BuddyCache: High-Performance Object Storage for Collaborative Strong-Consistency Applications in a WAN

Magnus E. Bjornsson
Department of Computer Science
Brandeis University
Waltham, MA 02454-9110
magnus@cs.brandeis.edu

Liuba Shrira
Department of Computer Science
Brandeis University
Waltham, MA 02454-9110
liuba@cs.brandeis.edu

ABSTRACT

Collaborative applications provide a shared work environment for groups of networked clients collaborating on a common task. They require strong consistency for shared persistent data and efficient access to fine-grained objects. These properties are difficult to provide in wide-area networks because of high network latency.

BuddyCache is a new transactional caching approach that improves the latency of access to shared persistent objects for collaborative strong-consistency applications in high-latency network environments. The challenge is to improve performance while providing the correctness and availability properties of a transactional caching protocol in the presence of node failures and slow peers.

We have implemented a BuddyCache prototype and evaluated its performance. Analytical results, confirmed by measurements of the BuddyCache prototype using the multiuser 007 benchmark indicate that for typical Internet latencies, e.g. ranging from 40 to 80 milliseconds round trip time to the storage server, peers using BuddyCache can reduce by up to 50% the latency of access to shared objects compared to accessing the remote servers directly.

Keywords

Cooperative caching, object storage systems, fine-grain sharing, transactions, wide-area network, fault-tolerance

1. INTRODUCTION

Improvements in network connectivity erode the distinction between local and wide-area computing and, increasingly, users expect their work environment to follow them wherever they go. Nevertheless, distributed applications may perform poorly in wide-area network environments. Network bandwidth problems will improve in the foreseeable future, but improvement in network latency is fundamentally limited. *BuddyCache* is a new object caching technique that addresses the network latency problem for collaborative applications in wide-area network environment.

Collaborative applications provide a shared work environment for groups of networked users collaborating on a common task, for example a team of engineers jointly overseeing a construction project. Strong-consistency collaborative applications, for example CAD systems, use client/server transactional object storage systems to ensure consistent access to shared persistent data. Up to now however, users have rarely considered running network storage systems over wide-area networks as performance would be unacceptable [23]. For transactional storage systems, the high cost of wide-area network interactions to maintain data consistency is a major cost limiting the performance. Therefore, in wide-area network environments, collaborative applications are often adapted to use weaker consistency storage systems [21]. Adapting an application to use weak consistency storage system requires significant effort since the application needs to be rewritten to deal with a different storage system semantics. If shared persistent objects could be accessed with low-latency, a new field of distributed strong-consistency applications could be opened.

Cooperative web caching [8, 9, 14] is a well-known approach to reducing client interaction with a server by allowing one client to obtain missing objects from a another client instead of the server. Collaborative applications seem a particularly good match to benefit from this approach since one of the hard problems, namely determining what objects are cached where, becomes easy in small groups typical of collaborative settings. However, cooperative web caching techniques do not provide two important properties needed by collaborative applications, strong consistency and efficient access to fine-grained objects. Cooperative object caching systems [2] provide these properties. However, they rely on interaction with the server to provide fine-grain cache coherence that avoids the problem of *false sharing* when accesses to unrelated objects appear to conflict because they occur on the same physical page. Interaction with the server increases latency. The contribution of this work is extending cooperative caching techniques to provide strong consistency and efficient access to fine-grain objects in wide-area environments.

Consider a team of engineers employed by a construction company overseeing a remote project and working in a shed at the construction site. The engineers use a collaborative CAD application to revise and update complex project design documents. The shared documents are stored in transactional repository servers at the company home site. The engineers use workstations running repository clients. The workstations are interconnected by a fast local Ethernet but the network connection to the home repository servers is slow. To improve access latency, clients fetch objects from repository servers and cache and access them locally. A coherence protocol ensures that client caches remain consistent when objects are modified. The performance problem facing the collaborative application is coordinating with the servers consistent access to shared objects.

With BuddyCache, a group of close-by collaborating clients, connected to storage repository via a high-latency link, can avoid interactions with the server if needed objects, updates or coherency information are available in some client in the group.

BuddyCache presents two main technical challenges. One challenge is how to provide efficient access to shared fine-grained objects in the collaborative group without imposing performance overhead on the entire caching system. The other challenge is to support fine-grain cache coherence in the presence of slow and failed nodes.

BuddyCache uses a "redirection" approach similar to one used in cooperative web caching systems [9]. A redirector server, interposed between the clients and the remote servers, runs on the same network as the collaborating group and, when possible, replaces the function of the remote servers. If the client request can not be served locally, the redirector forwards it to a remote server. When one of the clients in the group fetches a shared object from the repository, the object is likely to be needed by other clients. BuddyCache redirects subsequent requests for this object to the caching client. Similarly, when a client creates or modifies a shared object, the new data is likely to be of potential interest to all group members. BuddyCache uses redirection to support *peer update*, a lightweight "application-level multicast" technique that provides group members with consistent access to the new data committed within the collaborating group without imposing extra overhead outside the group.

Nevertheless, in a transactional system, redirection interferes with shared object availability. *Solo commit*, is a technique used by BuddyCache to avoid the undesirable client dependencies that reduce object availability when some client nodes in the group are slow, or clients fail independently. Solo commit supports fine-grained validation using inexpensive coarse-grained coherence information.

Since redirection supports the performance benefits of reducing interaction with the server but introduces extra processing cost due to availability mechanisms and request forwarding, this raises the question is the "cure" worse than the "disease"? We designed and implemented a BuddyCache prototype and studied its performance benefits and costs using analytical modeling and system measurements. We compared the storage system performance with and without BuddyCache and considered how the cost-benefit balance is affected by network latency.

Analytical results, supported by measurements based on the multi-user 007 benchmark, indicate that for typical Internet latencies BuddyCache provides significant performance benefits, e.g. for latencies ranging from 40 to 80 milliseconds round trip time, clients using the BuddyCache can reduce by up to 50% the latency of access to shared objects compared to the clients accessing the repository directly. These significant performance gains could make strong consistency object storage systems more attractive for collaborative applications in wide-area environments.

2. RELATED WORK

Cooperative caching techniques [19, 15, 11, 2, 27] provide access to client caches to avoid high disk access latency in an environment where servers and clients run on a fast local area network. These techniques use the server to provide redirection and do not consider issues of high network latency.

Multiprocessor systems and distributed shared memory systems [12, 4, 16, 17, 5] use fine-grain coherence techniques to avoid the performance penalty of false sharing but do not address issues of availability when nodes fail.

Cooperative Web caching techniques, (e.g. [9, 14]) investigate issues of maintaining a directory of objects cached in nearby proxy caches in wide-area environment, using distributed directory protocols for tracking cache changes. This work does not consider issues of consistent concurrent updates to shared fine-grained objects.

Cheriton and Li propose MMO [10] a hybrid web coherence protocol that combines invalidations with updates using multicast delivery channels and receiver-reliable protocol, exploiting locality in a way similar to BuddyCache. This multicast transport level solution is geared to the single writer semantics of web objects. In contrast, BuddyCache uses "application level" multicast and a sender-reliable coherence protocol to provide similar access latency improvements for transactional objects. Application level multicast solution in a middle-ware system was described by Pendarakis, Shi and Verma in [26]. The schema supports small multi-sender groups appropriate for collaborative applications and considers coherence issues in the presence of failures but does not support strong consistency or fine-grained sharing.

Yin, Alvisi, Dahlin and Lin [30, 29] present a hierarchical WAN cache coherence scheme. The protocol uses leases to provide fault-tolerant call-backs and takes advantage of nearby caches to reduce the cost of lease extensions. The study uses simulation to investigate latency and fault tolerance issues in hierarchical avoidance based coherence scheme. In contrast, our work uses implementation and analysis to evaluate the costs and benefits of redirection and fine grained updates in an optimistic system. Anderson, Eastham and

Vahdat in WebFS [28] present a global file system coherence protocol that allows clients to choose on per file basis between receiving updates or invalidations. Updates and invalidations are multicast on separate channels and clients subscribe to one of the channels. The protocol exploits application specific methods e.g. last-writer-wins policy for broadcast applications, to deal with concurrent updates but is limited to file systems.

