Management of Partially Safe Buffers

Sedat Akyürek and Kenneth Salem, *Member, IEEE Computer Society*

***Abstract*—Safe RAM is RAM which has been made as reliable as a disk. We consider the problem of buffer management in partially safe buffers, i.e., buffers which contain both safe RAM and volatile RAM. Buffer management techniques for partially safe buffers explicitly consider the safety of memory in deciding which data to place in the buffer, where to place it, and when to copy updates back to the disk. We present techniques for managing such buffers and study their performance using trace-driven simulations.**

***Index Terms*—Buffer management, disks, operating systems, performance, reliability, safe RAM.**

I. Introduction

SAFE RAM is RAM which has been made as reliable as a disk. Safe RAM can be implemented using batteries or uninterruptible power supplies plus redundancy for error correction and detection. Computing systems, such as transaction processing systems and operating systems, often guarantee the durability of modified data by propagating it to disks. Because it can be updated much more quickly than disks, safe RAM can greatly reduce response times for data updates. Safe RAM can also improve performance by reducing the total number of disk writes that must eventually be performed. Updated objects can be buffered temporarily in safe RAM in case they are updated again. Should this occur, two (or more) disk writes will have been combined into one when the object is finally written to the disk.

The disadvantage of safe RAM is its cost. A recent study [1] found that non-volatile RAM was about five times as expensive as volatile RAM, although this cost differential is likely to drop over time. Given that some cost differential exists, *partially safe* buffers make sense. Partially safe buffers are implemented using a mix of safe RAM and volatile RAM. We note that when speed rather than volatility is considered, a similar situation arises. Since faster memories are more expensive, system designers resort to a memory hierarchy, i.e., a mix of fast, expensive memory and slower, cheaper memory.

A variety of existing systems already implement partially safe buffers or assume their existence. These include both main memory buffers, e.g., in the POSTGRES storage system [17], and controller buffers such as the partially safe buffer implemented in the IBM 3990 storage controller [8].

This paper addresses two issues. First, we consider the management of partially safe buffers, with emphasis on replacement policies. Common (and effective) replacement policies, e.g., least-recently used (LRU), are blind to the safety or volatility of the buffer. For example, the LRU policy might place an updated object in the volatile buffer if that is where the least recently used object happens to reside. In a partially safe buffer, replacement decisions should take the safety of the buffer into account. An example of a policy which takes safety into account is one which makes all read replacements in the volatile buffer and all update replacements in the safe buffer.

We present four candidate replacement policies and evaluate their performance using trace-driven simulations. Our evaluation considers the following factors:

- *Size of the Safe Buffer.* Performance can be expected to improve as the safe buffer size is increased. However, the marginal benefits of additional buffer space will decrease with the buffer size. Furthermore, the buffer size affects different buffer management policies differently.
- *Workload Characteristics.* Workload characteristics, such as read/write ratios, "burstiness" of request arrivals, and request locality, affect the performance of the buffer. Our study considers a variety of workloads with differing characteristics.
- *Update Staging.* When a dirty object is replaced in a buffer it must be written back to the disks, causing a delay. To reduce these delays, dirty objects residing in the safe buffer can be written asynchronously to the disks. This is known as *staging*. Staging can reduce response times if disk utilization is not too high. Staging also affects the relative performance of the replacement policies.

The second goal of this paper is to compare partially safe buffers with volatile buffers. *Write-through* volatile buffers, which must guarantee the safety of every update as it occurs by forwarding it to a disk, will clearly be outperformed by a partially safe buffer. Our study quantifies this performance gap. A more interesting comparison can be made between partially safe buffers and volatile buffers in which unsafe updates can be permitted, at least temporarily. We will refer these buffers as *copy-back buffers.* Copy-back buffers are common in operating systems and database management systems.

Copy-back buffers share many of the advantages of partially safe buffers. Copy-back buffers remove disk I/O from the critical path of update requests, resulting in much faster response times. Furthermore, by buffering unsafe updates in the buffer, it may be possible to combine several updates to the same block or page into one, resulting in improved write throughput. Copy-back buffers also have a disadvantage: Applications must be willing to tolerate some lost work in the event of a failure. For example, updated data in a Unix file system buffer is not flushed to disk immediately. Instead, buffered updates are synchronized (i.e., copied back to the disks)

Manuscript received Dec. 19, 1991; revised Aug. 26, 1993. This work was supported by NSF Grant No. CCR-8908898 and by CESDIS.

