operation can be a *read* or a *write* operation, and it can be *coarse-grained* or *fine-grained*. Read and write coarse-grained operations execute a depth-first traversal of the entire graph of atomic parts in a composite part; read operations do not modify any object; and write operations modify all the atomic parts. The fine-grained operations read or write a random atomic part in the composite part graph. Coarse-grained write transactions modify 13.7% of the accessed objects and fine-grained write transactions modify 3.4% of the accessed objects.

5.3.1 Experimental Setup

We evaluated two different versions of our system: one that does not use fragment reconstruction and one that does. The first version of the system allows clients to cache pages with holes, but prevents clients and the server from fetching pages with holes from other client caches. Therefore, individual clients still benefit from fine-grained sharing support because they need not evict a page just because some object in that page was modified by another client. We do not evaluate this benefit because that has been done by others [4]. Instead, we evaluate the benefit of allowing clients to fetch pages with holes from other client caches.

We ran our experiments for a system with 8 clients, 4 each on 2 machines, and a single server on another machine. Each client manages its cache greedily, without global cache coordination. All the clients had a cache of 14 MB. The database size is 76% of the sum of the cache sizes of all clients, but only 16% of the database fits in the cache of a



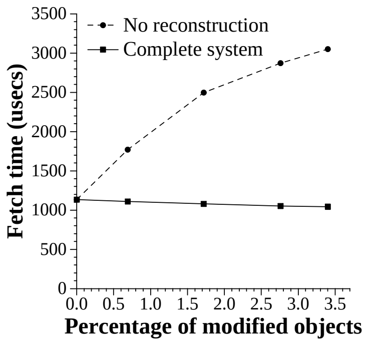


Figure 10: Predicted average fetch times for fine-grained sharing with 20% and 50% shared accesses (6 MB m-cache).

single client. The server had a small page cache (500 KB) used to store pages while modifications from the m-cache are being installed in them; the number of fetches serviced from this cache is negligible. The m-cache was 6 MB. These settings represent a reasonable configuration where most fetches are serviced from other clients' caches. These settings are adequate because we do not intend to prove that cooperative caching is a good technique, but to demonstrate the impact of the fragment reconstruction algorithm in a configuration where cooperative caching works well.

We ran the traversals twice starting from cold client caches. The results presented are from the second run to filter out cold cache misses.

5.3.2 Fragment Reconstruction

Figures 8 and 9 show the number of client fetches serviced at each level of the caching hierarchy for coarse and fine-grained transactions. The x-axis corresponds to the average percentage of objects modified by a transaction relative to the total number of objects it accesses. We varied the percentage of modified objects by changing the mix of read and write operations. The *m-cache* label corresponds to fetches that are serviced by sending the modifications cached in the m-cache; *reconstructible* includes all fetches that are serviced by reconstructing a page in another client's cache; *complete* corresponds to the fetches serviced by using a complete page in another client's cache; and *disk* includes all the fetches that are serviced from disk.

Figure 8 corresponds to a workload where operations are coarse grained, and therefore sharing is coarse-grained. In the graph on the left, 20% of the accesses go to the shared region; in the graph on the right, 50% go to the shared region. There are more fetches when the percentage of shared accesses is 50% because of reduced temporal locality.

As expected, in the coarse-grained workload the performance benefit of fragment reconstruction is not significant. Most fetches are serviced using complete pages in other client caches; at most 3% of fetches benefit from reconstruction. This happens because most pages in the database have objects from a single composite part graph, and write operations modify all atomic parts in this graph. Therefore, the last writer of a page is likely to have a complete version of the page. While not improving performance in this workload, our consistency mechanism does not degrade performance, because the cost of fetching complete pages from other client caches is mostly independent of split caching and fragment reconstruction (as explained in the previous section).

Figure 9 presents results for a workload where operations are fine grained and there is a high degree of false sharing. In the graph on the left, 20% of accesses go to the shared region; in the graph on the right, 50% go to the shared region. As expected, the importance of fragment reconstruction increases with the percentage of modified objects.

In this fine-grained sharing workload, up to 28% of fetches take advantage of fragment reconstruction and the number of disk reads needed to service fetch requests is reduced by a factor of up to 3.5. Each operation reads and writes a single atomic part in a composite part graph. Therefore, clients can access a cached copy of a page repeatedly without incurring coherence misses. Furthermore, a significant portion of the misses that do occur can be serviced from the m-cache, which provides a fast path for the transmission of new values when there is true sharing.

The portion of fetches in the reconstructible category is also high, because the copy of a page *p* cached by a client is likely to have modifications made by that client, but not modifications recently made by other clients. Therefore, it is likely that no client is caching a complete copy of *p*. When a client that is not caching *p* accesses an object in *p*, our scheme reconstructs the fragment cached by one of the clients and avoids a disk access.