Mazieres studies a bandwidth saving technique [23] to detect and avoid repeated file fragment transfers across a WAN when fragments are available in a local cache. BuddyCache provides similar bandwidth improvements when objects are available in the group cache.

3. BUDDYCACHE

High network latency imposes performance penalty for transactional applications accessing shared persistent objects in wide-area network environment. This section describes the BuddyCache approach for reducing the network latency penalty in collaborative applications and explains the main design decisions.

We consider a system in which a distributed transactional object repository stores objects in highly reliable servers, perhaps outsourced in data-centers connected via high bandwidth reliable networks. Collaborating clients interconnected via a fast local network, connect via high-latency, possibly satellite, links to the servers at the data-centers to access shared persistent objects. The servers provide disk storage for the persistent objects. A persistent object is owned by a single server. Objects may be small (order of 100 bytes for programming language objects [22]). To amortize the cost of disk and network transfer objects are grouped into physical pages.

To improve object access latency, clients fetch the objects from the servers and cache and access them locally. A transactional cache coherence protocol runs at clients and servers to ensure that client caches remain consistent when objects are modified. The performance problem facing the collaborating client group is the high latency of coordinating consistent access to the shared objects.

BuddyCache architecture is based on a request redirection server, interposed between the clients and the remote servers. The interposed server (the redirector) runs on the same network as the collaborative group and, when possible, replaces the function of the remote servers. If the client request can not be served locally, redirector forwards it to a remote server. Redirection approach has been used to improve the performance of web caching protocol. BuddyCache redirector supports the correctness, availability and fault-tolerance properties of transactional caching protocol [18]. The correctness property ensures one-copy serializability of the objects committed by the client transactions. The availability and fault-tolerance properties ensure that a crashed or slow client does not disrupt any other client's access to persistent objects.

The three types of client server interactions in a transactional caching protocol are the commit of a transaction, the fetch of an object missing in a client cache, and the exchange of cache coherence information. BuddyCache avoids interactions with the server when a missing object, or cache coherence information needed by a client is available within the collaborating group. The redirector always interacts with the servers at commit time because only storage servers provide transaction durability in a way that ensures committed data remains available in the presence of client or redirector failures. Figure 1 shows the overall BuddyCache architecture.

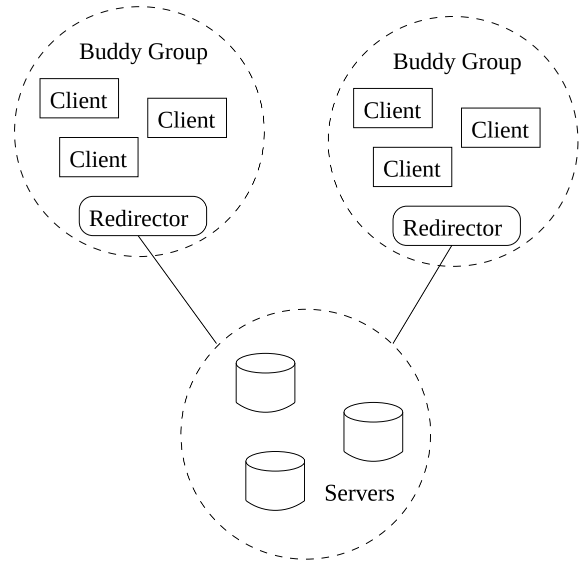

Figure 1: BuddyCache.

3.1 Cache Coherence

The redirector maintains a directory of pages cached at each client to provide cooperative caching [19, 15, 11, 2, 27], redirecting a client fetch request to another client that caches the requested object. In addition, redirector manages cache coherence.

Several efficient transactional cache coherence protocols [18] exist for persistent object storage systems. Protocols make different choices in granularity of data transfers and granularity of cache consistency. The current best-performing protocols use page granularity transfers when clients fetch missing objects from a server and object granularity coherence to avoid false (page-level) conflicts. The transactional caching taxonomy [18] proposed by Carey, Franklin and Livny classifies the coherence protocols into two main categories according to whether a protocol avoids or detects access to stale objects in the client cache. The BuddyCache approach could be applied to both categories with different performance costs and benefits in each category.

We chose to investigate BuddyCache in the context of OCC [3], the current best performing detection-based protocol. We chose OCC because it is simple, performs well in high-latency networks, has been implemented and we had access to the implementation. We are investigating BuddyCache with PSAA [31], the best performing avoidance-based protocol. Below we outline the OCC protocol [3]. The OCC protocol uses object-level coherence. When a client requests a miss-

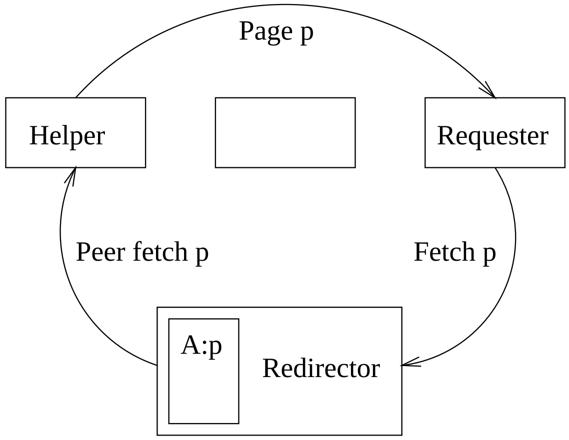

Figure 2: Peer fetch

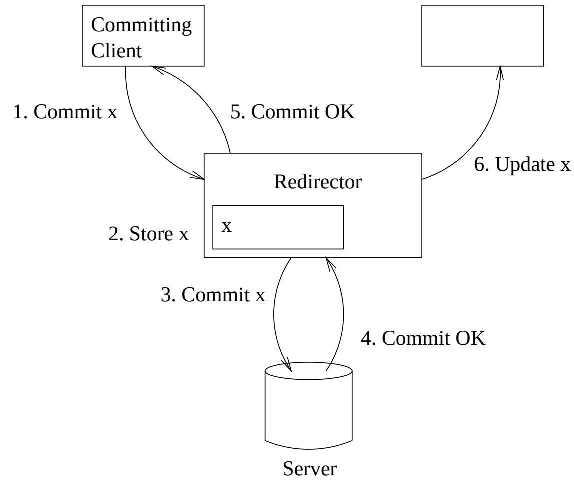

Figure 3: Peer update.

ing object, the server transfers the containing page. Transaction can read and update locally cached objects without server intervention. However, before a transaction commits it must be "validated"; the server must make sure the validating transaction has not read a stale version of some object that was updated by a successfully committed or validated transaction. If validation fails, the transaction is aborted. To reduce the number and cost of aborts, a server sends background object invalidation messages to clients caching the containing pages. When clients receive invalidations they remove stale objects from the cache and send background acknowledgements to let server know about this.

Since invalidations remove stale objects from the client cache, invalidation acknowledgement indicates to the server that a client with no outstanding invalidations has read up-to-date objects. An unacknowledged invalidation indicates a stale object may have been accessed in the client cache. The validation procedure at the server aborts a client transaction if a client reads an object while an invalidation is outstanding.

The "acknowledged invalidation" mechanism supports object-level cache coherence without object-based directories or per-object version numbers. Avoiding per-object overheads is very important to reduce performance penalties [3] of managing many small objects, since typical objects are small. An important BuddyCache design goal is to maintain this benefit.

Since in BuddyCache a page can be fetched into a client cache without server intervention (as illustrated in figure 2), cache directories at the servers keep track of pages cached in each collaborating group rather than each client. Redirector keeps track of pages cached in each client in a group. Servers send to the redirector invalidations for pages cached in the entire group. The redirector propagates invalidations from servers to affected clients. When all affected clients acknowledge invalidations, redirector can propagate the "group acknowledgement" to the server.