S. Akyürek with the Department of Computer Science, University of Maryland, College Park, Maryland 20742.

K. Salem is with the Computer Science Department, University of Waterloo, Waterloo, Ont., N2L 3G1, Canada; e-mail kmsalem@uwaterloo.ca.

IEEECS Log Number C95002.

periodically. A failure may cause updates that occur after the most recent synchronization point to be lost.

In database management systems, buffer managers that permit unsafe updates are said to use a ¬*FORCE* [7] policy. Such buffer managers are often used in conjunction with a separate mechanism, such as after-image (REDO) logging [7] and periodic checkpointing, to guarantee the safety of all updates.[1] The combination of logging, checkpointing, and a ¬FORCE volatile buffer provides advantages similar to those of partially safe buffers: Write response times are fast and the safety of updates is guaranteed. However, a partially safe buffer can provide the guarantee more simply, since no REDO logging or checkpointing is necessary. Furthermore, checkpointing mechanisms introduce a tradeoff between failure recovery time and overhead during normal operation. (More frequent checkpoints reduce recovery time at the expense of additional overhead.) Partially safe buffers require no such tradeoff.

Our performance study compares partially safe buffers and volatile copy-back buffers. Partially safe buffers are the superior alternative if they can provide competitive request response times, since they do not lose updates to failures. In database management systems, partially safe buffers can be viewed as simple alternatives to REDO logging and checkpointing.

A. Related Work

A number of existing or proposed systems already employ safe RAM to improve performance. As we have already noted, the IBM 3990 disk controller [8] is capable of buffering updates in its non-volatile memory. Both volatile and non-volatile buffers are maintained, and updates appear in both. Dirty data in the safe buffer is staged to disk asynchronously when possible. XPRS [18], a POSTGRES-based transaction processing system, buffers frequently updated data blocks in safe RAM to provide fast recovery. The techniques used to manage the safe buffer are not described there. The Phoenix file system [5] maintains an entire file system in safe RAM.

In [3], the use of partially safe buffers in transaction processing systems is discussed. An analytic queueing model is used to estimate the size of the safe buffer required to avoid synchronous disk writes, when asynchronous staging is used. (Synchronous writing is required when the safe buffer becomes filled with dirty data.) However, that work does not consider buffer management, which is the focus of the model in this paper. Our model incorporates the impact of the safe buffer and the buffer management policies on read response times as well as write response times, and is based on trace-driven simulations rather than an analytic model.

Stochastic simulations described in [12] compare several alternative safe RAM configurations in database systems. Safe extended main memory, solid state disk, and safe disk (controller) buffers are considered. That work uses a different safe RAM model than ours. In [12], read-referenced data that is located in the safe extended main memory must be transferred to volatile main memory to be read. Furthermore, direct transfer of data between the safe RAM and the disks is not possible. The focus of that work is on comparing the alternative safe RAM configurations, and not on buffer management.

Two recent studies have used trace-driven simulations to evaluate safe buffers in operating systems. One study [1] was based on traces from the Sprite file system, while the second used traces from Unix file systems [13]. The traces used in [13] are block level traces similar to our own, while the Sprite traces are captured at a higher level. In particular, the Sprite traces do not include requests for file system meta-data, such as i-nodes [9]. Such references account for a substantial fraction of the requests (particularly write requests) in our traces and in those of [13].

Both of these studies used a safe RAM configuration called *write-aside*, which we did not consider in our study. Under this configuration, the safe RAM handles all update requests, but it cannot be used to satisfy read requests. This configuration is similar to that implemented by the IBM 3990 disk controller and the model studied in [3]. The Sprite study also considered a *unified* configuration similar to the one used in this study. Under the unified model, read requests can be satisfied from either the safe buffer or the volatile buffer. In the Sprite study, the unified partially safe buffer was managed using a policy similar[2] to the *LRU Global/Write Purge* policy described here. Other policies were not considered. The Sprite study concluded that the unified configuration provided better overall performance.

