Opportunistic Log: Efficient Installation Reads in a Reliable Storage Server

James O'Toole Liuba Shrira

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139 {james,liuba}@lcs.mit.edu

Abstract

In a distributed storage system, client caches managed on the basis of small granularity objects can provide better memory utilization then page-based caches. However, object servers, unlike page servers, must perform additional disk reads. These *installation reads* are required to install modified objects onto their corresponding disk pages. The *opportunistic log* is a new technique that significantly reduces the cost of installation reads. It defers the installation reads, removing them from the modification commit path, and manages a large pool of pending installation reads that can be scheduled efficiently.

Using simulations, we show that the opportunistic log substantially enhances the I/O performance of reliable storage servers. An object server without the opportunistic log requires much better client caching to outperform a page server. With an opportunistic log, only a small client cache improvement suffices.

Our results imply that efficient scheduling of installation reads can substantially improve the performance of large-scale storage systems and therefore introduces a new performance tradeoff between page-based and object-based architectures.

1 Introduction

A distributed storage system provides long-lived data objects accessed by clients over a network. As such systems scale to support many clients they tend to become I/O bound.

A fundamental design decision concerns the granularity of the data exported by the server to the clients. The server can export small-granularity *objects* (units semantically meaningful to the application), or it can export large fixed-size *pages* (the unit of disk transfer). The relative merits of this fundamental design choice are still the subject of debate [2, 3, 4, 7].

Client caches managed on the basis of small granularity objects can provide better memory utilization than page-based caches because pages are likely to include unneeded data that cannot be discarded from the cache [3, 4, 7, 14, 21]. On the other hand, object servers may need to do extra I/O when a modified object needs to be updated on disk. If the containing page is not in the object server cache, this page has to be read back from the disk. The modified object is then installed in the page, and the page can be written out. These extra *installation reads* do not happen in page servers because the whole page with the modified objects is sent back from the client.

We expect many object modifications to require installation reads in large-scale client-server storage systems. There will be many clients using the server, so the pages containing the combined working set of the clients will not fit inside of the server cache memory.

In this paper, we introduce a simple new approach for organizing I/O that substantially enhances the I/O performance of object servers. A standard technique to make updates atomic is to record them in a stable write-ahead log [10]. We keep this log of modifications in memory and defer installation reads from the commit path. This allows the accumulation of a large pool of pending installation reads that can be scheduled efficiently using well-known disk scheduling techniques [13, 25].

How does one evaluate the impact of scheduled installation reads on the overall performance of an object storage system? One way is to study how it affects the performance tradeoffs between object and page-based architectures. The comparison of the I/O performance of the object and page architectures depends crucially on the difference in how well the respective client caches perform. One extreme is if the loads generated by the client caches are exactly the same, in which case the page server has an advantage. At

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.

the other extreme, if the object architecture makes the client cache more effective, this should decrease the server load, giving the advantage to the object server. Between these two extremes there is a break even point where the two systems have equivalent performance.

We construct a simple performance model that focuses on the I/O costs in object and page servers. Using simulations, we study the effect of scheduled installation reads on the break even point. The results show that when server I/O performance is a bottleneck our techniques have a dramatic effect on the competition between page servers and object servers. Without scheduled installation reads, break even requires a substantial improvement in the client cache performance. With our techniques only a small improvement suffices.

Although there exists substantial work on optimizing deferred writes [13, 20, 25], and read-ahead is used to optimize sequential reads, we are the first (to our knowledge) to take advantage of the opportunity to optimize the installation reads that are important in object servers. We think that file systems could also benefit from optimized installation reads when performing updates to file system meta-data and very small files. Conventional file systems do perform installation reads in these and other situations, but because they do not have a log they cannot defer the installation reads and schedule them efficiently.

Previous studies [2, 7] compared the communication and concurrency control cost of page-based and object-based architectures. Our study is the first to identify the fundamental I/O performance tradeoff between the efficiency of object cache and the cost of installation reads. Our results imply that efficient scheduling of installation reads can substantially improve the performance of large-scale object systems, and hence, our techniques affect the performance tradeoffs between page-based and object-based architectures.

In the following sections, we introduce the basic reliable object server design (Section 2) and describe the opportunistic log design (Section 3). We then describe our experimental setup, presenting the experimental server configurations and the simulation model (Section 4). We present simulation results that illustrate the value of our techniques (Section 5). Finally, we discuss related research (Section 6) and our conclusions (Section 7).

2 Reliable Object Servers

This section introduces the basic design of a reliable object server. A reliable object server provides transactional updates to persistent objects. Objects managed by the server are grouped into *pages* on disk. Objects are small, and a single page can contain many objects. Pages are the unit of disk transfer and caching at the server. We assume that objects on disk are updated in place.[1]

For concreteness in this presentation, we assume optimistic concurrency control and in-memory commit. These features of our object server architecture are derived from the Thor [17] persistent object system. Nevertheless, we believe that the new technique we are proposing is independent of these two assumptions.