3.2 Light-weight Peer Update

When one of the clients in the collaborative group creates or modifies shared objects, the copies cached by any other client become stale but the new data is likely to be of potential interest to the group members. The goal in BuddyCache is to provide group members with efficient and consistent access to updates committed within the group without imposing extra overhead on other parts of the storage system.

The two possible approaches to deal with stale data are cache invalidations and cache updates. Cache coherence studies in web systems (e.g. [7]) DSM systems (e.g. [5]), and transactional object systems (e.g. [18]) compare the benefits of update and invalidation. The studies show the benefits are strongly workload-dependent. In general, invalidation-based coherence protocols are efficient since invalidations are small, batched and piggybacked on other messages. Moreover, invalidation protocols match the current hardware trend for increasing client cache sizes. Larger caches are likely to contain much more data than is actively used. Update-based protocols that propagate updates to low-interest objects in a wide-area network would be wasteful. Nevertheless, invalidation-based coherence protocols can perform poorly in high-latency networks [10] if the object's new value is likely to be of interest to another group member. With an invalidation-based protocol, one member's update will invalidate another member's cached copy, causing the latter to perform a high-latency fetch of the new value from the server.

BuddyCache circumvents this well-known bandwidth vs. latency trade-off imposed by update and invalidation protocols in wide-area network environments. It avoids the latency penalty of invalidations by retaining and propagating updates committed by one client to other clients within the group. This avoids the bandwidth penalty of updates because servers propagate invalidations to the redirectors. As far as we know, this use of localized multicast in BuddyCache redirector is new and has not been used in earlier caching systems.

The peer update works as follows. An update commit request from a client arriving at the redirector contains the object updates. Redirector retains the updates and propagates the request to the coordinating server. After the transaction commits, the coordinator server sends a commit reply

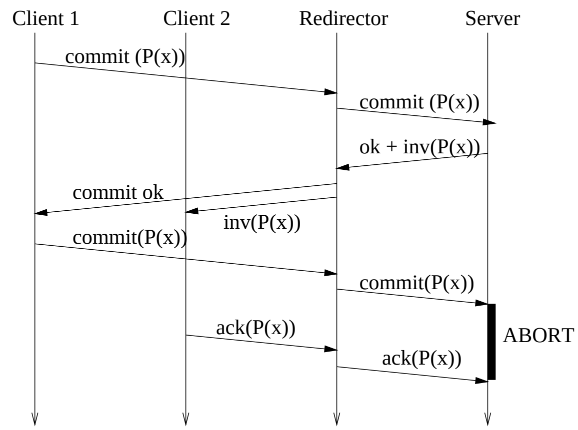

Figure 4: Validation with Slow Peers

to the redirector of the committing client group. The redirector forwards the reply to the committing client, and also propagates the retained committed updates to the clients caching the modified pages (see figure 3). Since the groups outside the BuddyCache propagate invalidations, there is no extra overhead outside the committing group.

3.3 Solo commit

In the OCC protocol, clients acknowledge server invalidations (or updates) to indicate removal of stale data. The straightforward "group acknowledgement" protocol where redirector collects and propagates a collective acknowledgement to the server, interferes with the availability property of the transactional caching protocol [18] since a client that is slow to acknowledge an invalidation or has failed can delay a group acknowledgement and prevent another client in the group from committing a transaction. E.g. an engineer that commits a repeated revision to the same shared design object (and therefore holds the latest version of the object) may need to abort if the "group acknowledgement" has not propagated to the server.

Consider a situation depicted in figure 4 where a client $C1$ commits a transaction T that reads the latest version of an object x on page P recently modified by $C1$. If the commit request for T reaches the server before the collective acknowledgement from $C2$ for the last modification of x arrives at the server, the OCC validation procedure considers x to be stale and aborts T (because, as explained above, an invalidation unacknowledged by a client, acts as indication to the server that the cached object value is stale at the client).

Note that while invalidations are not required for the correctness of the OCC protocol, they are very important for the performance since they reduce the performance penalty of aborts. The asynchronous invalidations are an important part of the reason OCC has competitive performance with PSAA [31], the best performing avoidance-based protocol [3].

Nevertheless, since invalidations are sent and processed asynchronously, invalidation processing may be arbitrarily delayed at a client. Lease-based schemes (time-out based) have been proposed to improve the availability of hierarchical callback-based coherence protocols [30] but the asynchronous nature of invalidations makes the lease-based approaches inappropriate for asynchronous invalidations.

The *Solo commit* protocol allows a client with up-to-date objects to commit a transaction even if the group acknowledgement is delayed due to slow or crashed collaborators. The protocol requires clients to include extra information with the transaction read sets in the commit message, to indicate to the server the objects read by the transaction are up-to-date.

Object version numbers could provide a simple way to track up-to-date objects but, as mentioned above, maintaining per object version numbers imposes unacceptably high overheads (in disk storage, I/O costs, and directory size) in the entire object system when objects are small [22]. Instead, BuddyCache uses a coarse-grain page version number to identify a fine-grain object version. A page version number is incremented at a server when at transaction that modifies objects on the page commits. Updates committed by a single transaction and corresponding invalidations are therefore uniquely identified by the modified page version number.

Page version numbers are propagated to clients in fetch replies, commit replies and with invalidations, and clients include page version numbers in commit requests sent to the servers. If a transaction fails validation due to missing "group acknowledgement", the server checks page version numbers of the objects in the transaction read set and allows the transaction to commit if the client has read from the latest page version.

Note that the use of version numbers can enable independent commits but page version checks only detect page-level conflicts. To detect object-level conflicts and avoid the problem of false sharing we need the "acknowledged invalidations". Section 4 describes the details of the implementation of solo commit support for fine-grain sharing.

3.4 Group Configuration

The BuddyCache architecture supports multiple concurrent peer groups. Potentially, it may be faster to access data cached in another peer group than to access a remote server. In such case extending BuddyCache protocols to support multi-level peer caching could be worthwhile. We have not pursued this possibility for several reasons.

In web caching workloads, simply increasing the population of clients in a proxy cache often increases the overall cache hit rate. In BuddyCache applications, however, we expect sharing to result mainly from explicit client interaction and collaboration, suggesting that inter-group fetching is unlikely to occur. Moreover, measurements from multi-level web caching systems indicate that a multi-level system may not be advantageous unless the network connection between the peer groups is very fast. We are primarily interested in environments where closely collaborating peers have fast close-range connectivity, but the connection between peer groups may be slow. As a result, we decided that support for inter-group fetching in BuddyCache is not a high priority right now.

To support heterogenous resource-rich and resource-poor peers, the BuddyCache redirector can be configured to run either in one of the peer nodes or, when available, in a separate node within the site infrastructure. Moreover, in a resource-rich infrastructure node, the redirector can be configured as a stand-by peer cache to receive pages fetched by other peers, emulating a central cache somewhat similar to a regional web proxy cache. From the BuddyCache cache coherence protocol point of view, however, such a stand-by peer cache is equivalent to a regular peer cache and therefore we do not consider this case separately in the discussion in this paper.

4. IMPLEMENTATION

In this section we provide the details of BuddyCache implementation. We have implemented BuddyCache in Thor client/server object-oriented database [22]. Thor supports high performance access to distributed objects and therefore provides a good test platform to investigate BuddyCache performance.

4.1 Base Storage System

Thor 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 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 contains cache directory and a recoverable modified object cache called the *mcache*. The directory keeps track of which pages are cached by which clients. The mcache holds recently modified objects that have not yet been written back to their pages on disk. As mcache fills up, a background process propagates modified objects to the disk [20, 25].

4.2 Base Cache Coherence

Transactions are serialized using optimistic concurrency control OCC [3] described in Section 3.1. We provide some of the relevant OCC protocol implementation details. 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 mcache. The mcache is recoverable, i.e. if the server crashes, the mcache is reconstructed at recovery by scanning the log.

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 and the 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 *incomplete* with "holes" in place of the invalidated objects. Performing invalidation on an object basis means that false sharing does not cause unnecessary aborts; keeping incomplete pages in the client cache means that false sharing does not lead to unnecessary cache misses. Clients acknowledge invalidations to indicate removal of stale data as explained in Section 3.1. 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.