5.3.3 Average Fetch Latency

With the fetch counts obtained above, we used the analytic model to predict the average fetch time for each workload (using parameters for modern disks and networks, as described in Section 5.2). Figure 10 shows the average fetch times predicted for both versions of the system for the ex-

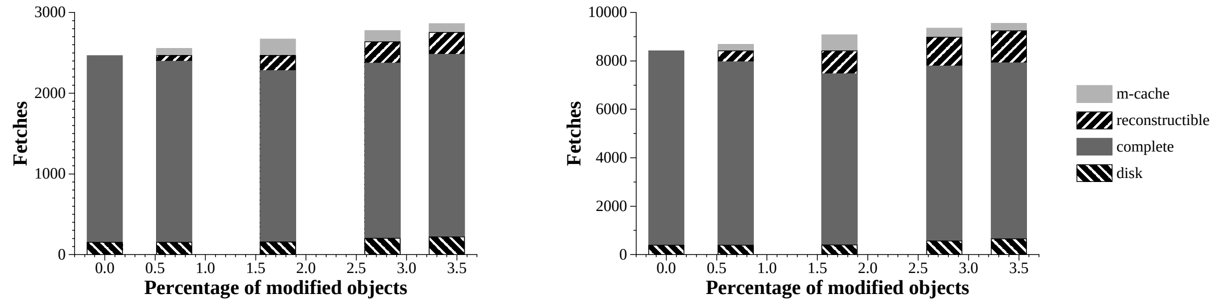


Figure 11: Fetch breakdown for fine-grained sharing with 20% and 50% shared accesses (2 MB m-cache).

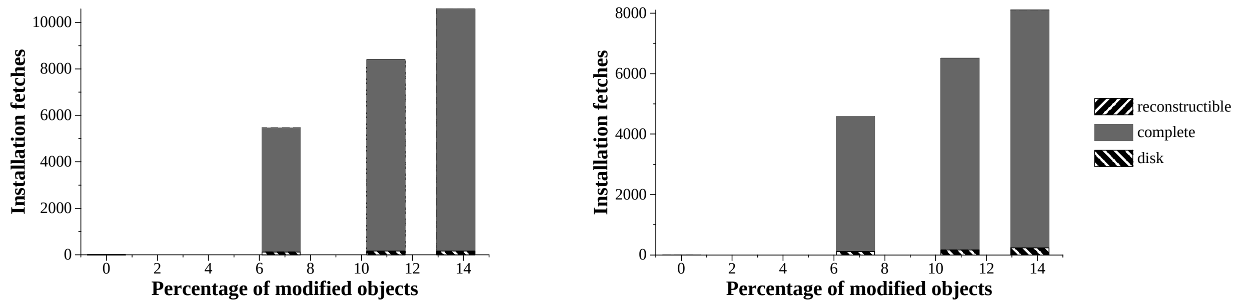


Figure 12: Installation fetch breakdown for coarse-grained sharing with 20% and 50% shared accesses (6 MB m-cache).

perimental points shown in Figure 9. We can observe that reconstruction reduces the average fetch time significantly, by up to a factor of 3, when compared to a version of our prototype where clients can only fetch complete pages from other client caches. The total number of fetches increases with the percentage of modified objects because there are more coherence misses. In either system, a significant portion of these misses is serviced from the m-cache. In the system with reconstruction, this causes the average fetch time to decrease when the percentage of writes increases. In the system without reconstruction, this benefit is offset by the increase in the number of fetches from disk (to fetch the reconstructible pages), and the average fetch time increases with the percentage of modified objects.

The fetch frequency counts potentially depend on the rate of m-cache truncation. The 6 MB m-cache is not being truncated in the fine-grained sharing workload because all modifications fit in the m-cache. Therefore, we ran the same experiments with a 2 MB m-cache. The results are presented in Figure 11. Even with this small m-cache, there is no m-cache truncation for workloads with less than 2.5% modified objects. Figures 9 and 11 show that when there are at least 2.5% modified objects, a 2 MB m-cache causes more fetches to be satisfied with complete pages or disk reads, and fewer fetches to be satisfied from the m-cache or with reconstructible pages. These effects occur because the m-cache is smaller (so that we hit there less often); the server propagates updates from the m-cache to clients on installation fetches (so that there are more complete pages); and because some pages at clients become unreconstructible when the m-cache is truncated. However, the differences between the two graphs are small and do not affect the predicted average fetch times significantly. With the small m-cache, the predicted average fetch time increases by at most 6% in the worst case. We also ran experiments with a 1 MB m-cache and the difference in predicted average fetch times (relative to the experiments with the 2 MB m-cache) is always less than 1%. Therefore, we conclude that reducing the size of the m-cache does not significantly degrade the performance of fragment reconstruction for these workloads.