Both of these studies showed that small amounts of safe RAM can provide significant reductions in write traffic. This general result corroborates one of our own. Neither of these studies considered the effects of asynchronous staging from the safe buffer, presumably because their principle performance metric was the reduction in write traffic due to the safe buffer. (Staging is a technique for improving response times at the expense of increased traffic behind the cache.) Neither study evaluated policies for managing both read and write requests under the unified partially safe cache configuration. This is the primary focus of our study.

Several techniques have been suggested for improving the disk performance, assuming that updates are temporarily buffered. Safe RAM is ideal for use in combination with these techniques, since buffered updates will not be lost in the event of a failure. Ng [11] suggests that safe RAM can be used to eliminate the write penalty associated with duplexed disks. Buffered writes can also be "piggy-backed" onto read operations [16], allowing the disk to be updated with very little cost. Either of these techniques can be combined easily with the buffer management techniques described in this paper. However, we do not consider such extensions here.

Finally, several proposed memory-resident transaction processing system designs [4], [6], [10] rely on safe memory to commit transaction updates quickly. Small safe memories have also been used to provide fast recovery in the Sprite file system [2]. In those systems, safe memory management is tied closely to the transaction manager or the file system. In contrast, the techniques described in this paper are not tied to the semantics of any particular data manager.

[1] Actually, only the safety of *committed* updates is guaranteed.

[2] The policy in [1] copies data from the safe cache to the volatile cache in some situations. The policies described here do not.

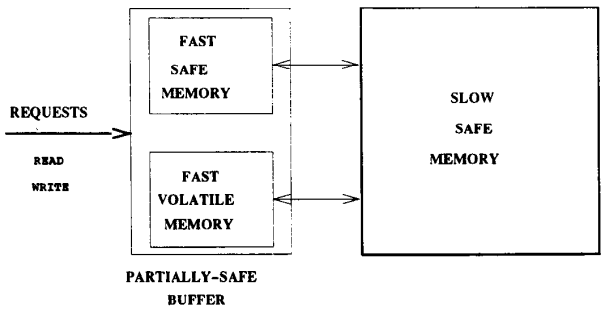


Fig. 1. Storage model.

In the next two sections we present several buffer management techniques for partially safe buffers, our simulation model, and the traces that drive it. A comparison of the buffer management techniques based on the simulation model is presented in Section IV. Section V describes how asynchronous staging can be used to further improve the performance of the partially safe buffer. Finally, in Section VI, we compare partially safe buffers with volatile, copy-back buffers.

II. Managing Partially Safe Buffers

We have used a simple model to study partially safe buffers. The storage system is assumed to manage a collection of fixed-size objects, each with a unique identifier. (Objects may correspond to pages or file blocks in a real system.) Three types of storage are available to hold them. Slow, safe storage (e.g., disks) maintains a copy of every object. A faster buffer also exists, and is divided into safe and volatile parts. Copies of some of the objects reside in the buffer. The model is illustrated in Fig. 1.

The system's load consists of read and write requests. The buffer manager must satisfy these requests by observing the following rules:

- *Read Rule*: When an object is read, a copy of the object should be buffered (either safe or volatile).
- *Update Rule*: When an object is updated, an updated copy should reside in safe memory (either fast or slow). Later we will consider volatile copy-back buffers, which do not observe this rule.

We have considered two binary dimensions along which to classify buffer management techniques, giving rise to four classes of buffer managers. The first dimension defines the buffer manager's behavior in the event of a read miss. The second defines its behavior in the event that an updated object is already in the volatile buffer.

A. Read Miss Policy

According to the read rule, read requests are satisfied immediately if the requested object is in (either part of) the buffer. A *read miss* occurs if the object is not in either buffer, in which case it must be read from the disk. We have considered the following two policies for handling read misses.