Fetching

Clients fetch objects to a local client cache as objects are read, update objects in their cache as they are written, and send back updated objects in a commit request to the server. To improve performance, the client prefetches objects. We assume for simplicity that the prefetching unit is the page containing the object that the client is reading.

Committing

The clients and the server together execute a concurrency control and cache consistency protocol that ensures all transactions are serializable and all client caches are "almost" up-to-date. Since we use an optimistic concurrency control scheme, a client transaction may read an out-of-date value, in which case it will be aborted by the server when it sends its commit request. We assume the need to abort is detected at the server using a validation phase that does not require any disk accesses; see Adya [1] for the details.

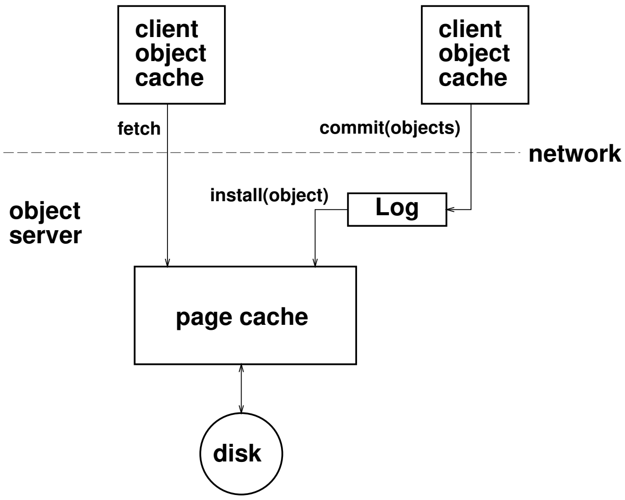


Figure 1: Clients using an object-based server.

When the client sends a commit request, the server records a commit record in a stable log, as shown in Figure 1. The commit record contains the objects modified by the client. A client's transaction is considered committed once its commit entry is present in the log.

[1] An alternative log-structured disk organization for objects would also be possible and is considered by Ghemawat [9].

Installing

After a transaction commits, modifications from the log are applied to the appropriate pages; we call this update process *installation*. Note that an install may require a disk read if the page to be updated is not currently cached. This is an important point, and throughout the paper we are careful to distinguish between disk reads initiated by a client fetch request and reads initiated by an install in the object server. Installs are synchronized with fetch requests, to ensure that fetches retrieve the current committed object state.

Writing

Installations modify pages in the cache, but the dirty pages are not immediately written to disk. We assume that dirty pages are accumulated to permit efficient disk arm scheduling when writing.

3 The Opportunistic Log

We suggest that the server may substantially improve disk I/O performance by opportunistically choosing log entries for installation. We expect opportunistic log processing to reduce the disk I/O requirements of an object server in several ways. The benefits of this method may be so substantial as to justify allocating a large fraction of server primary memory to the log.

When an updated object is to be installed onto its containing page, that page must be read from the disk if it is not already in the server cache. Installation reads can represent a significant source of disk load in an object server (see Section 4). By delaying the installation and performing installation reads opportunistically on the basis of disk head position, we can expect to dramatically reduce the expected cost of an installation read.

There is substantial previous work on delayed processing of disk write operations [13, 25]. Some methods applied to delayed disk writes involve writing pages at new locations [5, 8, 22] and would not work with disk reads. However, standard disk scheduling methods based on head position apply equally well to read operations. In particular, Seltzer et. al. [25] have shown that when a pool of 1000 operations is available, greedy algorithms can reduce the cost of individual operations to a fraction of the normal random-access cost.

3.1 Basic Tradeoffs

In addition to reducing the *cost* of installation reads, we expect to reduce the *number* of installation reads. With enough object modifications waiting in the log, there will sometimes be multiple object modifications that belong in the same page. In this case, by installing these modifications together, only a single installation read will be required. It might even make sense to preferentially choose installation reads that will combine multiple pending log entries in this way.

An installation read will also be avoided when a client fetch operation requires a fetch read of a page for which there exists a pending installation. Note however, that such a fetching read needs to be serviced right away since the client is waiting for it, and cannot benefit as much from efficient disk arm scheduling. This indicates that when a large fraction of the installation reads is avoided by preceding fetch reads, the benefit of scheduling installations is reduced.

In these examples, the installation reads are being avoided when installations occur nearby in time to each other or to a fetch request involving the same disk page. Even without opportunistic log processing, the cache memory in the server would help avoid installation reads when temporal locality is present. However, the object modification log should enlarge the window of opportunity for exploiting temporal locality because the log can store modified objects more compactly than the page cache.

However, there is a tradeoff between allocating memory to the page cache and allocating it to the log. The page cache provides a pool of dirty pages for writing and the log provides a pool of dirty objects that require installation reads. These disk operations will have lower individual costs when a larger pool of candidate operations is available to the scheduler. If too much memory is allocated to one pool, then the disk cost of the operations selected from the other pool will increase.