5.3.4 Installation Read Avoidance

This section presents results showing the number of installation fetches serviced at each level of the caching hierarchy. Figure 12 corresponds to a workload where operations are coarse-grained, and Figure 13 corresponds to a workload where operations are fine-grained. In the graphs on the left, 20% of accesses go to the shared region; in the graphs on the right, 50% go to the shared region. We use a 6 MB m-cache for the coarse-grained workload and a 2 MB m-cache for the fine-grained workload. The number of installation fetches increases with the percentage of modified objects as expected, and there are no installation fetches with fewer than 2.7% modified objects. In the coarse-grained workload, there are very few installation fetches from reconstructible pages for the same reasons that there were very few fetches from reconstructible pages in this workload.

The results show that only a very small percentage of installation fetches are serviced from disk; our technique reduces the number of installation reads by a factor of 52.

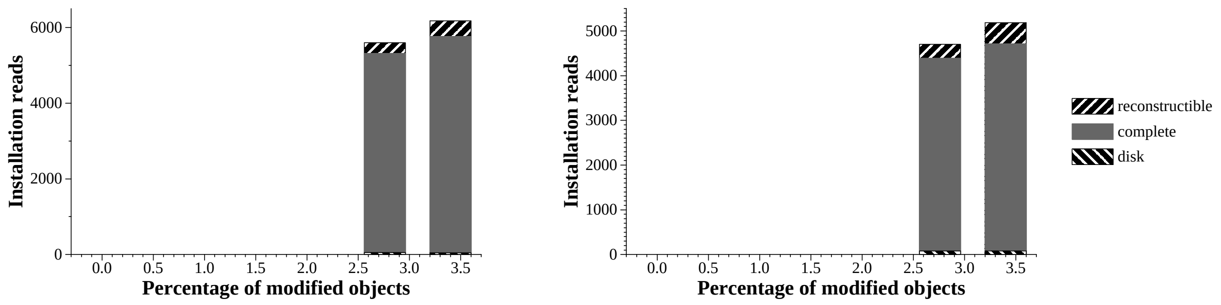


Figure 13: Installation fetch breakdown for fine-grained sharing with 20% and 50% shared accesses (2 MB m-cache).

Therefore, servicing installation reads from client caches can substantially reduce disk bandwidth consumption. This technique also reduces the average installation latency predicted by the analytic model by a factor of approximately 11 for both types of workloads (1.15 msec as opposed to 12.6 msec).

6 Conclusions

This paper has presented a new architecture, split caching with fragment reconstruction, for use in transactional distributed object storage systems. The architecture allows such systems to benefit from two techniques that have been shown to improve performance, cooperative caching and support for fine-grained sharing.

The paper also presents results of performance studies that investigated the benefits of our approach. The studies show that our techniques add little overhead in cases where there is no fine-grained sharing, and provide substantial performance improvements when fine-grained sharing affects the pages fetched from the cooperative cache. In particular, fetching recent updates from the m-cache to fill in holes provides good performance when there is true sharing, and fragment reconstruction allows use of pages in the cooperative cache when there is false sharing. In addition, our techniques substantially reduce the load on server disks by reducing disk reads for both fetches and installations. These results justify further study of the new architecture to complete the analysis, compare it with other techniques, and determine the best policy for performing page replacement in the cooperative cache.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *SIGMOD*, 1995.

[2] M. Carey et al. A Status Report on the OO7 OODBMS Benchmarking Effort. In *OOPSLA Proceedings*, 1994.

[3] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data caching tradeoffs in client-server DBMS architectures. In *SIGMOD*, 1991.

[4] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *SIGMOD*, 1994.

[5] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *SIGMOD*, 1993.

[6] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *In ACM Transactions on Computer Systems*, August 1994.

[7] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *SIGMETRICS*, 1994.

[8] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *OSDI*, 1994.

[9] A. Delis and N. Roussopoulos. Performance and Scalability of Client-Server Database Architecture. In *VLDB*, 1992.

[10] M. Feeley, J. Chase, V. Narasayya, and H. Levy. Integrating Coherency and Recoverability in Distributed Systems. In *OSDI*, 1994.

[11] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *SOSP*, 1995.

[12] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *VLDB*, 1992.

[13] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases.* PhD thesis, Massachusetts Institute of Technology, 1995.

[14] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases.* PhD thesis, Massachusetts Institute of Technology, 1997.

[15] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, 1992.

[16] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Henessy. The Directory Based Cache Coherence Protocol for the DASH multiprocessor. In *ISCA*, 1990.

[17] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *SIGMOD*, 1996.

[18] J. O'Toole and L. Shrira. Shared data management needs adaptive methods. In *Proceedings of IEEE Workshop on Hot Topics in Operating Systems*, 1995.

[19] James O'Toole and Liuba Shrira. Opportunistic Log: Efficient Installation Reads in a Reliable Object Server. In *OSDI*, 1994.

[20] P. Sarkar and J. Hartman. Efficient Cooperative Caching Using Hints. In *OSDI*, 1996.

[21] Seagate Technology. http://www.seagate.com, March 1997.

[22] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *SOSP*, 1996.