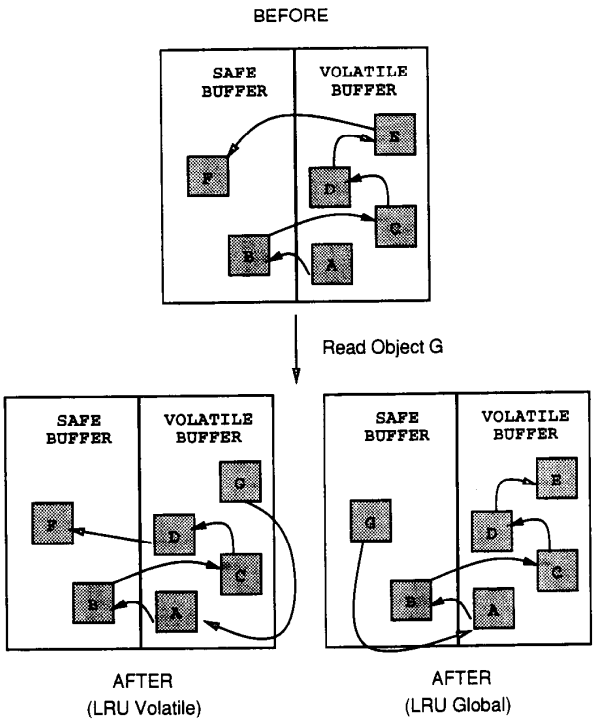


Fig. 2. Read miss policies compared. The figures illustrate the effect of a read miss under the two read miss polices. It is assumed that both the safe and volatile buffers are full, so that the read miss must cause a replacement. The chains of arrows indicate recency of use, with $A \rightarrow B$ indicating that A is more recently used than B. Under the *LRU Volatile* policy, object G replaces object E, since E is the least recently used object in the volatile buffer. Under *LRU Global*, object G replaces object F in the safe buffer instead, since F is the least recently used overall.

- *LRU Volatile.* This policy specifies that missed reads cause a replacement of the least recently used object in the volatile buffer. Of course, if free space is available in the volatile buffer, replacement is not necessary.
- *LRU Global.* This policy specifies that missed reads cause a replacement of the least recently used object in the whole buffer. Under this policy, it is possible that read replacements will be performed in the safe buffer.

Fig. 2 illustrates the behavior of two read miss policies. The *LRU Global* policy is particularly appealing when the volatile portion of the buffer is relatively small, because all read requests need not be channeled through the volatile buffer.

B. Write Allocation Policy

So that the write rule can be satisfied without "writing through" to the disk, updated blocks are always placed in the safe buffer. If necessary, the least recently used object in the safe buffer is replaced to make room. The write allocation policy specifies the behavior of the *volatile* buffer in the event of an update. This policy is necessary in case an object is in the volatile buffer when it is updated. We have considered two write allocation policies.

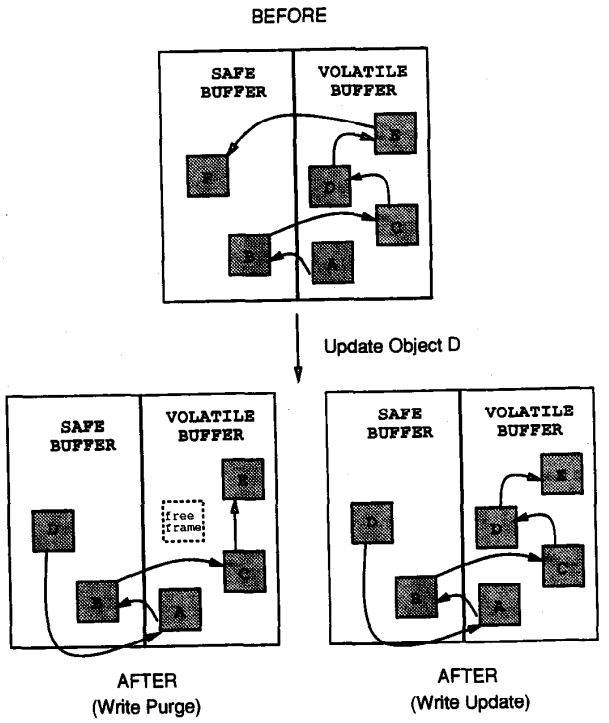


Fig. 3. Write allocation policies compared. The figures illustrate the effect of an update under the two write allocation polices. The chains of arrows indicate recency of use, as in Fig. 2. Under the *Write Purge* policy, the update to D causes it to be removed from the volatile buffer. Object D is placed in the safe buffer, replacing F, the oldest safe object. (We have assumed here that the safe buffer is full.) Under *Write Update*, object D is updated in the volatile buffer and remains there after the update. Note that a subsequent read miss will cause object E to be replaced in the volatile buffer in this case. Under the *Write Purge* policy, E would not have to be replaced, since a free slot is available in the volatile buffer.