3.2 Prototype Design

We have described how a large in-memory log of delayed object installations can offer optimization opportunities. We must choose a simple prototype design that can be used in our simulation experiments. The reliable object server must maintain a log of object updates that have not yet been installed onto corresponding pages. Here, for simplicity, we will assume that the server memory is non-volatile.

Thus, the opportunistic server can use a simple scheduler that considers only disk position information to select delayed installations for processing, as shown in Figure 2. The scheduler is free to process and discard the log entries in an arbitrary order. The prototype design will use a greedy scheduler that always selects the delayed installation read that has the shortest positioning time, based on the most recent disk head position. The scheduler uses a branch-and-bound implementation of the shortest positioning time algorithm [25]. We estimate that a good branch-and-bound implementation would make approximate shortest positioning time calculations over thousands of disk operations feasible in current disk controllers.

Without non-volatile memory, the server would need to write the log to a stable disk to commit transactions. The log entries would also be retained in volatile memory for installation processing. The installation processing would

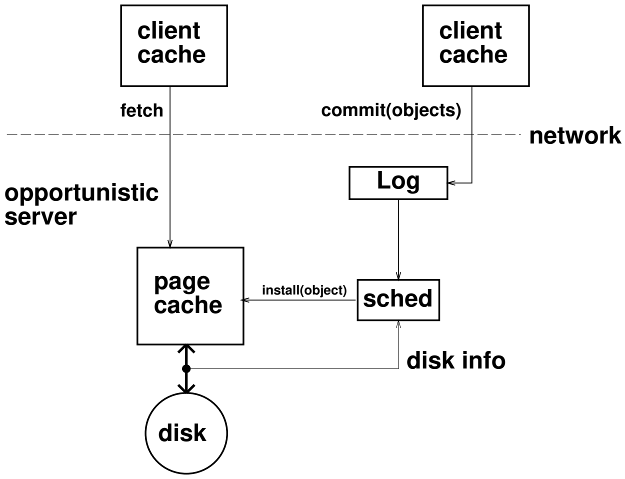


Figure 2: Object Server with Opportunistic Log

be somewhat complicated by the need to bound length of the log, as in a conventional databases with a write-ahead log [10]. In this case the opportunistic log might need to use an age-weighted scheduling algorithm.

4 Experimental Evaluation

To evaluate our technique we studied the effect of the opportunistic log on the performance of a persistent object system. We focus here on the I/O requirements of the persistent object system because these systems tend to become disk bound when they scale to support many clients. Moreover, the current hardware trend towards processor performance outstripping I/O performance appears likely to exacerbate this problem in the foreseeable future.

We performed a series of experiments by simulating a system of clients and a reliable server. The goal of our simulations is not to predict the actual performance of the system precisely but to compare fairly the disk I/O requirements of the system with and without the prototype opportunistic log.

In our simulations we use a simplified server workload with a uniform access pattern and fairly large objects. A workload generated by real client applications would probably exhibit a skewed access pattern and some temporal locality. However, in a large-scale system the server will observe the workloads of many clients mixed together. With enough clients, the server memory will not be large enough to extract much benefit from the locality present in an individual client's request stream. The combined working set of all the clients will not fit in the server cache memory in pages, so the pages fetched by a client will be quickly displaced by other traffic passing through the server cache.

We assume large objects because this is a worst-case scenario for our proposed opportunistic log design. With smaller objects, the in-memory log will store more pending installations per page of memory used.

We use a simplistic client model with a single parameter directly controlling how much work the client can do out of his local cache. The performance of a client cache should be reflected in the number of fetch operations in the request mix observed by the server, so we choose the *fetch ratio* as our workload parameter. The fetch ratio is defined as the ratio of the number of fetch operations generated by the client cache system to the number of transaction commit operations. Traditionally studies like ours have focused on the cache hit ratio, but this measure is not very useful in a system with prefetching.

4.1 Impact of the Opportunistic Log

Our first experiments investigate the impact of the opportunistic log on the performance of a reliable object server. In these experiments, the client behavior is fixed. First, we isolate the two main effects of the opportunistic log: eliminating installation reads through absorption and reducing installation read cost through disk scheduling.

To isolate these two effects we implement three object server configurations and compare their throughput. An *opportunistic server* implements the design presented in 3.2 and therefore benefits from both absorption and disk scheduling. A *basic server* uses the log to defer and schedule disk writes but does not defer or schedule installation reads. This configuration represents an object server with an in-memory log [18]. An *absorbing server* defers installations but does not use intelligent disk scheduling for installation reads. This configuration benefits from absorption only. Section 5.1 presents and analyses these results.