4.3 Redirection

The redirector runs on the same local network as the peer group, in one of the peer nodes, or in a special node within the infrastructure. It maintains a directory of pages available in the peer group and provides fast centralized fetch redirection (see figure 2) between the peer caches. To improve performance, clients inform the redirector when they evict pages or objects by piggybacking that information on messages sent to the redirector.

To ensure up-to-date objects are fetched from the group cache the redirector tracks the status of the pages. A cached page is either *complete* in which case it contains consistent values for all the objects, or *incomplete*, in which case some of the objects on a page are marked *invalid*. Only complete pages are used by peer fetch. The protocol for maintaining page status when pages are updated and invalidated is described in Section 4.4.

When a client request has to be processed at the servers, e.g., a complete requested page is unavailable in the peer group or a peer needs to commit a transaction, the redirector acts as a server proxy: it forwards the request to the server, and then forwards the reply back to the client. In addition, in response to invalidations sent by a server, the redirector distributes the update or invalidation information to clients caching the modified page and, after all clients acknowledge, propagates the group acknowledgement back to the server (see figure 3). The redirector-server protocol is, in effect, the client-server protocol used in the base Thor storage system, where the combined peer group cache is playing the role of a single client cache in the base system.

4.4 Peer Update

The peer update is implemented as follows. An update commit request from a client arriving at the redirector contains the object updates. Redirector retains the updates and propagates the request to the coordinator server. After a transaction commits, using a two phase commit if needed, the coordinator server sends a commit reply to the redirector of the committing client group. The redirector forwards the reply to the committing client. It waits for the invalidations to arrive to propagate corresponding retained (committed) updates to the clients caching the modified pages (see figure 3.)

Participating servers that are home to objects modified by the transaction generate object invalidations for each cache group that caches pages containing the modified objects (including the committing group). The invalidations are sent lazily to the redirectors to ensure that all the clients in the groups caching the modified objects get rid of the stale data.

In cache groups other than the committing group, redirectors propagates the invalidations to all the clients caching the modified pages, collect the client acknowledgements and after completing the collection, propagate collective acknowledgements back to the server.

Within the committing client group, the arriving invalidations are not propagated. Instead, updates are sent to clients caching those objects' pages, the updates are acknowledged by the client, and the collective acknowledgement is propagated to the server.

An invalidation renders a cached page unavailable for peer fetch changing the status of a *complete* page p into an *incomplete*. In contrast, an update of a *complete* page preserves the complete page status. As shown by studies of the fragment reconstruction [2], such update propagation allows to avoid the performance penalties of false sharing. That is, when clients within a group modify different objects on the same page, the page retains its *complete* status and remains available for peer fetch. The effect is similar to "eager" fragment reconstruction [**?**].

We have also considered the possibility of allowing a peer to fetch an incomplete page (with invalid objects marked accordingly) but decided against this possibility because of the extra complexity involved in tracking invalid objects.

4.5 Vcache

The *solo* commit validation protocol allows clients with up-to-date objects to commit independently of slower (or failed) group members. As explained in Section 3.3, the solo commit protocol allows a transaction T to pass validation if extra coherence information supplied by the client indicates that transaction T has read up-to-date objects. Clients use page version numbers to provide this extra coherence information. That is, a client includes the page version number corresponding to each object in the read object set sent in the commit request to the server. Since a unique page version number corresponds to each committed object update, the page version number associated with an object allows the validation procedure at the server to check if the client transaction has read up-to-date objects.

The use of coarse-grain page versions to identify object versions avoids the high penalty of maintaining persistent object versions for small objects, but requires an extra protocol to maintain the mapping from a cached object to the identifying page version (*ObjectToVersion*). The main implementation issue is concerned with maintaining this mapping efficiently.

Servers associate page version numbers with the invalidations. If an unacknowledged invalidation is pending for an object x read by a transaction T, the validation procedure checks if the version number for x in T's read set matches the version number for highest pending invalidation for x, in which case the object value is current, otherwise T fails validation.

We note again that the page version number-based checks, and the invalidation acknowledgement-based checks are complimentary in the solo commit validation and both are needed. The page version number check allows the validation to proceed before invalidation acknowledgements arrive but by itself a page version number check detects page-level conflicts and is not sufficient to support fine-grain coherence without the object-level invalidations.

We now describe how the client manages the mapping *ObjectToVersion*. The client maintains a page version number for each cached page. The version number satisfies the following invariant VP about the state of objects on a page: if a cached page P has a version number v, then the value of an object o on a cached page P is either invalid or it reflects at least the modifications committed by transactions preceding the transaction that set P's version number to v.

New object values and new page version numbers arrive when a client fetches a page or when a commit reply or invalidations arrive for this page. The new object values modify the page and, therefore, the page version number needs to be updated to maintain the invariant VP. A page version number that arrives when a client fetches a page, replaces the page version number for this page. Such an update preserves the invariant VP. Similarly, an in-sequence page version number arriving at the client in a commit or invalidation message advances the version number for the entire cached page, without violating VP. However, invalidations or updates and their corresponding page version numbers can also arrive at the client out of sequence, in which case updating the page version number could violate VP. For example, a commit reply for a transaction that updates object x on page P in server S_1, and object y on page Q in server S_2, may deliver a new version number for P from the transaction coordinator S_1 before an invalidation generated for an earlier transaction that has modified object r on page P arrives from S_1 (as shown in figure 5).

The cache update protocol ensures that the value of any object o in a cached page P reflects the update or invalidation with the highest observed version number. That is, obsolete updates or invalidations received out of sequence do not affect the value of an object.

To maintain the *ObjectToVersion* mapping and the invariant VP in the presence of out-of-sequence arrival of page version numbers, the client manages a small version number cache *vcache* that maintains the mapping from an object into its

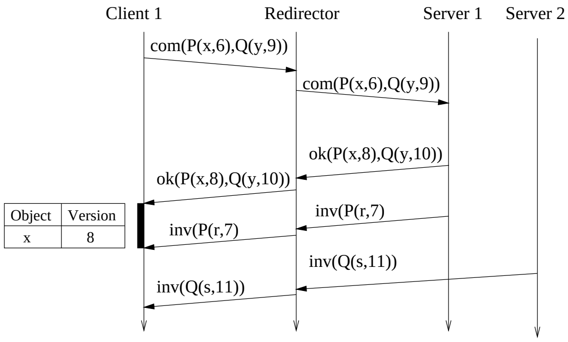

Figure 5: Reordered Invalidations

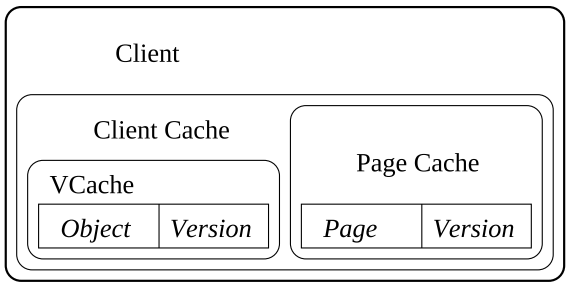

Figure 6: ObjectToVersion map with vcache

corresponding page version number for all reordered version number updates until a complete page version number sequence is assembled. When the missing version numbers for the page arrive and complete a sequence, the version number for the entire page is advanced.

The *ObjectToVersion* mapping, including the vcache and page version numbers, is used at transaction commit time to provide version numbers for the read object set as follows. If the read object has an entry in the vcache, its version number is equal to the highest version number in the vcache for this object. If the object is not present in the vcache, its version number is equal the version number of its containing cached page. Figure 6 shows the *ObjectToVersion* mapping in the client cache, including the page version numbers for pages and the vcache.

Client can limit vcache size as needed since re-fetching a page removes all reordered page version numbers from the vcache. However, we expect version number reordering to be uncommon and therefore expect the vcache to be very small.