- *Write Purge*. This policy specifies that objects are deleted from the volatile buffer when they are updated. The space occupied by the deleted object is marked free, and is available to hold a new object when the next read miss occurs.
- *Write Update*. This policy specifies that objects in the volatile buffer remain there if they are updated. The object is updated in the volatile buffer to reflect its new value. Since updated objects are always placed in the safe buffer as well, this policy may result in two copies of an object being buffered simultaneously, one safe and the other volatile. An object updated in the volatile buffer is not considered to have been "used" by the update. Thus, if the least recently used object in the volatile buffer is updated, it will still be replaced when the next read miss occurs (assuming that the *LRU Volatile* read miss policy is being used).

Fig. 3 illustrates the two write allocation polices. In [15], a third policy, called *Write Allocate*, is discussed. Under this policy, updated objects would be installed in the volatile buffer if they were not already there. (The object would be updated in place, as under the *Write Update* policy, if it was already in the volatile buffer.) This policy does not seem appropriate for the volatile buffer, since updated objects are automatically installed in the safe buffer anyway. We do not consider the *Write Allocate* policy further in this paper.

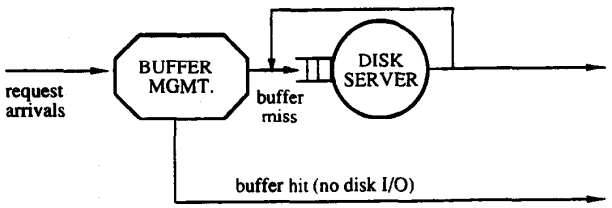


Fig. 4. The simulation model.

TABLE I
BUFFER SIZES ARE SPECIFIED RELATIVE TO THE REFERENCE SET SIZE, WHICH IS TRACE-DEPENDENT. THE REFERENCE SET SIZES FOR EACH TRACE ARE GIVEN IN TABLE II.

Parameter	Default Value
Size of the Safe Buffer	3% of reference set
Size of the Volatile Buffer	7% of reference set
Mean Disk Service Time	25 ms.

III. THE SIMULATION MODEL

We have developed the simulation model illustrated in Fig.4 to study the buffer management techniques described in the previous section. The simulator is driven by traces of read and write requests. Each trace request (described in more detail below) includes a block number, a read/write flag, and an arrival timestamp.

The arrival process for the buffer is determined by the trace request timestamps. At each request's arrival time, the simulator determines which blocks must be moved between the buffer and the disks, according to the read and update rules, the buffer management policy being implemented, and the buffer size. A single request results in as few as zero and as many as two disk operations. One disk operation may be required to bring the requested block into the buffer. A second operation may be required to write a replaced, dirty block back to the disk. If two disk operations result from a single request they are initiated sequentially, as illustrated by the loop in Fig. 4. The response time of a request is the sum of the response times of the disk operations that it generates. Requests that generate no disk operations are defined to have a response time of zero.

Disk service times are assumed to be exponentially distributed, and requests are served in FIFO order. We considered a more elaborate disk model in which seek times were calculated using the block numbers of successive requests and additional assumptions about the disk's geometry. While such a model would probably result in more accurate absolute service times, we felt that it was unlikely to have a strong impact on the *relative* performance of the various buffer managers. For the sake of simplicity, we elected to use the simpler exponential model.

The simulation parameters and their default values are summarized in Table I. The simulator reports a variety of statistics for each run, including counts of I/O operations, mean

TABLE II
REFERENCE TRACE SUMMARY.

Trace Name	Trace Type	Number of Records	Number of Reads	Numbers of Writes	Reference Set Size (RSS)	Num. Records/ RSS
serverA	*post-buffer*	180000	125984 (70%)	54016 (30%)	7139	25.2
clientA	*post-buffer*	8000	3311 (41%)	4689 (59%)	1058	7.6
serverB	*pre-buffer*	370000	308790 (83%)	61210 (17%)	5319	69.6
clientB	*pre-buffer*	22000	18122 (82%)	3878 (18%)	1050	21.0

response times for read and write requests, and utilization, service time, and waiting time at the disk server. It is implemented using the CSIM simulation library [14].