To understand the impact of the opportunistic log on server cache performance, we explore the tradeoffs involved in choosing the log size. We compare the throughput of an opportunistic server with widely varying log sizes and show the tension between more efficient disk arm scheduling for installation reads and for writes. Section 5.2 presents and analyses these results.

4.2 Comparison with Page Server

Our next experiments investigate the impact of the opportunistic log on the overall performance of an object storage system. The opportunistic log optimizes the installation reads that occur in object servers. Since installation reads do not occur in page based servers, optimization of installation reads affects the performance tradeoffs between page-based and object-based systems.

Several studies have indicated that object caching at the client can provide better utilization of client memory [3, 4, 14, 21] than page caching because the cache can hold more objects that are useful to the client. Improved client memory utilization translates into fewer fetch operations, and therefore reduced I/O load at the server. Therefore, object servers might outperform page servers when client object caching reduces their fetch ratio enough to compensate for the added I/O load of installation reads.

To examine how opportunistic I/O affects the competition between page servers and object servers, we implement a page server and compare the throughput of the servers while varying the client fetch ratio. We measure how much the fetch ratio of the object cache must decrease to make the object server outperform the page server. Section 5.3 presents and analyses the results of these experiments.

When considering the choice between page and object servers we are deliberately ignoring the question of whether the server uses page level or object level concurrency control. The choice of concurrency control granularity, though very important in absolute terms, is orthogonal to the question of the relative I/O costs of the object and page server architectures. Using our validation techniques [1], either page or object level concurrency control can be used with both object and page servers.

4.3 Server Designs

We implemented four reliable server designs within the simulation model. The first section below describes the common features of the reliable server designs. The following sections describe how the four servers process transaction commits and object modifications.

Common Server Model

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 maintains a cache of pages that is used to respond to client fetch requests and to buffer transaction modifications that must later be written to disk. The server supports prefetching by sending to the client the whole page containing the requested object. Since we assume the use of non-volatile memory in the server, there is no disk logging cost. However, we would not expect the cost of logging to a dedicated disk to be a limiting factor in system throughput, so ignoring the logging cost is a safe assumption.

When the number of clean pages in the cache drops below the *WriteTrigger* threshold, the server writes one dirty page to the disk. The page is selected using the shortest positioning time algorithm [25].

In the object-based server, the simulator models the transaction log as a collection of modified objects. A portion of the non-volatile primary memory is statically allocated to hold log entries. The *Objects-per-page* parameter defines the number of log entries that can be stored per page of memory allocated. We chose to set *Objects-per-page* to 2 because this seems to be a very conservative choice.

Concurrency control at the server is described by the *ValidationTime* parameter, which defines the cpu time required to validate a transaction. We chose not to model transaction aborts because they are indistinguishable from read-only transactions for our purposes. The object servers differ from each other in how they install object modifications onto pages (see Section 4), but in all cases the installation uses *InstallationTime* cpu time. Table 1 shows the server simulation parameters.

Server	
Database size (full disk)	335,500 pages
Page size	4 Kbytes
Objects-per-page	2
Server memory (10% of database)	33,550 pages
ValidationTime	5 μsecs
InstallationTime	1 msec
WriteTrigger	$<$ 1000 clean pages
IReadTrigger	$<$ 50 empty log entries

Table 1: Server Parameters

Basic Object Server

In the basic object server, only the modified object is included in the transaction commit message sent by the client. The server adds the modified object to the log and sends a confirming message to the client. If the page containing the modified object is not in the cache, then the server immediately initiates a disk read for that page. When the page is available, the modified object is installed onto the page. Then the page is marked dirty and the log entry is removed.

Absorbing Object Server

In the absorbing object server, as before, the modified object is added to the log when a transaction commits. The confirming message is then sent 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. Whenever a page is read from the disk, whether due to a fetch read or an installation read, all modified objects that belong to that page are installed onto it.

Opportunistic Object Server

The opportunistic object server postpones installations as in the absorbing server design. However, when the *IReadTrigger* is invoked the server selects a pending installation from the log opportunistically, using the shortest positioning time algorithm, as described in Section 3.2.

Client Workload Parameters	
Access pattern	Uniform
WriteRatio	20%
FetchRatio	20%
ClientThinkTime	200 msec
Number of clients	20 ... 70

Table 2: Workload Parameters

Page Server

In the page server, the whole page containing the modified object is included in the transaction commit message sent by the client. The server primary memory is non-volatile and the page-based server therefore does not require a log. In systems without non-volatile memory, the page server might have a greater logging cost than the object servers if the page server logs entire pages. However, we would not expect the log disk to be a limiting factor, and in any case modern page-based systems have techniques that allow them to do object-based logging. The page server also uses optimistic concurrency control and in-memory commit. The server stores the page into non-volatile cache memory and marks it dirty. The server then sends a confirming message to the client.

4.4 System Parameters

The simulated system is described by the parameters shown in Table 3. In general, the hardware parameters reflect our assumption that future processor and network speeds will increase substantially relative to disk seek latency. We discuss the simplifications and assumptions in the sections that follow.