5. BUDDYCACHE FAILOVER

A client group contains multiple client nodes and a redirector that can fail independently. The goal of the failover protocol is to reconfigure the BuddyCache in the case of a node failure, so that the failure of one node does not disrupt other clients from accessing shared objects. Moreover, the failure of the redirector should allow unaffected clients to keep their caches intact.

We have designed a Failover protocols for BuddyCache but have not implemented it yet. Appendix 9 outlines the protocol.

6. PERFORMANCE EVALUATION

BuddyCache redirection supports the performance benefits of avoiding communication with the servers but introduces extra processing cost due to availability mechanisms and request forwarding. Is the "cure" worse then the "disease?" To answer the question, we have implemented a BuddyCache prototype for the OCC protocol and conducted experiments to analyze the performance benefits and costs over a range of network latencies.

6.1 Analysis

The performance benefits of peer fetch and peer update are due to avoided server interactions. This section presents a simple analytical performance model for this benefit. The avoided server interactions correspond to different types of client cache misses. These can be cold misses, invalidation misses and capacity misses. Our analysis focuses on cold misses and invalidation misses, since the benefit of avoiding capacity misses can be derived from the cold misses. Moreover, technology trends indicate that memory and storage capacity will continue to grow and therefore a typical BuddyCache configuration is likely not to be cache limited.

The client cache misses are determined by several variables, including the workload and the cache configuration. Our analysis tries, as much as possible, to separate these variables so they can be controlled in the validation experiments.

To study the benefit of avoiding cold misses, we consider cold cache performance in a read-only workload (no invalidation misses). We expect peer fetch to improve the latency cost for client cold cache misses by fetching objects from nearby cache. We evaluate how the redirection cost affects this benefit by comparing and analyzing the performance of an application running in a storage system with BuddyCache and without (called Base).

To study the benefit of avoiding invalidation misses, we consider hot cache performance in a workload with modifications (with no cold misses). In hot caches we expect BuddyCache to provide two complementary benefits, both of which reduce the latency of access to shared modified objects. Peer update lets a client access an object modified by a nearby collaborating peer without the delay imposed by invalidation-only protocols. In groups where peers share a read-only interest in the modified objects, peer fetch allows a client to access a modified object as soon as a collaborating peer has it, which avoids the delay of server fetch without the high cost imposed by the update-only protocols.

Technology trends indicate that both benefits will remain important in the foreseeable future. The trend towards increase in available network bandwidth decreases the cost of the update-only protocols. However, the trend towards increasingly large caches, that are updated when cached objects are modified, makes invalidation-base protocols more attractive.

To evaluate these two benefits we consider the performance of an application running without BuddyCache with an application running BuddyCache in two configurations. One, where a peer in the group modifies the objects, and another where the objects are modified by a peer outside the group.

Peer update can also avoid invalidation misses due to false-sharing, introduced when *multiple* peers update different objects on the same page concurrently. We do not analyze this benefit (demonstrated by earlier work [2]) because our benchmarks do not allow us to control object layout, and also because this benefit can be derived given the cache hit rate and workload contention.

6.1.1 The Model

The model considers how the time to complete an execution with and without BuddyCache is affected by invalidation misses and cold misses.

Consider k clients running concurrently accessing uniformly a shared set of N pages in BuddyCache (BC) and $Base$. Let $t_{fetch}(S)$, $t_{redirect}(S)$, $t_{commit}(S)$, and $t_{compute}(S)$ be the time it takes a client to, respectively, fetch from server, peer fetch, commit a transaction and compute in a transaction, in a system S, where S is either a system with BuddyCache (BC) or without ($Base$). For simplicity, our model assumes the fetch and commit times are constant. In general they may vary with the server load, e.g. they depend on the total number of clients in the system.

The number of misses avoided by peer fetch depends on k, the number of clients in the BuddyCache, and on the client co-interest in the shared data. In a specific BuddyCache execution it is modeled by the variable r, defined as a number of fetches arriving at the redirector for a given "version" of page P (i.e. until an object on the page is invalidated).

Consider an execution with cold misses. A client starts with a cold cache and runs read-only workload until it accesses all N pages while committing l transactions. We assume there are no capacity misses, i.e. the client cache is large enough to hold N pages. In BC, r cold misses for page P reach the redirector. The first of the misses fetches P from the server, and the subsequent $r-1$ misses are redirected. Since each client accesses the entire shared set $r = k$.

Let $T_{cold}(Base)$ and $T_{cold}(BC)$ be the time it takes to complete the l transactions in $Base$ and BC.

$$\begin{aligned} T_{cold}(Base) &= N * t_{fetch}(Base) \\ &\quad +(t_{compute} + t_{commit}(Base)) * l \end{aligned} \quad (1)$$

$$\begin{aligned} &T_{cold}(BC) = \\ &\quad N * \left(\frac{1}{k} * t_{fetch}(BC) + (1-\frac{1}{k}) * t_{redirect}\right) \\ &\quad +(t_{compute} + t_{commit}(BC)) * l \end{aligned} \quad (2)$$

Consider next an execution with invalidation misses. A client starts with a hot cache containing the working set of N pages. We focus on a simple case where one client (writer) runs a workload with modifications, and the other clients (readers) run a read-only workload.

In a group containing the writer (BC_W), peer update eliminates all invalidation misses. In a group containing only readers (BC_R), during a steady state execution with uniform updates, a client transaction has $miss_{inv}$ invalidation misses. Consider the sequence of r client misses on page P that arrive at the redirector in BC_R between two consequent invalidations of page P. The first miss goes to the server, and the $r-1$ subsequent misses are redirected. Unlike with cold misses, $r \leq k$ because the second invalidation disables redirection for P until the next miss on P causes a server fetch.

Assuming uniform access, a client invalidation miss has a chance of $1/r$ to be the first miss (resulting in server fetch), and a chance of $(1-1/r)$ to be redirected.

Let $T_{inval}(Base)$, $T_{inval}(BC_R)$ and $T_{inval}(BC_W)$ be the time it takes to complete a single transaction in the $Base$, BC_R and BC_W systems.

$$\begin{aligned} T_{inval}(Base) &= miss_{inv} * t_{fetch}(Base) \\ &\quad +t_{compute} + t_{commit}(Base) \end{aligned} \quad (3)$$

$$\begin{aligned} T_{inval}(BC_R) &= miss_{inv} * (\frac{1}{r} * t_{fetch}(BC_R) \\ &\quad +(1-\frac{1}{r}) * t_{redirect}(BC_R)) \\ &\quad +t_{compute} + t_{commit}(BC_R) \end{aligned} \quad (4)$$

$$T_{inval}(BC_W) = t_{compute} + t_{commit}(BC_W) \quad (5)$$

In the experiments described below, we measure the parameters N, r, $miss_{inv}$, $t_{fetch}(S)$, $t_{redirect}(S)$, $t_{commit}(S)$, and $t_{compute}(S)$. We compute the completion times derived using the above model and derive the benefits. We then validate the model by comparing the derived values to the completion times and benefits measured directly in the experiments.

6.2 Experimental Setup

Before presenting our results we describe our experimental setup. We use two systems in our experiments. The $Base$ system runs Thor distributed object storage system [22] with clients connecting directly to the servers. The *Buddy* system runs our implementation of BuddyCache prototype in Thor, supporting peer fetch, peer update, and solo commit, but not the failover.

Our workloads are based on the multi-user OO7 benchmark [13]; this benchmark is intended to capture the characteristics of many different multi-user CAD/CAM/CASE applications, but does not model any specific application. We use OO7 because it is a standard benchmark for measuring object storage system performance. The OO7 database contains a tree of assembly objects with leaves pointing to three *composite* parts chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic* parts linked by *connection* objects; each atomic part has 3 outgoing connections. We use a *medium* database that has 200 atomic parts per composite part. The multi-user database allocates for each client a "private" module consisting of one tree of assembly objects, and adds an extra "shared" module that scales proportionally to the number of clients.