A. The Traces

The traces were gathered from workstations running a customized version of the SunOS 3.2 operating system kernel. The kernel was modified to produce a trace record for each block I/O request (read or write) to file systems residing on the workstation's disk(s). Trace records are deposited in a kernel buffer, from which they can be read by a user-level process using a special "pseudo-device" driver. Requests destined for particular file systems can later be filtered from the traces if desired.

Each trace record includes a unique identifier[3] for the requested data, the size of the request, a read or write tag, and some additional pieces of information. Each record also indicates whether or not the request "hit" the I/O buffer cache on the traced machine.

Because of the way the tracing facility is implemented, each write request in our traces represents a request to flush a block from from the file system buffer to disk, rather than the actual modification of the block in the buffer. In Unix, these block flush requests are often generated by a special synchronization process which periodically[4] flushes dirty data from the buffer. Thus, the block flush request may appear in the trace some time after the block was actually updated in the buffer. Because the synchronization process is periodic, it also means that many of the block flushes occur in periodic bursts in the reference stream. Rather than attempting to guess the actual block update times, we used the traces as is, with the bursty write request pattern, to drive our simulations. Fortunately, the (simulated) safe buffer tends to smooth out the effects of these bursts anyways. The primary effect of this decision is that the write response times (under all of the buffer management policies) are higher than they would otherwise be when the safe buffer is very small. Where this effect is apparent in our simulation results in the next section, we have been careful to point it out.

Two types of traces were used in our experiments. A *pre-buffer* trace includes all I/O requests, including those that hit the traced system's buffer cache. Such a trace is representative of the request streams seen by a main memory buffer, such as a file system's block buffer. The request response times we measure using these traces represent times that an application program observes as it makes requests to a file system with a partially safe buffer.

A *post-buffer* trace includes only those I/O read requests that missed the buffer cache on the traced machine, plus all write requests. These traces are representative of the request streams that might be seen by a storage controller with a partially safe buffer (such as the IBM 3990), since they include only those requests that "fell through" the buffer on the traced machine.[5] The request response times we measure using these traces represent times that the file system observes as it makes requests to an underlying storage device equipped with a partially safe buffer.

Requests in the traces vary in size from 1K bytes to 8K bytes, with the vast majority of the requests being for 8K bytes. For the purposes of these simulations, all requests were treated as 8K requests, i.e., when a small block is requested, the large block that it is a part of is requested instead. Thus, the 8K byte blocks, each of which has a unique identifier, are the database "objects" of our model.

Each trace covers a period of about 12 hours, starting in mid-morning, on a weekday. Traces were taken from two workstations. One of these serves as a network file server. The other is a client workstation with its own private disk. Each of the four traces was recorded on a different day. The traces from the server record references to a single large file system housing primarily shared data and executable files. The client traces trace references to the *local* filesystem on the client workstation. This file system holds primarily user files. We experimented with a *pre-buffer* and a *post-buffer* trace from each of the workstations.

Some of the trace characteristics are are summarized in Table II. The reference set size refers to the total number of objects referenced at least once in the trace. This number is much smaller than the total number of objects stored on the disk, since many objects are not referenced at all. The last column of the table gives a crude indication of the locality present in the request stream. Cumulative read request inter-arrival time distributions for each of the traces are shown in Fig. 5. Inter-arrival times were measured with 20 millisecond resolution, as this was the resolution of the system clocks on the traced machines.

In general, the *post-buffer* traces on both the client and the server have a higher percentage of updates. The *client* traces exhibit less reference locality than those of the server. Finally,

[3] An identifier consists of major and minor device numbers and a block number.

[4] Every 30 seconds.

[5] The client and server workstations from which the traces were recorded each have buffer sizes of approximately 2 megabytes.