Client Workload

The simulated clients each contain a cpu and a local memory for caching either objects or pages depending on the type of server in use. Each client executes a sequence of transactions. Each transaction accesses a single object in the database. If necessary, the client fetches the object from the server. Then the client 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.

The workload parameters are shown in Table 2. They are based loosely on similar parameters in other studies of object servers [2, 7]. We chose not to model the contents of the client cache because only the workload presented to the servers is relevant to our comparisons. Therefore we simply use the *FetchRatio* parameter to control the workload. The *FetchRatio* is defined as the ratio of the number of fetch operations generated by the client cache system to the number of transaction commit operations.

Network	
Message latency	1 msec
Per-message cpu overhead	100 μsecs
Disk (HP97560)	
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

Table 3: System Parameters

Network

The network model provides unlimited bandwidth and a fixed message delivery latency. Each message transmission also incurs a small cpu overhead. We chose to ignore network contention because we do not expect network bandwidth to be the limiting factor in the performance of reliable servers.

An unlimited bandwidth network favors page-based architectures because they send larger commit messages (containing whole pages). Thus, this is a convenient and safe simplification when analyzing the disk I/O requirements of competing server designs.

Disk

The disk modeled by the simulator provides FIFO servicing of requests issued by the server. The disk geometry and other performance characteristics are taken from the HP97560 drive, as described by Wilkes [23]. We chose the HP97560 model because it is simple, accurate, and available. We believe that newer and faster drives will make opportunistic I/O even more important because transfer time is decreasing much faster than seek time [23]. That trend should increase the relative value of locality-based optimizations.

5 Simulation Results

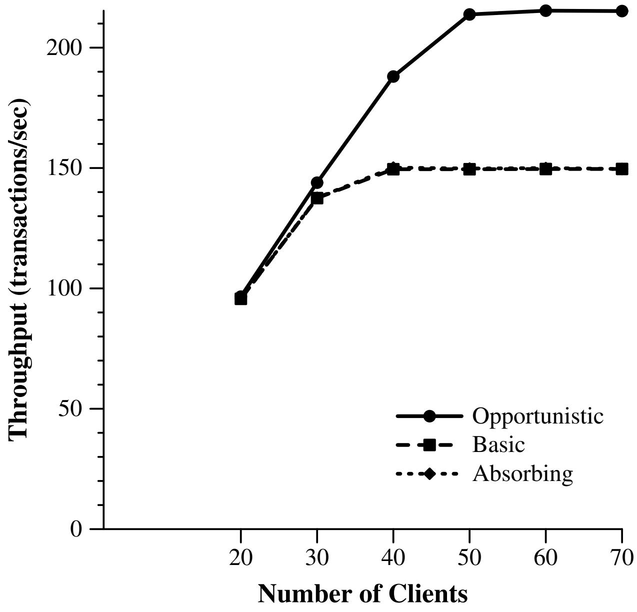
In the sections that follow we first examine the relative performance of the object server designs. Then we explore the tradeoffs involved in choosing the log size. Finally we show how opportunistic I/O affects the competition between page-based and object-based servers.

We use auxiliary metrics to examine the impact of installation reads on server performance. To produce the simulation results we randomly generated several transaction traces for each workload. We sampled the behavior of each server over intervals so large that we observed no significant variation, and then averaged the results.

5.1 Opportunistic I/O

To test the value of opportunistic log processing we compare the performance of the object servers. For the moment, we arbitrarily choose to allocate 1500 pages to the log in the absorbing and opportunistic servers.

Figure 3 shows the throughput of all three servers plotted versus the number of clients in the simulation. Note that with fewer than 30 clients the disk is not saturated and the throughput rises linearly. However, when the system is fully loaded, we can see that the opportunistic server performs significantly better than the other two servers. In this simulation opportunistic scheduling of installation reads increased the maximum server throughput by over 40%.



Metrics for 50 clients, fetch ratio 20%

Object Server	tx/sec	I-Abs.	ICost	IDsk
Basic	150	28%	19.0 ms	41%
Absorbing	150	29%	19.0 ms	40%
Opportunistic	214	29%	4.8 ms	15%

Figure 3: Opportunistic I/O Improves Throughput

The table in Figure 3 provides throughput, installation absorption, and installation read disk costs for all three servers. The installation absorption figure (*I-Abs.*) indicates the percentage of installations that did not require an installation read. The absorbing server did not actually increase absorption very much; 28% of installations are absorbed even in the base system just because of fetch operations and expected server cache hits. Of course, this result reflects our workload: 20% of the modifying transactions fetch and therefore have no installation reads, and 10% of the remaining modifications hit in the server cache.

