Hybrid Caching for Large-Scale Object Systems *(Think Globally, Act Locally)*

James O'Toole
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA

Liuba Shrira
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA

Abstract

Object-based client caching allows clients to keep more frequently accessed objects while discarding colder objects that reside on the same page. However, when these objects are modified and sent to the server, it may need to read the corresponding page from disk to install the update. These *installation reads* are not required with a page-based cache because whole pages are sent to the server.

We describe a hybrid system that permits clients to cache objects and pages. The system uses a simple cache design that combines the best of object caching and page caching. The client increases its cache hit ratio as in object-based caching. The client avoids some installation reads by sending pages to the server when possible. Using simulated workloads we explore the performance of our design and show that it can offer a significant performance improvement over both pure object caching and pure page caching on a range of workloads.

1 Introduction

In a client/server persistent object system, objects are fetched from the server over the network into the client cache, manipulated locally, and the modifications are sent to be committed at the server. In a scalable system many clients will be competing for server resources. Given current hardware trends, we assume that the server will be disk I/O bound. Therefore it is important to design the client/server caching system to reduce the disk I/O bandwidth consumed at the server.

Previous studies have shown that when hot data is densely packed on pages, page-based caching performs well. When the hot objects are sparsely packed,

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136, in part by the National Science Foundation under Grant CCR-8822158, and in part by the Department of the Army under Contract DABT63-92-C-0012.

object-based caching works better because it is able to hold more hot objects in the cache. However, there is an additional cost associated with object caching: On commit, an object cache sends to the server the modified object, but it does not send the page. To install the modified object onto its containing page, the server may need to read the page from the disk if it is not present in the server cache. In a previous study [13] we have shown that the cost of these installation reads can be significant.

In this paper we present a design of a hybrid cache that manages at the client both pages and objects. We suggest a hybrid cache management policy that uses a simple eviction rule to avoid some installation reads. The modified eviction rule protects some objects from eviction in order to help keep pages intact. By keeping a page intact, the client can send the whole page to the server when it modifys an object on that page. This enables the server to avoid an installation read.

To study the performance of the hybrid cache we construct a simple performance model that focuses on the I/O costs in object and page caching. Using simulation we compare the throughput of a system with object-based caching, page-based caching and hybrid caching over a range of workloads. We consider the workloads where page caching is advantageous (densely packed hot objects) workloads where object caching is advantageous (sparsely packed hot objects) and workloads that represent a combination of both. Our results show that when disk I/O is the performance bottleneck, the hybrid system can outperform both pure object caching and pure page caching.

In the following sections, we introduce the basic scalable persistent object system design (Section 2) and describe the hybrid cache and its policy (Section 3). We then introduce our simulation model (Section 4), present the experimental system configurations (Section 5), and present simulation results that illustrate the value of our techniques (Section 6). Finally, we discuss related research (Section 7) and our conclusions (Section 8).

2 Persistent Object Systems

This section introduces our baseline persistent object system and describes the context of our work. The system supports atomic modifications to persistent objects. We briefly discuss our assumptions about the system architecture. The persistent objects are stored on disk at the server. We assume that objects are grouped into *pages* and updated in place. Pages are the unit of disk transfer and caching at the server. Objects are small and a page can contain many objects. Client cache performance is the dominant factor that we consider because our focus is on reducing the disk load at the server.

Client Caching

In a persistent object system, clients fetch data from a server and operate on it in a local cache. In an object-based architecture, clients fetch objects, update them, and send back updated objects in a commit request to the server. In a page-based architecture, the client and server exchange whole pages, as shown in Figure 1.