We expect a typical BuddyCache configuration not to be cache limited and therefore focus on workloads where the objects in the client working set fit in the cache. Since the goal of our study is to evaluate how effectively our techniques deal with access to shared objects, in our study we limit client access to shared data only. This allows us to study the effect our techniques have on cold cache and cache consistency misses and isolate as much as possible the effect of cache capacity misses.

To keep the length of our experiments reasonable, we use small caches. The OO7 benchmark generates database modules of predefined size. In our implementation of OO7, the "private" module size is about 38MB. To make sure that the entire working set fits into the cache we use a single private module and choose a cache size of 40MB for each client. The OO7 database is generated with modules for 3 clients, only one of which is used in our experiments as we explain above. The objects in the database are clustered in 8K pages, which are also the unit of transfer in the fetch requests.

We consider two types of transaction workloads in our analysis, read-only and read-write. In OO7 benchmark, read-only transactions use the T1 traversal that performs a depth-first traversal of entire composite part graph. Write transactions use the T2b traversal that is identical to T1 except that it modifies all the atomic parts in a single composite. A single transaction includes one traversal and there is no sleep-time between transactions. Both read-only and read-write transactions always work with data from the same module. Clients running read-write transactions don't modify in every transaction, instead they have a 50% probability of running read-only transactions.

The database was stored by a server on a 40GB IBM 7200RPM hard drive, with a 8.5 average seek time and 40 MB/sec data transfer rates. In Base system clients connect directly to the database. In Buddy system clients connect to the redirector that connects to the database. We run the experiments with 1-10 clients in Base, and one or two 1-10 client groups in Buddy. The server, the clients and the redirectors ran on a 850MHz Intel Pentium III processor based PC, 512MB of memory, and Linux Red Hat 6.2. They were connected by a 100Mb/s Ethernet and had The server had a 50MB cache (of which 6MB were used for the modified object buffer), the client had a 40MB cache. The experiments ran in Utah experimental testbed emulab.net [1].

6.3 Basic Costs

This section analyses the basic cost of the requests in the Buddy system during the OO7 runs.

	Latency [ms]			
	Base		Buddy	
	3 group	*5 group*	*3 group*	*5 group*
Fetch	1.3	1.4	2.4	2.6
Commit	2.5	5.5	2.4	5.7

Table 1: Commit and Server fetch

Operation	Latency [ms]
PeerFetch	1.8 – 5.5
−AlertHelper	0.3 – 4.6
−CopyUnswizzle	0.24
−CrossRedirector	0.16

Table 2: Peer fetch

6.3.1 Redirection

Fetch and commit requests in the BuddyCache cross the redirector, a cost not incurred in the Base system. For a request redirected to the server (server fetch) the extra cost of redirection includes a local request from the client to redirector on the way to and from the server. We evaluate this latency overhead indirectly by comparing the measured latency of the Buddy system server fetch or commit request with the measured latency of the corresponding request in the Base system.

Table 1 shows the latency for the commit and server fetch requests in the Base and Buddy system for 3 client and 5 client groups in a fast local area network. All the numbers were computed by averaging measured request latency over 1000 requests. The measurements show that the redirection cost of crossing the redirector in not very high even in a local area network. The commit cost increases with the number of clients since commits are processed sequentially. The fetch cost does not increase as much because the server cache reduces this cost. In a large system with many groups, however, the server cache becomes less efficient.

To evaluate the overheads of the peer fetch, we measure the peer fetch latency (*PeerFetch*) at the requesting client and break down its component costs. In peer fetch, the cost of the redirection includes, in addition to the local network request cost, the CPU processing latency of crossing the redirector and crossing the helper, the latter including the time to process the help request and the time to copy, and unswizzle the requested page.

We directly measured the time to copy and unswizzle the requested page at the helper, (*CopyUnswizzle*), and timed the crossing times using a null crossing request. Table 2 summarizes the latencies that allows us to break down the peer fetch costs. CrossRedirector, includes the CPU latency of crossing the redirector plus a local network round-trip and is measured by timing a round-trip null request issued by a client to the redirector. AlertHelper, includes the time for the helper to notice the request plus a network round-trip, and is measured by timing a round-trip null request issued from an auxiliary client to the helper client. The local network latency is fixed and less than 0.1 ms.

The AlertHelper latency which includes the elapsed time from the help request arrival until the start of help request processing is highly variable and therefore contributes to the high variability of the PeerFetch time. This is because the client in Buddy system is currently single threaded and therefore only starts processing a help request when blocked waiting for a fetch- or commit reply. This overhead is not inherent to the BuddyCache architecture and could be eliminated by a multi-threaded implementation in a system with pre-emptive scheduling.

6.3.2 Version Cache

The solo commit allows a fast client modifying an object to commit independently of a slow peer. The solo commit mechanism introduces extra processing at the server at transaction validation time, and extra processing at the client at transaction commit time and at update or invalidation processing time.

The server side overheads are minimal and consist of a page version number update at commit time, and a version number comparison at transaction validation time.

The version cache has an entry only when invalidations or updates arrive out of order. This may happen when a transaction accesses objects in multiple servers. Our experiments run in a single server system and therefore, the commit time overhead of version cache management at the client does not contribute in the results presented in the section below. To gauge these client side overheads in a multiple server system, we instrumented the version cache implementation to run with a workload trace that included reordered invalidations and timed the basic operations.

The extra client commit time processing includes a version cache lookup operation for each object read by the transaction at commit request preparation time, and a version cache insert operation for each object updated by a transaction at commit reply processing time, but only if the updated page is missing some earlier invalidations or updates. It is important that the extra commit time costs are kept to a minimum since client is synchronously waiting for the commit completion. The measurements show that in the worst case, when a large number of invalidations arrive out of order, and about half of the objects modified by T2a (200 objects) reside on reordered pages, the cost of updating the version cache is 0.6 ms. The invalidation time cost are comparable, but since invalidations and updates are processed in the background this cost is less important for the overall performance. We are currently working on optimizing the version cache implementation to further reduce these costs.

6.4 Overall Performance

This section examines the performance gains seen by an application running OO7 benchmark with a BuddyCache in a wide area network.

6.4.1 Cold Misses

To evaluate the performance gains from avoiding cold misses we compare the cold cache performance of OO7 benchmark running read-only workload in the Buddy and Base systems. We derive the times by timing the execution of the systems in the local area network environment and substituting 40 ms and 80 ms delays for the requests crossing the redirector and the server to estimate the performance in the wide-area-network. Figures 7 and 8 show the overall time to complete 1000 cold cache transactions. The numbers were obtained by averaging the overall time of each client in the group. The results show that in a 40 ms network Buddy system reduces significantly the overall time compared to the Base system, providing a 39% improvement in a three client group, 46% improvement in the five client group and 56% improvement in the ten client case.

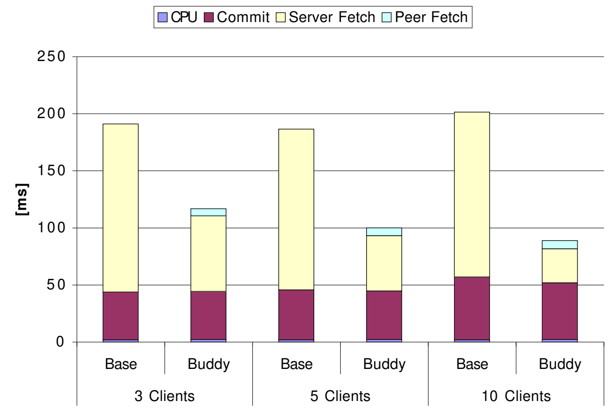

Figure 7: Breakdown for cold read-only 40ms RTT

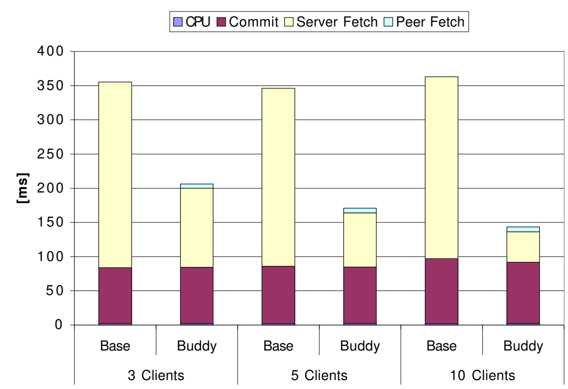

Figure 8: Breakdown for cold read-only 80ms RTT

The overall time includes time spent performing client computation, direct fetch requests, peer fetches, and commit requests.