The installation read cost (*ICost*) is the average number of milliseconds that the disk arm was occupied (seek, rotate, and transfer) by an individual installation read operation. The aggregate cost of all installation reads (*IDsk*) gives the percentage of time that the disk arm was in use by all installation read operations in total. From these numbers we see that because the opportunistic server can choose installation reads from a pool of about 3000 log entries, it is able to reduce the average installation read cost nearly fourfold. Essentially all of the performance improvement shown in Figure 3 is due to this.

5.2 Log Size

We examine the performance of the opportunistic server with widely varying log sizes in Table 4. Here we see the throughput, the installation metrics, and corresponding write metrics. Note that for the log size used in the previous section (1500 pages), the remaining page cache contains over 30,000 pages. The server writes when this cache is almost fully dirty, and the opportunistic choice among 30,000 dirty pages leads to a very attractive write cost of only 3.0 milliseconds.

These measurements show that increasing the log is initially very valuable as the installation read cost drops rapidly. However, later improvements are smaller and require much greater increases in log size. Of course, this is the effect we expected, as described by Seltzer [25].

Metrics for 50 clients, fetch ratio 20%

Log	tx/s	I-Abs.	ICost	IDsk	WCost	WDsk
26	159	28%	16.2ms	37%	3.0ms	9%
60	195	28%	8.2ms	23%	3.0ms	11%
325	207	28%	6.0ms	18%	3.0ms	11%
1500	214	29%	4.8ms	15%	3.0ms	12%
6000	217	33%	3.8ms	11%	3.2ms	12%
12000	217	36%	3.3ms	9%	3.4ms	13%

Table 4: Opportunistic server performance and log size

It is important to note that in this simulation, there was not much increase in total throughput from increasing the log size beyond 1500 pages. There was actually no improvement between the last two rows of the table even though the installation cost dropped. This is explained by the increase in write cost. When the log grew by 6000 pages, the available pool of dirty pages dropped enough to increase

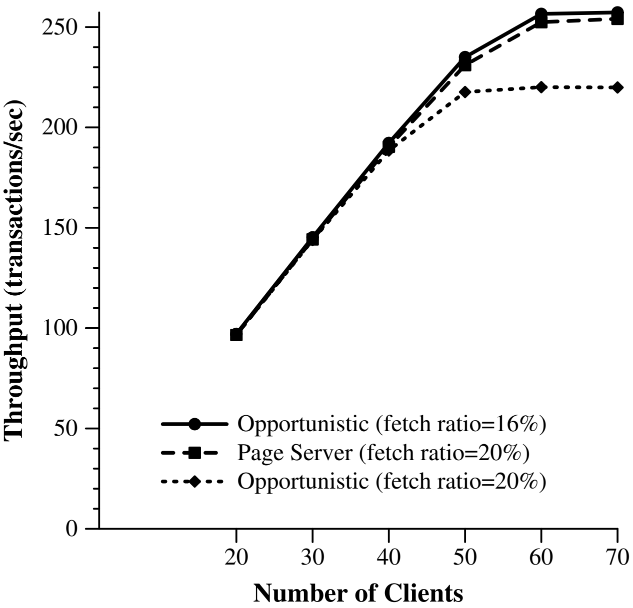


Figure 4: Opportunistic Object Server vs. Page Server

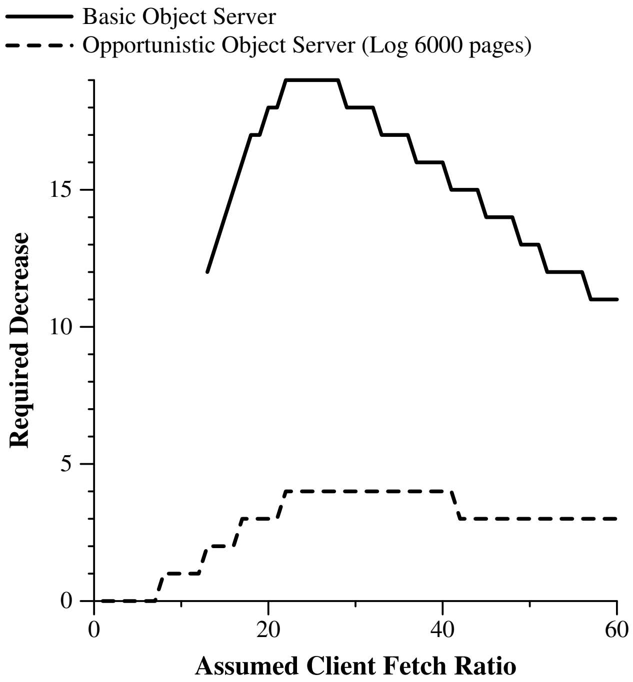


Figure 5: Object Caching vs. Page Caching

the write cost by 0.2ms. Although not shown here, the increase in log size also reduced the server cache hit ratio by 2 percentage points. The combined increase in disk usage due to these two effects effectively canceled out the other improvements.