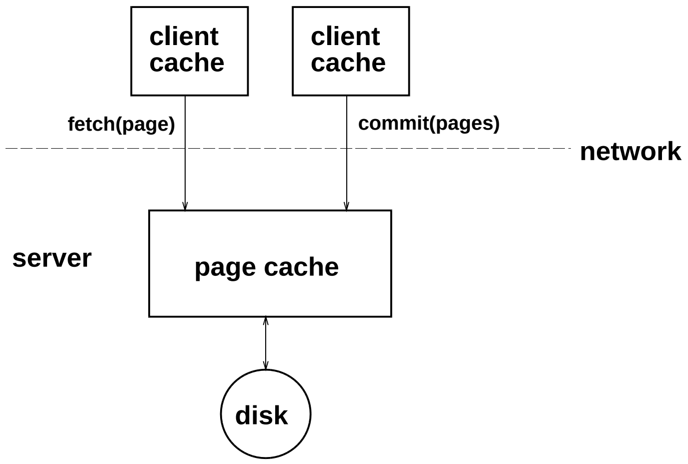

Figure 1: Clients using a Page Server

Previous work shows that each approach may be superior to the other depending on how the objects (on pages) are accessed by application programs [6]. When the clustering of objects on pages corresponds to the client access pattern, the page-based architecture should work well. On the other hand, object-based systems may pack frequently accessed objects more densely into the client cache. There are also other issues that complicate matters: swizzling, object prefetching, object clustering strategies, etc. Our focus is on disk performance, so we are ignoring these issues here.

Transaction Validation

We assume a server architecture similar to that of the Thor persistent object system [11]. The features that we assume are optimistic concurrency control with in-memory commit [2]. If the client and server are using a page-based architecture, then we assume that the page server also uses optimistic concurrency control and in-memory commit.

We use an optimistic concurrency control scheme, so a transaction that reads stale objects is aborted when the server rejects its commit request. The server uses a concurrency control protocol to ensure that all committing transactions are serialized. Committing transactions are validated by the server using a method that does not require disk access; see Adya [2] for the details. The server notifies clients when cached objects are modified, so that client caches are “almost” up-to-date.

When considering the choice between page and object servers we ignore the question of whether the server uses page level or object level concurrency control. Though this choice of granularity is important to the semantics and performance of transaction validation, it is orthogonal to the I/O costs that concern us here.

Installing Modifications

When a transaction commits in a page-based architecture, the entire page is returned to the server. For simplicity, we assume that the server is implemented using a non-volatile memory so that committing a transaction does not require an immediate disk write. Without non-volatile memory, the server would record committed modifications in a log on a dedicated disk.

When a transaction commits in an object-based architecture, the client sends modified objects back to the server. After validation, the server records the modified objects in a non-volatile log. Later, modifications from the log are applied to their corresponding pages; we call this update process *installation*.

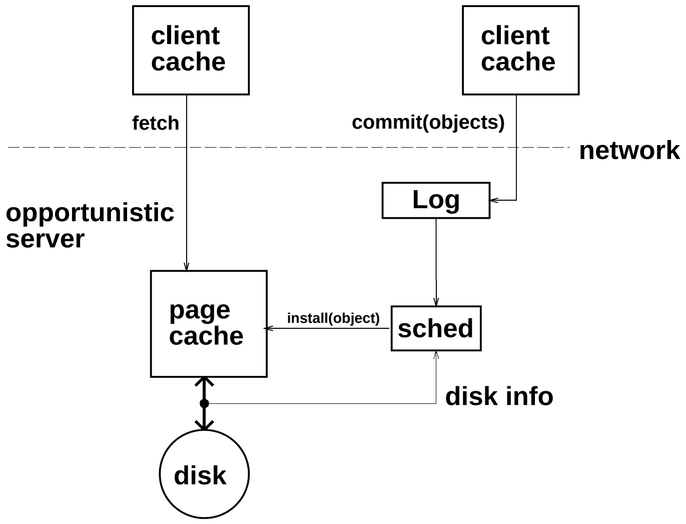

Figure 2: Object Server with Opportunistic Log

Note that installing an object modification may require a disk read if the corresponding page is not in the server cache. In recent work, we found that these *installation reads* can have a large impact on the performance of large-scale object servers [13]. We showed that the transaction log provides a large pool of pending installation reads that can be processed opportunistically, as shown in Figure 2. In particular, we showed that using well-known disk scheduling techniques allows the cost of installation reads to be reduced to a fraction of the normal random-access cost.