In the three client group, Buddy and Base incur almost the same commit cost and therefore the entire performance benefit of Buddy is due to peer fetch avoiding direct fetches.

In the five client group the server fetch cost for individual client decreases because with more clients faulting in a fixed size shared module into BuddyCache, each client needs to perform less server fetches.

Figure 8 shows the overall time and cost break down in the 80 ms network. The BuddyCache provides similar performance improvements as with the 40ms network. Higher

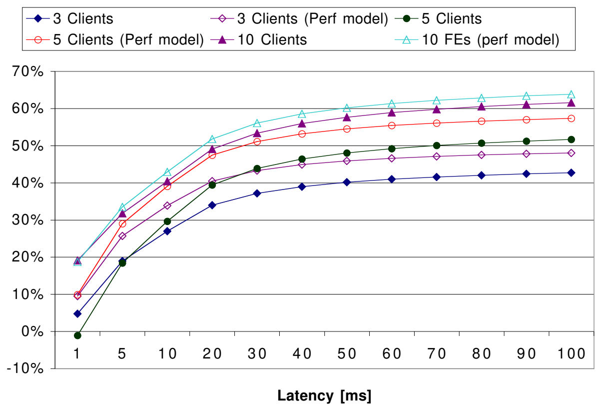

Figure 9: Cold miss benefit

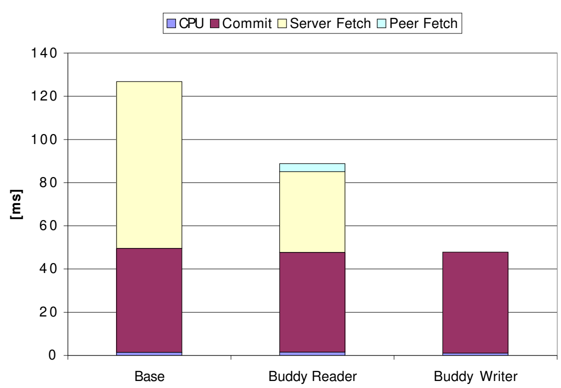

Figure 10: Breakdown for hot read-write 40ms RTT

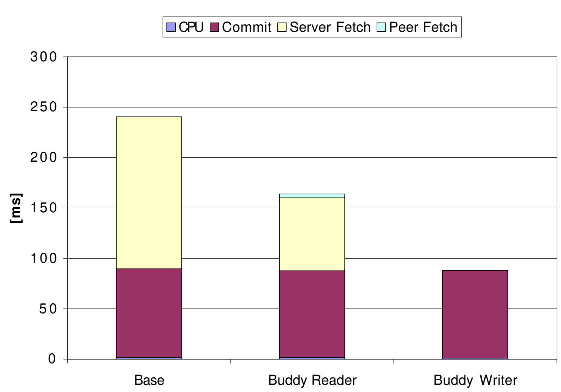

Figure 11: Breakdown for hot read-write 80ms RTT

network latency increases the relative performance advantage provided by peer fetch relative to direct fetch but this benefit is offset by the increased commit times.

Figure 9 shows the relative latency improvement provided by BuddyCache (computed as the overall measured time difference between Buddy and Base relative to Base) as a function of network latency, with a fixed server load. The cost of the extra mechanism dominates BuddyCache benefit when network latency is low. At typical Internet latencies 20ms-60ms the benefit increases with latency and levels off around 60ms with significant (up to 62% for ten clients) improvement.

Figure 9 includes both the measured improvement and the improvement derived using the analytical model in section 6.1.1. Remarkably, the analytical results predict the measured improvement very closely, albeit being somewhat higher than the empirical values. The reason why the simplified model works well is it captures the dominant performace component, network latency cost.

6.4.2 *Invalidation Misses*

To evaluate the performance benefits provided by BuddyCache due to avoided invalidation misses, we compared the hot cache performance of the Base system with two different Buddy system configurations. One of the Buddy system configurations represents a collaborating peer group modifying shared objects (Writer group), the other represents a group where the peers share a read-only interest in the modified objects (Reader group) and the writer resides outside the BuddyCache group.

In each of the three systems, a single client runs a read-write workload (writer) and three other clients run read-only workload (readers). Buddy system with one group containing a single reader and another group containing two readers and one writer models the Writer group. Buddy system with one group containing a single writer and another group running three readers models the Reader group. In Base, one writer and three readers access the server directly. This simple configuration is sufficient to show the impact of BuddyCache techniques.

Figures 10 and 11 show the overall time to complete 1000 hot cache OO7 read-only transactions. We obtain the numbers by running 2000 transactions to filter out cold misses and then time the next 1000 transactions. Here again, the reported numbers are derived from the local area network experiment results.

The results show that the BuddyCache reduces significantly the completion time compared to the Base system. In a 40 ms network, the overall time in the Writer group improves by 62% compared to Base. This benefit is due to peer update that avoids all misses due to updates. The overall time in the Reader group improves by 30% and is due to peer fetch that allows a client to access an invalidated object at the cost of a local fetch avoiding the delay of fetching from the server. The latter is an important benefit because it shows that on workloads with updates, peer fetch allows an invalidation-based protocol to provide some of the benefits of update-based protocol.

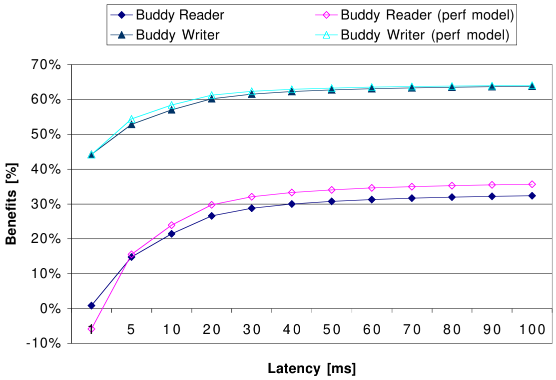

Figure 12: Invalidation miss benefit

Note that the performance benefit delivered by the peer fetch in the Reader group is approximately 50% less than the performance benefit delivered by peer update in the Writer group. This difference is similar in 80ms network.

Figure 12 shows the relative latency improvement provided by BuddyCache in Buddy Reader and Buddy Writer configurations (computed as the overall time difference between Reader and Base relative to Base, and Buddy Writer and Base relative to Base) in a hot cache experiment as a function of increasing network latency, for fixed server load.

The peer update benefit dominates overhead in Writer configuration even in low-latency network (peer update incurs minimal overhead) and offers significant 44-64% improvement for entire latency range.

The figure includes both the measured improvement and the improvement derived using the analytical model in section 6.1.1. As in cold cache experiments, here the analytical results predict the measured improvement closely. The difference is minimal in the 'writer group', and somewhat higher in the 'reader group' (consistent with the results in the cold cache experiments). As in cold cache case, the reason why the simplified analytical model works well is because it captures the costs of network latency, the main performance bottleneck.

7. CONCLUSION

Collaborative applications provide a shared work environment for groups of networked clients collaborating on a common task. They require strong consistency for shared persistent data and efficient access to fine-grained objects. These properties are difficult to provide in wide-area network because of high network latency.

This paper described *BuddyCache*, a new transactional cooperative caching [19, 15, 11, 2, 27] technique that improves the latency of access to shared persistent objects for collaborative strong-consistency applications in high-latency network environments. The technique improves performance yet provides strong correctness and availability properties in the presence of node failures and slow clients.