5.3 Objects vs. Pages

Now that we have identified a good log size for the opportunistic object server (6000 pages), we would like to compare its performance to the page server. As we discussed in Section 4, the two servers face essentially identical concurrency control costs. And although the page server might consume more network bandwidth, this is not included in our simulation model. Therefore, we expect the page server to have higher throughput because it does not need to perform installation reads.

However, if object-based client caching provides better client cache performance, then the object server might be superior because it would have a lower fetch ratio. The difference in performance will depend on the FetchRatio, the WriteRatio, the server memory size, and the actual cost of installations and fetches. We varied the client FetchRatio and found that a 4 percentage point decrease in the fetch ratio was required for the opportunistic object server to surpass the page server. Figure 4 shows the throughput of the opportunistic server (for 20% and 16% fetch ratios) and the page server (20% only).

We see from this example that when the client fetch ratio is 20%, the opportunistic server requires a 4 point fetch ratio decrease to surpass the throughput of the page server. Each fetch read avoided compensates for roughly four installation reads, because installation reads are less costly than fetch reads (compare the smallest and largest values of *ICost* in Table 4). In this configuration, about 16% of all transactions require an installation read, so removing fetches from 4% of all transactions will compensate for the disk cost of installation reads.

To determine the required fetch ratio decrease over a wide range of client cache performance, we simulated the basic object server, the opportunistic object server, and the page server while varying the client fetch ratio between 1% and 60%, using a 50 client workload. Figure 5 plots the decrease in client fetch ratio required by the two object servers in order to equal the throughput of the page server system. Note that below the 20% fetch ratio, the required decrease is much lower because in this range the 50 client workload is not fully loading the disk arm.

The reduced installation cost we saw in Table 4 makes the object server worthwhile if client object caching reduces the fetch ratio on the server by just a few percentage points. In contrast, when installation reads are treated the same as fetch reads, object caching is not advantageous for this workload unless it decreases the client fetch ratios by at least 10–20 percentage points. We believe these results are indicative of the importance of optimizing the asynchronous disk read operations that will be required in object-based systems.

6 Related Work

To put our work in perspective we consider other persistent object systems, systems that use a log for efficient disk access, work on disk scheduling, and studies that compare object and page architectures for persistent object systems.

Many persistent object systems use the more traditional page based architecture [11, 19, 12]. Other systems [6, 15] use object server architectures but have not specifically addressed the problem of installation reads.

There is an enormous literature on centralized and distributed databases (see Jim Gray's book [10] for an excellent survey). Most modern databases use a stable log to store modifications to insure durability of updates. The log is not used as a source of updates to the database during normal operations, since database code modifies the pages directly in memory. Therefore there are no installation reads.

RVM [24] is a portable package that provides durable updates to virtual memory regions. RVM stores the modifications to VM regions in a stable write-ahead log on disk, propagates the modifications from the log to an (additional) persistent copy of the data on disk, and performs installation reads when containing pages are not in memory. However, portability concerns, the assumption that the persistent data fits in primary memory, and the fact that clients fetch data from one disk copy while stable updates are propagated to another disk copy of the database, make RVM performance considerations very different from our work.

Harp [18] is a replicated NFS server that uses a replicated in-memory write-ahead log to store durable modifications to Unix files. The log defers and propagates the modifications efficiently. However, since Harp is implemented on top of the Unix file system, it does not access the disk directly and does not deal with installation reads.

Much earlier effort has been invested in optimizing disk scheduling and so widening the performance gap between unoptimized and optimized disk access. Read-ahead is widely used to optimize sequential reads [16]. Recent studies by Seltzer, Chen and Ousterhout [25] and by Jacobson and Wilkes [13] aggressively take advantage of large memories to efficiently schedule deferred writes. Our techniques capitalize on this disk scheduling work. We are not aware of any other work besides ours that is directly concerned with efficient scheduling of deferred reads.

Several studies have investigated the design choices for persistent object system architecture. Dewitt et. al. [7] focuses on the question of distributing the functionality of a persistent object system between the client and the server. Day [4] studies whether to fetch data on a page or an object basis. Cheng and Hurson [3] demonstrated how object server architecture can enable more efficient client cache utilization. Carey et. al. [2] studied concurrency control issues in object, page and hybrid servers. None of these studies considered installation reads. However, in our recent work [21] we have investigated further the I/O performance tradeoffs related to the granularity of caching in large-scale object storage systems. We explored a hybrid design that selectively caches at the client either objects or pages, trading fetch reads for (optimized) installation reads.

Our work on deferred reads would also relate to work on prefetching, if prefetching reads were treated more like installation reads. The opportunistic log reduces the expected cost of installation reads by taking advantage of the fact that they are asynchronous (non-blocking) reads. It also seems possible that the lower cost of asynchronous reads should be considered when evaluating the cost of prefetching for clients. Prefetch requests could be treated as asynchronous read operations and scheduled opportunistically, as long as the client is not currently waiting for their results.