3 Hybrid Caching

In practice, we expect the objects that are frequently used by a client to be sometimes packed densely into pages and sometimes not. Therefore, we are motivated to design a hybrid system that permits clients to cache both objects and pages. This allows clients to selectively retain some objects from a page and discard the others, or to keep a page intact. In a hybrid system, clients may be able to take advantage of increased cache memory utilization while also avoiding some installation reads.

3.1 Hybrid Server

To enable the client to cache pages, the server must provide whole pages to the client when responding to fetch requests. Then the client will be able to return whole pages to the server, at least when it has retained the whole page in its cache. Previous work tells us that object servers can help avoid fetches by sending groups of related objects in response to fetch requests. We would expect a hybrid system to do this just as well, but we note that sending objects instead of pages may produce more installation reads.

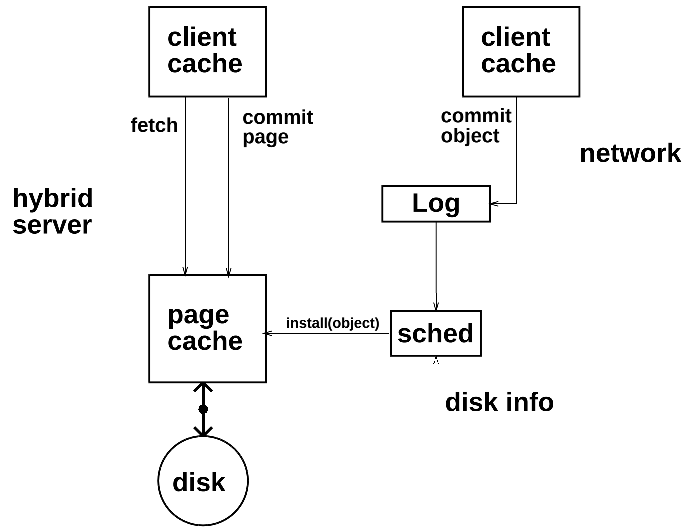

Figure 3: Hybrid Server accept Page and Object Commits

So the hybrid server provides whole pages to the client. The server accepts commit requests from the client for either whole pages or individual objects, as shown in Figure 3. When the server receives a commit request that provides a page, it validates the transaction according to the individual object that was modified (i.e. the hybrid server uses object-based concurrency control). If the transaction is valid, then the server can consider whether to use the containing page to avoid an installation read. This is possible if the other objects on the page are not stale. Otherwise, it is still possible that the valid objects on the page can be combined with pending installations to produce the whole page. But in any case, we should at least expect the hybrid server to be able to avoid an installation read whenever a page-based system would avoid concurrency control conflicts within the page.

3.2 Cache Tradeoffs

When the client receives a page from the server after a cache miss, it may already have information about how hot the objects on the page are. This kind of information could be used to guide promotion or eviction policies in any kind of cache. In a hybrid design, this information could be especially useful.

If the page contains just a few hot objects, it might be best to keep these objects but discard their cold companions. Discarding the rest of the page will prevent a later page-commit, and may very likely mean that more installation reads are required at the server. However, the memory occupied by these cold companions might be better used to hold yet more hot objects. This is what makes object-based caching work. Also, an opportunistic log [13] can make installation reads much less expensive than disk reads produced by fetch operations, because installation reads can be deferred and scheduled, unlike fetch reads which are blocking the progress of clients and must be performed immediately. Therefore the tradeoff generally favors increasing the client cache utilitization at the expense of additional installation reads.

In contrast, when a page contains mostly hot objects, it may be worth keeping a few cold companions in the cache to avoid generating more installation reads at the server. If the cache is very effective, the memory occupied by a few cold objects may not be so valuable. Keeping the cold objects that help eliminate installation reads is then beneficial (to the server). In some sense we intend that the client cache manager think globally and act locally.

3.3 Cache Policy

We do not know how to make perfect caching decisions in a hybrid system. In the discussion here, we present the motivation and intuition for the hybrid cache design rules. The concrete cache design that results from these rules is presented in Section 5.2. Here are some basic design rules for how a hybrid cache should work:

- When objects arrive in the cache they should be treated fairly, because they may be either hot or cold. If the cache is working well, then incoming objects will usually be cold. Otherwise, they will usually be hot.
- When an object in the cache is accessed, it should be promoted individually, because we infer that it may be hot, but we are not so sure about its page-companions.