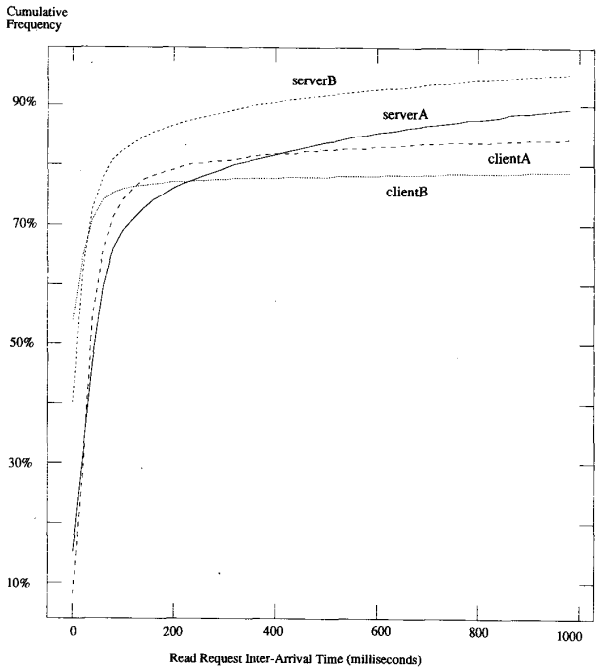


Fig. 5. Cumulative trace request inter-arrival time distributions.

read requests in the client traces have burstier inter-arrival times than those in the server traces. This can can be seen from the flatness of the client trace distributions (especially clientB) in Fig. 5. These curves indicate that almost all inter-arrival times are either very short (less than 100 milliseconds) or very long (greater than one second). The probability of inter-arrival times greater than one second is at least twice as great in the clientB as in either server trace.

IV. Performance of the Buffer Management Policies

We performed several experiments to determine the performance of the buffer manager under various combinations of the read miss and write allocation policies. Of primary interest in our first experiment is the effect of the safe buffer size on performance. In this experiment, the sizes of the safe and volatile buffers were varied, *while keeping the total buffer size (safe plus volatile) fixed* at 10% of the reference set size for each trace. This allows us to distinguish the effects of changes in safe buffer size from the effects of changes in the total buffer size. Reference set sizes for each trace are given in Table II.

Figs. 6 and 7 show the mean write and read response times for each of the four buffer management policies. Fig. 6 shows that a relatively small safe buffer, 3% or 4% of the reference set size, is sufficient to reduce the mean write response time to just a few milliseconds for the server traces. (The high write response times for very small safe buffer sizes is primarily an artifact of the write bursts in our traces.) The reduction was not as significant for the client traces for reasons which we will discuss shortly.

For comparison, Table III gives the mean read and write response times achieved by an all-volatile buffer of the same total size. Write response times for the all-volatile buffer are high because all updates are written through the buffer to the disk to ensure their safety, and because of the write bursts. Read response times are higher for the client traces because read requests are burstier in those traces.

The buffer management policies do not have a very strong effect on write response times. The *LRU Global* policy does result in somewhat lower write response times than *LRU Volatile* for the client traces when the safe buffer is large. This is because of clean data placed in the safe buffer by the *LRU Global* policy. (We discuss this further in Section IV.C.) However, both buffer management and buffer sizes *do* have an impact on read response times, as Fig. 7 illustrates. Furthermore, the behavior we observe is trace dependent. In the following sections, we discuss the impact of the buffer management polices on read response times and the reasons for the trace dependencies.

A. Effect of the Read Miss Policy

The read miss policy's effect is best illustrated by the steep rise in read response time under the *LRU Volatile* policy in Figs. 7(a) and 7(c). As the safe buffer grows and the volatile buffer shrinks, the *LRU Volatile* policy is unable to take advantage of the larger safe buffer. The *LRU Global* policy, on the other hand, simply performs more read replacements in the safe buffer as it grows.

TABLE III
Response Times Comparison of Volatile and Partially Safe Buffers. Partially Safe Buffers are Managed Using the *LRU Volatile/Write Update* Policy. Times are Reported in Milliseconds.