7 Conclusion

Persistent object systems combine the flexibility of modern programming languages with the efficiency and reliability of modern databases. Modern programming languages support variable size objects of user defined types. Databases support efficient access to large collections of a fixed set of system defined types. We believe persistent object systems will become increasingly important as applications scale in size and complexity.

It is still an open question what is the best architecture for a large-scale persistent object system. Should the server export fixed size pages (the unit of disk transfer) or should it export objects (units semantically meaningful to the application)?

In this paper, we introduced the opportunistic log method for organizing I/O. The opportunistic log substantially enhances the performance of object servers. This new technique uses an in-memory log to provide opportunities for scheduling asynchronous reads.

The opportunistic log is applicable to any disk cache manager that must handle large numbers of partial-block updates, as in file systems. If a block being partially updated is not in the cache, then the block must be read from disk before the update can be processed (by "installing" the updated bytes in the block and writing it back to disk). Seeks and transfers for these "installation reads" make partial block updates expensive. Client caching makes the problem worse by reducing the likelihood that a block to be updated will be resident in the server's cache: Although the needed block would have been read into the cache in order to handle the client's request to read the data, the block is likely to have been flushed from the server cache by the time the client's request to modify the data arrives.

The opportunistic log technique devotes a portion of the server cache memory to holding a pool of unprocessed partial-block update requests. As the pool grows, installation reads needed to process these updates can be scheduled greedily, assuming the disk head can be scheduled effectively by the software.

To investigate the performance advantage provided by the opportunistic log, we built a simplified simulation model that focuses on the I/O costs in page and object servers, and captures the important characteristics of a large-scale object storage system. We investigated and found that the optimal log size should balance the improved cost of scheduled installation reads against the improved cost of scheduled writes. We then studied how our new technique affects the performance payoff from a more efficient client cache. Our results show that the opportunistic log technique affects the break even point between object and page server performance. Without the opportunistic log, an object server must have a much more efficient client cache to outperform a page server. With the opportunistic log a small improvement suffices.

In contrast to previous studies that compared object and page systems, our work is the first to identify and explore the fundamental tradeoff between the efficiency of an object cache and the cost of installation reads. Although there exists substantial work on intelligent disk scheduling of deferred writes, we are the first to take advantage of the opportunity to schedule the large pool of installation *reads* that are important to the performance of object servers.

Modern storage systems have not paid sufficient attention to optimizing installation reads. Our simulation results strongly suggest that this previously overlooked factor should be considered in future storage architectures.

Acknowledgments

We would like to thank the ASPLOS and OSDI referees for their comments. Thanks especially to referee #6, for one paragraph of our conclusion, and to Karsten Schwan, for serving as the shepherd for our paper. We also thank our readers: Brian Bershad, Gregory Ganger, David Gifford, Maurice Herlihy, Barbara Liskov, and John Wroclawski.

References

[1] Atul Adya. A Distributed Commit Protocol for Optimistic Concurrency Control. Master's thesis, Massachusetts Institute of Technology, February 1994.

[2] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proceedings of SIGMOD 1994*, 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, MIT-EECS, In preparation.

[5] W. de Jonge, F. Kaashoek, and W. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of the 14th Symposium on Operating Systems Principles*, Asheville, NC, December 1993. ACM.

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

[7] 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.

[8] R. English and A. Stepanov. Loge: a Self-Organizing Disk Controller. In *Proceedings of Winter USENIX*, 1992.

[9] S. Ghemawat. *Disk Management for Object-Oriented Databases*. PhD thesis, MIT-EECS, In preparation.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

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

[12] Object Design Inc. An Introduction to Object Store, Release 1.0. Burlington, Massachusetts, 1989.

[13] David M. Jacobson and John Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett-Packard Laboratories, February 1991.

[14] Alfans Kemper and Donal Kossman. Dual-Buffering Strategies in Object Bases. In *Proceedings of the 20th Conference on Very-Large Databases*, Santiago, Chile, 1994.

[15] 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.

[16] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3bsd UNIX Operating System*. Addison-Wesley, 1989.

[17] 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.

[18] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[19] 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.

[20] Jeffrey C. Mogul. A Better Update Policy. In *USENIX Summer Conference, Boston*, 1994.

[21] James O'Toole and Liuba Shrira. Hybrid Caching for Large-Scale Object Systems. In *Proceedings of the 6th Workshop on Persistent Object Systems*, Tarascon, France, September 1994. ACM.

[22] M. Rosenblum and J.K. Ousterhout. The Design and Implementation of a Log Structured File System. In *Proc. of the 13th Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991. ACM.

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

[24] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere, and J. Kistler. Lightweight Recoverable Virtual Memory. In *Proc. of the 14th Symposium on Operating Systems Principles*, Asheville, NC, December 1993. ACM.

[25] M. Seltzer, P. Chen., and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of Winter USENIX*, 1990.