These two ideas reflect basic facts about object caching, but promoting based on page relationships will be sometimes useful. We've chosen to ignore this issue to simplify our work. If page-relationships can be used in object promotion, the hybrid cache should do what the best object cache does.

The goal of our hybrid cache policy is to avoid installation reads when possible. This new motivation should affect the caching policy primarily in the area of eviction because eviction (hopefully) relates to cold objects. Here is a simple rule that captures the essential idea:

- When an object is about to be evicted, and it is the first object of its page to be evicted, give it another chance if its page-companions appear to be hot.

This policy expresses the basic motivation behind hybrid caching and completes our hybrid cache design. It seems likely to protect a cold object when doing so is likely to eliminate installation reads. It will not protect cold objects that are from mostly-cold pages.

Ideally, we would also like to base our eviction decisions on how likely the hot objects are to be *written*. There is less reason to protect a cold object whose hot page-companions are rarely modified. So predictive information about the frequency of update of objects would also be useful.

In practice, it seems entirely likely that the best sources of predictive information about object access patterns will depend on the application. Object types or historical information might be useful and could be collected by the server or the client. However, we do not aspire here to solve this part of the cache design problem. Our goal here is simply to explore the interaction between caching policy in a hybrid design and the cost of installation reads.

We know from previous work that installation reads can be optimized to be less costly than fetch reads. Therefore, we want the hybrid cache to behave mostly like an object cache and obtain the maximum possible benefit from packing hot objects into the client cache. When the pressure to evict cold objects is reduced, the hybrid cache should keep the cold objects that will eliminate the most installation reads.

4 Simulation Model

To examine cache performance tradeoffs, we built a simulator for a system of clients and a reliable server. The simulator emphasizes the disk I/O requirements of the server because we expect client cache performance to affect aggregate system throughput by loading the disk. The simulation model is described by the parameters shown in Table 1. We discuss simplifications and assumptions in the sections that follow.

4.1 Network

The network provides unlimited bandwidth and a fixed message delivery latency. Each message transmission also incurs a small cpu overhead. We ignore contention because we assume that network bandwidth will not significantly affect the performance of reliable servers. If network bandwidth were a limiting factor, then we would expect object caching to benefit because objects are smaller than pages, so commit messages for objects are smaller. In general we use low message costs to reflect our expectation that network performance will be improving much faster than disk performance in the foreseeable future.

4.2 Disk

The disk services requests issued by the server in FIFO order. The disk geometry and other performance characteristics are taken from the HP97560 drive described by Wilkes [14]. We chose this disk because it is simple, accurate, and available.

Network	
Message latency	1 msec
Per-message cpu overhead	100 μsecs
Disk	
Rotational speed	4002 rpm
Sector size	512 bytes
Sectors per track	72
Tracks per cylinder	19
Cylinders	1962
Head switch time	1.6 msec
Seek time ($\leq$ 383 cylinders)	$3.24 + 0.4\sqrt{d}$ ms
Seek time ($>$ 383 cylinders)	$8.00 + 0.008d$ ms
Server	
Database size (full disk)	335,500 pages
Page size	4 Kbytes
Log-size	100 pages
Log-entries-per-page	5
Server memory (1% of database)	3,355 pages
ValidationTime	5 μsecs
InstallationTime	1 msec
WriteTrigger	$>$ 2000 dirty pages
IReadTrigger	$<$ 50 empty log entries

Table 1: System Parameters

4.3 Server

The server processes fetch and commit requests from clients by reading and writing relevant database pages that are stored on an attached disk. The server has a non-volatile primary memory that holds cached pages. When a fetch request is received from a client, the page corresponding to the requested object is read from the disk into the server cache if necessary. The entire page is then sent to the client.