BuddyCache uses redirection to fetch missing objects directly from group members caches, and to support *peer update*, a new lightweight "application-level multicast" technique that gives group members consistent access to the new data committed within the collaborating group without imposing extra overhead outside the group. Redirection, however, can interfere with object availability. *Solo commit*, is a new fine-grain validation technique that only incurs the cost of coarse-grain versions and allows a client in a group to commit independently of slow or failed peers.

We have designed and implemented BuddyCache prototype in Thor distributed transactional object storage system [22] and evaluated the benefits and costs of the system over a range of network latencies. Analytical results, supported by the system measurements using the multi-user 007 benchmark indicate, that for typical Internet latencies BuddyCache provides significant performance benefits, e.g. for latencies ranging from 40 to 80 milliseconds round trip time, clients using the BuddyCache can reduce by up to 50% the latency of access to shared objects compared to the clients accessing the repository directly.

The main contributions of the paper are:

1. extending cooperative caching techniques to support consistent fine-grain access in high-latency environments,
2. an implementation of the system prototype that yields strong performance gains over the base system,
3. analytical and measurement based performance evaluation of the costs and benefits of the new techniques capturing the dominant performance cost, high network latency.

8. REFERENCES

[1] 'emulab.net', the utah network emulation facility. supported by NSF grant ANI-00-82493.

[2] A. Adya, M. Castro, B. Liskov, U. Maheshwari, and L. Shrira. Fragment reconstruction: Providing global cache coherence in a transactional storage system. *Proceedings of the International Conference on Distributed Computing Systems*, May 1997.

[3] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrencty control using loosely synchronized clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995.

[4] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. In *IEEE Computer, Vol. 29, No. 2*, 1996.

[5] C. Anderson and A. Karlin. Two Adaptive Hybrid Cache Coherency Protocols. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96)*, 1996.

[6] M. Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley, 1994.

[7] P. Cao and C. Liu. Maintaining strong cache consistency in the world wide web. In *17th International Conference on Distributed Computing Systems.*, volume 47 of *IEEE Transactions on Computers*, pages 445–57, April 1998.

[8] J. Chase, S. Gadde, and M. Rabinovich. Directory structures for scalable internet caches. Technical Report CS-1997-18, Dept. of Computer Science, Duke University, November 1997.

[9] J. Chase, S. Gadde, and M. Rabinovich. Not all hits are created equal: Cooperative proxy caching over a wide-area network. In *Third International WWW Caching Workshop*, June 1998.

[10] D. R. Cheriton and D. Li. Scalable web caching of frequently updated objects using reliable multicast. *2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[11] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.

[12] S. Dwarkadas, H. Lu, A.L. Cox, R. Rajamony, and W. Zwaenepoel. Combining compile-time and run-time support for efficient software distributed shared memory. In *Proceedings of IEEE, Special Issue on Distributed Shared Memory, Vol. 87, No. 3*, 1999.

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

[14] Li Fan, Pei Cao, Jussara Almeida, and Andrei Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of ACM SIGCOMM*, 1998.

[15] M. Feeley, W. Morgan, F. Pighin, A. Karlin, and H. Levy. Implementing global memory management in a workstation cluster. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[16] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Integrating coherency and recoverablity in distributed systems. In *Proceedings of the First Usenix Symposium on Operating sustems Design and Implementation*, May 1994.

[17] P. Ferreira and M. Shapiro et al. Perdis: design, implementation, and use of a persistent distributed store. In *Recent Advances in Distributed Systems, LNCS 1752, Springer-Verlag*, 1999.

[18] M. J. Franklin, M. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. In *ACM Transactions on Database Systems*, volume 22, pages 315–363, September 1997.

[19] Michael Franklin, Michael Carey, and Miron Livny. Global memory management for client-server dbms architectures. In *Proceedings of the 19th Intl. Conference on Very Large Data Bases (VLDB)*, 1992.

[20] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1997.

[21] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *Proceedings of the ACM CSCW Conference*, September 1988.

[22] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, June 1999.

[23] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *18th ACM Symposium on Operating Systems Principles*, 2001.

[24] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing*, 1988.

[25] J. O'Toole and L. Shrira. Opportunistic log: Efficient installation reads in a reliable object server. In *OSDI*, 1994.

[26] D. Pendarakis, S. Shi, and D. Verma. Almi: An application level multicast infrastructure. In *3nd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[27] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Usenix Symposium on Operation Systems Design and Implementation*, 1996.

[28] A. M. Vahdat, P. C. Eastham, and T. E Anderson. Webfs: A global cache coherent file system. Technical report, University of California, Berkeley, 1996.

[29] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical cache consistency in a WAN. Department of Computer Science, University of Texas at Austin.

[30] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[31] M. Zaharioudakis, M. J. Carey, and M. J. Franklin. Adaptive, Fine-Grained Sharing in a Client-Server OODBMS: A Callback-Based Approach. *ACM Transactions on Database Systems*, 22(4):570–627, December 1997.

9. APPENDIX

This appendix outlines the BuddyCache Failover protocol.

To accommodate heterogeneous clients including resource-poor hand-helds we do not require the availability of persistent storage in the BuddyCache peer group. BuddyCache design assumes that client caches and redirector data structures do not survive node failures.

A failure of a client or a redirector is detected by a membership protocol that exchanges periodic "I am alive messages" between group members and initiates a failover protocol. The failover determines the active group participants, reelects a redirector if needed, reinitializes the BuddyCache data structures in the new configuration and restarts the protocol. The group reconfiguration protocol is similar to the one presented in [24]. Here we describe how the failover manages BuddyCache state.

To restart BuddyCache protocol, the failover needs to resynchronize the redirector page directory and client-server request forwarding so that active clients can continue running transactions using their caches. In the case of a client failure, the failover removes the crashed client pages from the directory. Any response to an earlier request initiated by the failed client is ignored except a commit reply, in which case the redirector distributes the retained committed updates to active clients caching the modified pages.

In the case of the redirector failure, the failover protocol reinitializes sessions with the servers and clients, and rebuilds the page directory using a protocol similar to one in [6]. The newly restarted redirector asks the active group members for the list of pages they are caching and the status of these pages, i.e. whether the pages are *complete* or *incomplete*.

Requests outstanding at the redirector at the time of the crash may be lost. A lost fetch request will time out at the client and will be retransmitted. A transaction running at the client during a failover and committing after the failover is treated as a regular transaction, a transaction trying to commit during a failover is aborted by the failover protocol. A client will restart the transaction and the commit request will be retransmitted after the failover. Invalidations, updates or collected update acknowledgements lost at the crashed redirector could prevent the garbage collection of pending invalidations at the servers or the vcache in the clients. Therefore, servers detecting a redirector crash retransmit unacknowledged invalidations and commit replies. Unique version numbers in invalidations and updates ensure that duplicate retransmitted requests are detected and discarded.

Since the transaction validation procedure depends on the cache coherence protocol to ensure that transactions do not read stale data, we now need to argue that BuddyCache failover protocol does not compromise the correctness of the validation procedure. Recall that BuddyCache transaction validation uses two complementary mechanisms, page version numbers and invalidation acknowledgements from the clients, to check that a transaction has read up-to-date data.

The redirector-based invalidation (and update) acknowledgement propagation ensures the following invariant. When a server receives an acknowledgement for an object o modification (invalidation or update) from a client group, any client in the group caching the object o has either installed the latest value of object o, or has invalidated o. Therefore, if a server receives a commit request from a client for a transaction T reading an object o after a failover in the client group, and the server has no unacknowledged invalidation for o pending for this group, the version of the object read by the transaction T is up-to-date independently of client or redirector failures.

Now consider the validation using version numbers. The transaction commit record contains a version number for each object read by the transaction. The version number protocol maintains the invariant VP that ensures that the value of object o read by the transaction corresponds to the highest version number for o received by the client. The invariant holds since the client never applies an earlier modification after a later modification has been received. Retransmition of invalidations and updates maintains this invariant. The validation procedure checks that the version number in the commit record matches the version number in the unacknowledged outstanding invalidation. It is straightforward to see that since this check is an end-to-end client-server check it is unaffected by client or redirector failure.

The Failover protocol has not been implemented yet.