The non-volatile primary memory is also used as a transaction log. *Log-size* pages of the primary memory are statically allocated to hold log entries. The *Log-entries-per-page* parameter defines the number of log entries that can be stored per page of log memory. The log entries represent the collection of modified objects that have not yet been installed.

Concurrency control at the server is described by the *ValidationTime* parameter, which defines the cpu time required to validate a transaction. Aborted transactions are indistinguishable from read-only transactions for our purposes. If the client has provided an entire page containing a modified object in the commit request, then the server stores the page into its cache memory and marks it dirty. The server then sends a confirming message to the client. When the number of dirty pages in the cache exceeds the *WriteTrigger* threshold, the server writes one dirty page to the disk. The page is selected using the shortest positioning time algorithm [15].

If the client provides only the modified object in the transaction commit request, then the server adds the object to the log and sends a confirming message to the client. If the page containing the object is in the cache, then the object is installed immediately. However, if the page is not in the cache, then the installation is postponed. When the number of empty log entries decreases below the *IReadTrigger* threshold the server issues an installation read to obtain the page needed for a pending log entry. The server selects the pending installation from the log opportunistically, as described in previous work [13]. An installation read is initiated for the page that has the shortest positioning time, which can be determined fairly quickly using a branch and bound implementation.

Whenever a page enters the cache, whether due to a disk read or from the client, all modified objects that belong to that page are installed onto it. Every installation consumes *InstallationTime* cpu time at the server.

5 Experimental Setup

For our simulations we chose a workload setup that provides skewed accesses both among pages and within individual pages. We also implemented several cache designs for comparison in our simulations. The workload setup and cache designs are described separately below.

5.1 Client Workload

Each simulated client contains a cpu and a local memory for caching objects. The client executes a sequence of transactions, each operating on a single object in the database. If necessary, the client sends a fetch request for the object to the server and waits for the server to respond with a full page containing the object. The client then computes for *ClientThinkTime* and possibly modifies the object (*WriteRatio*). Finally the client ends the transaction by sending a commit request to the server. After the commit is confirmed by the server, the client immediately starts its next transaction. Table 2 lists the parameters that control the workload generated by the clients.

We expect scalable object systems to have many clients that compete for the server memory. However, we found it impractical to simulate more than 50 clients because our simulator uses too much memory. Therefore, we artificially chose a very small server memory (1% of the database).

We expect each client in real systems to operate mostly on private data and to cache most or all of this data locally. Each simulated client directs 90% of its operations at some set of hot pages within the database. These hot pages do not overlap with the hot pages of other clients. The other 10% of the transactions use the rest of the database, including the hot pages belonging to other clients.

In order to experiment with hot pages that have hot objects more densely or more sparsely packed upon them, we designate some of the objects on each page to be hotter than the others. Each page contains 10 objects, and 99% of the transactions that access the page choose one of the hot objects on that page. The number of hot pages and hot objects within pages varies by actual workload and is described with the experimental results in Section 6.

Client Workload Parameters	
Number of clients	50
Client cache size	1000 pages
ClientThinkTime	100 msec
Page Access Pattern	90% Hot, 10% Cold
HotWriteRatio	20%
ColdWriteRatio	0%
Objects-per-page	10
Object Access Pattern	99% to hot (per page)
Number of hot pages	(various, see section 6)
Hot objects per page	(see section 6)

Table 2: Workload Parameters

We chose this access pattern so that the page-based cache will be unaware of the number of hot objects per page, because changing the number of hot objects on a page does not change the frequency of access for the page. However, this means that the hot objects on a sparsely-hot page will be much hotter than on a densely-hot page. We don't know whether this aspect hot object distribution is realistic. Section 6.3 discusses the relevance of the hot object distribution to the relative performance of hybrid and non-hybrid cache designs.

5.2 Cache Designs

The cache designs treat the client memory as a chain of objects. This "usage chain" is nearly an LRU chain. We implemented three client cache designs for use with the simulation model. The cache designs differ in how objects are promoted and how they are selected for eviction. Figure 4 depicts how each cache design moves data into, within, and back out of the cache.

Page Cache

In the page cache design, a page that is fetched from the server is entered into the middle of the usage chain. If a later transaction references the page, then it is promoted to the top of the usage chain. Whenever a page enters the cache, the page at the bottom of the chain is evicted to make room. The client always sends whole pages to the server when committing a modified object.

Object Cache

In the client object cache, as in the page cache, the page received from the server in response to a fetch request is placed in the middle of the usage chain. However, when an object that is cached is referenced, only that object is promoted to the top of the chain. When a page enters the cache, some objects at the bottom of the chain are evicted to make room. The client sends only the modified object to the server in a transaction commit message.

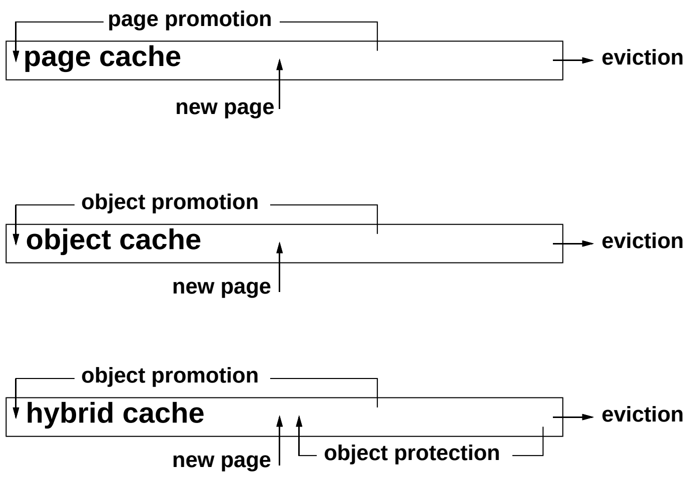

Figure 4: Movement of Data in the Cache Designs

Hybrid Cache

In the hybrid cache, the client enters and promotes objects in the usage chain exactly as in the object cache. However, the eviction rule is biased against evicting an object that belongs to a well-used page. When space is required, the object at the bottom of the chain is examined. If all of its page-companions are in the cache, and if more than half of them are currently in the upper-half of the usage chain, then this object is not evicted. Instead, this object is moved to the middle of the chain. When the client is committing a transaction, the whole page containing the modified object is sent to the server if all of the objects on that page are available in the client cache. Otherwise, only the modified object is supplied to the server.

6 Experiments

In the sections that follow, we focus on aggregate system throughput when the disk drive is fully utilized. We use one workload to illustrate the importance of installation reads. We use another to show the importance of object caching in increasing the client cache hit ratio. Finally, we examine a more realistic workload and show that hybrid caching performs much better than the other methods.

6.1 Dense Hot Workload

To illustrate how installation reads can hurt the performance of an object cache, we chose a workload in which each client directs its hot accesses to 600 hot pages. In this workload, 9 of the 10 objects on each page are hot objects. There are thus 5,400 hot objects packed very densely onto the hot pages.

The table in Figure 5 shows the throughput (tx/sec) of all three designs

Metrics for 50 clients

Cache Design	%hit	tx/sec	I-Cost	I-Disk
Pages	90%	375	—	0%
Objects	85%	228	6.0 ms	21%
Hybrid	90%	373	5.8 ms	<1%

Figure 5: Dense Workload (600 hot pages, all are dense)

when the system is fully loaded. We can see that the page cache performs much better than the object cache. The table also provides the client cache hit ratio (%hit) and installation read costs. The last two columns of the table provide the individual installation read cost (I-Cost) in milliseconds and the aggregate cost (I-Disk) of all installation reads as a percentage of total disk bandwidth.

Of course, the page cache produces only page-commit requests at the server, so there are no installation reads. In contrast, in the pure object system, every writing transaction can potentially cause an installation read. Installation reads consume 21% of the available disk bandwidth in this simulation. The hybrid design converted almost all object commits into page commits, eliminating essentially all installation reads.

Yet, in this example the biggest improvement still comes from the client cache hit ratio. The page and hybrid systems get a 90% hit ratio, but the object system only achieves 85%. The page system is keeping all hot pages in the cache because when a hot access takes place that hits in the cache, the entire page is promoted. The hybrid design also performs well, but because of its eviction rule. The modified eviction rule is making it less likely that hot data will be removed from the cache. Eliminating installation reads freed up 21% of the disk bandwidth, but the increase in the cache hit ratio decreased the disk load due to fetch operations by an even greater amount (about 33%). The result is that the hybrid design nearly equals the page server in throughput. However, the lesson is that for the hybrid design to compete with a page server on a dense hot workload, changes to the promotion rule might be needed.

6.2 Sparse Hot Workload

As another extreme case, we examine a workload where the hot objects are scattered across many pages, one per page. Each client directs its hot accesses to 1,000 hot pages. On each hot page, only a single one of the 10 objects are designated as hot. In this workload, there are one thousand hot objects scattered very sparsely one per page. Object caching should now perform much better than page caching because page caching will not be able to keep the hot pages entirely within the client cache.

The table in Figure 6 contains the performance metrics for 50 clients using the sparse hot workload. The object cache achieves a much higher hit ratio and also higher system throughput. The improvement in throughput is obtained in spite of the installation reads, which now consume 28% of the disk bandwidth.

Note that each installation read consumes only 6 milliseconds of disk time,

Metrics for 50 clients

Cache Design	%hit	tx/sec	I-Cost	I-Disk
Pages	77%	198	—	0%
Objects	89%	262	6.3 ms	28%
Hybrid	89%	296	5.8 ms	18%

Figure 6: Sparse Workload (1000 hot pages, all are sparse)

although the average seek time for this disk is approximately 19 milliseconds. This is due to the effect of the opportunistic log [13]. Without this improvement, the object system would have had much worse performance than the page system.

In the simulation shown here, the hybrid design converted about one third of the object commits to page commits, presumably because many of the colder objects on the hot pages are resident in the cache. It appears that some hot pages are being kept whole. Since the hottest objects fit easily into the object cache, it makes sense that the hybrid eviction rule helps.

6.3 Mixed Workload

Finally, we examine a somewhat more realistic workload where some of the hot pages are densely packed with hot objects and some are not. We now set the total number of hot pages to 1,200, of which 600 pages have 9 hot objects per page and the other 600 pages each have only one hot object.

Metrics for 50 clients

Cache Design	%hit	tx/sec	I-Cost	I-Disk
Pages	67%	142	—	0%
Objects	82%	197	6.0 ms	18%
Hybrid	85%	251	6.0 ms	11%

Figure 7: Mixed Density Workload (1200 hot pages, 600 are dense)

Figure 7 shows the results for the mixed workload. The hybrid cache now performs significantly better than the other designs. It has a much higher hit ratio than the page cache because there are sparsely-hot pages in the workload. It also causes fewer installation reads than the object cache because it keeps densely-hot pages intact in the cache and uses page commits when updating them.

As in the densely hot workload, some of the improvement of the hybrid system relative to the object system is due to an increase in cache hit ratio. However, the important feature of the hybrid cache design is that it biases eviction policy against removing a cold object that has hot page-companions. This policy helps eliminate installation reads for all the modifications that go to densely-hot pages.

It is important to note that the hot objects in this workload are not equally hot. The sparsely-hot pages get just as many total accesses as the densely-hot pages, even though there are 5,400 densely packed hot objects and only 600 sparsely-hot objects. We should consider whether this reflects plausible assumptions about how objects might be clustered onto pages.

We imagine it to be more likely that the hottest objects would be better packed. We plan to simulate such a workload soon, but we can already guess what results we will see. Because most operations will then involve densely packed objects, the hybrid system will save a much larger fraction of the installation reads as compared to a pure object system. Yet, as long as there are some hot objects that are sparse within their pages, the increase in cache hit ratio due to squeezing them into the cache will ensure that the hybrid system performs better than the page system. So, the relative advantage offered by a hybrid cache design on real workloads may be larger than for the mixed workload shown here.

7 Related Work

To put our work in perspective we consider studies addressing the overall architecture choice for persistent object systems, and studies comparing object and page-based client cache designs.

Many persistent object systems use the more traditional page-based architecture where all interaction between clients and servers takes place at the granularity of individual pages [1, 7, 12, 9]. Other systems [5, 10] use object server architectures but do not specifically address the problem of installation reads.

Dewitt et. al. [6] is one of the first studies that investigated the design choices for a persistent object system architecture. The study focused on the question of distributing the functionality of the persistent object system between the client and the server. It measured and compared a page based system and an object system that fetched a single object at a time. Though the functionality of their object server is different from ours, and single object fetching affects the comparison, our work capitalizes on Dewitt's basic findings that the relative performance of page and object caches are very sensitive to how well hot objects are packed on pages and the relative sizes of the client cache and the client working set.

Similarly, Cheng and Hurson [3] demonstrated how an object server architecture can enable more efficient client cache utilization.

Numerous studies [4, 8, 16, 17] have addressed issues related to comparing object- and page-based client cache designs, emphasizing the importance of pointer swizzling costs to the client. Some of these studies considered using hybrid approaches. The important contrast with our work is that our focus is on the impact of client caching decisions on the critical shared resource: the server disk.

We are not aware of any other work that explores a hybrid object and page based client cache design in light of the cost imposed by installation reads on the server disk.

8 Conclusion

Previous studies considered the tradeoffs in the performance of an object and page-based client cache in terms of the cost of in memory data structure manipulation [4, 8, 16, 17] and in terms of recovery cost [16]. In a scalable object system, client cache design has important effects on the disk load at the server [13].

We explored a cache design that takes into account a previously overlooked aspect of the server disk load: installation reads. We proposed that after first optimizing the client cache to reduce the disk load due to fetches, it is then important to concentrate on avoiding unnecessary installation reads.

We designed a hybrid system that permits clients to cache both objects and pages. The cache uses a simple eviction policy to reduce unnecessary installation reads. To investigate the performance of the hybrid system, we built a simulator and compared the throughput of I/O bound systems with object-based caching, page-based caching and hybrid caching. Our results show that when disk I/O is the system performance bottleneck, the hybrid system can outperform both pure object caching and pure page caching.

References

[1] Using the EXODUS Storage Manager V2.0.0. Technical report, Department of Computer Sciences, University of Wisconsin-Madison, January 1982. Technical documentation.

[2] Atul Adya. A distributed commit protocol for optimistic concurrency control. Master's thesis, Massachusetts Institute of Technology, February 1994.

[3] Jia bing R. Cheng and A. R. Hurson. On the performance issues of object-based buffering. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, pages 30–37, 1991.

[4] M. Day. *Managing a Cache of Swizzled Objects and Surrogates.* PhD thesis, miteecs, In preparation.

[5] O. Deux et al. The story of O_2. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[6] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Velez. A study of three alternative workstation-server architectures for object oriented database systems. In *Proceedings of the 16th Conference on Very Large Data Bases*, pages 107–121, Brisbane, Australia, 1990.

[7] M. Hornick and S. Zdonik. *A Shared, Segmented Memory System for an Object-Oriented Database*, pages 273–285. Morgan Kaufmann, 1990.

[8] Antony L. Hosking and J. Eliot B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 288–303, 1993.

[9] Object Design Inc. An Introduction to Object Store, Release 1.0. 1989.

[10] W. Kim et al. Architecture of the orion next-generation database system. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):109–124, June 1989.

[11] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, San Mateo, California, 1993.

[12] D. Maier and J. Stein. Development and implementation of an object-oriented dbms. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[13] James O'Toole and Liuba Shrira. Opportunistic Log: Efficient Reads in a Reliable Object Server. In *Proceedings of the First Conference on Operating Systems Design and Implementation*, 1994.

[14] Chris Ruemmler and John Wilkes. Modelling disks. Technical Report HPL-93-68rev1, Hewlett-Packard Laboratories, December 1993.

[15] M. Seltzer, P. Chen., and J. Ousterhout. Disk scheduling revisited. In *Proceedings of Winter USENIX*, 1990.

[16] Seth J. White and David J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the 18th VLDB Conference*, pages 419–431, 1992.

[17] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992.