	0% Safe Buffer Size 10% Vol. Buffer Size		3% Safe Buffer Size 7% Vol. Buffe r Size		5% Safe Buffer Size 5% Vol. Buffer Size	
Trace	Read Response Time	Write Response Time	Read Response Time	Write Response Time	Read Response Time	Write Response Time
serverA	10.71	288.89	8.91	2.23	10.14	1.39
clientA	14.20	77.89	14.69	16.08	14.07	15.96
serverB	5.11	415.23	4.03	1.52	4.26	0.20
clientB	32.03	66.43	38.70	8.82	39.51	8.87

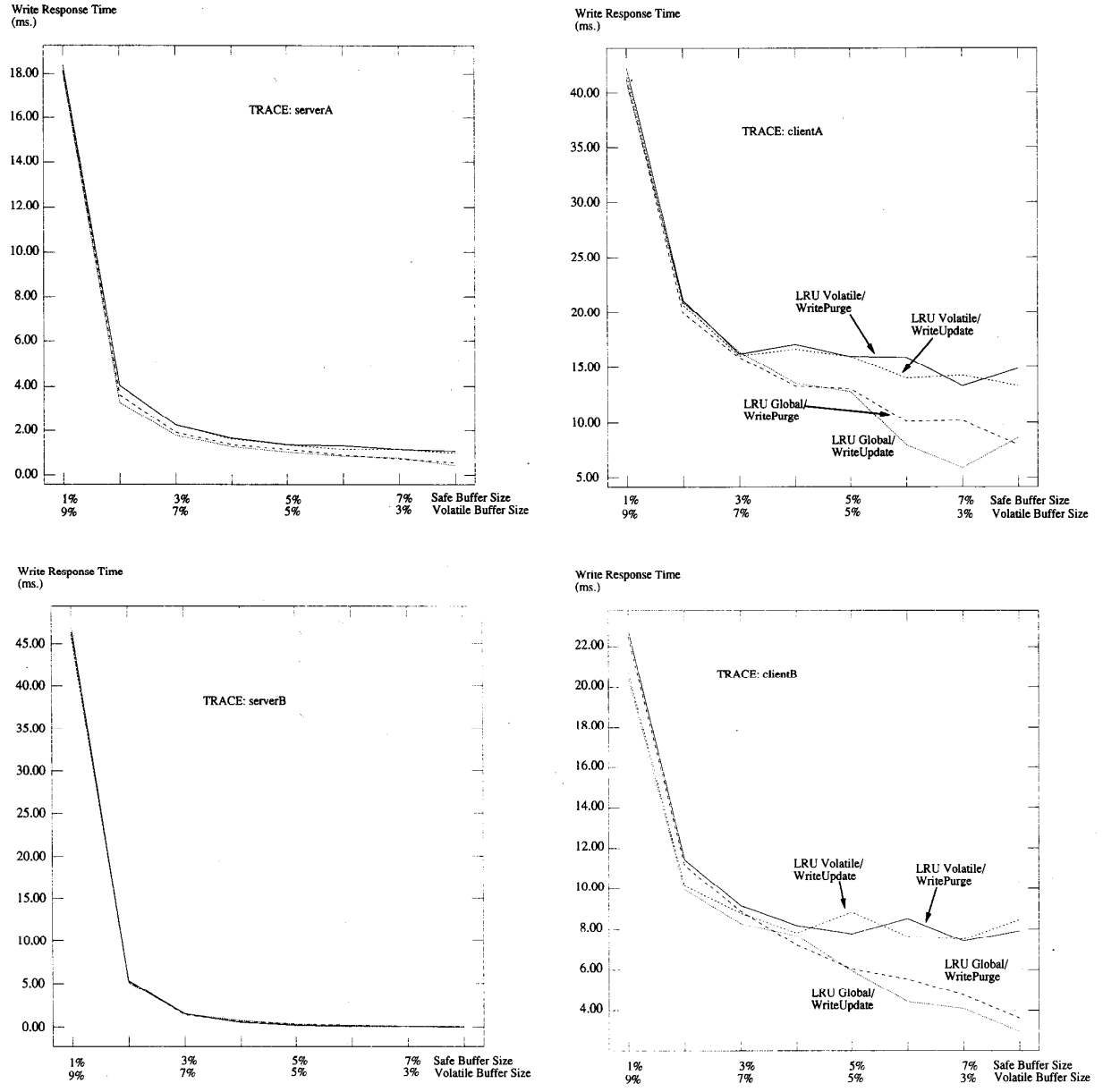


Fig. 6. Mean write response times.

Because of the inflexibility of the *LRU Volatile* policy, a safe buffer that is too large is wasted. This suggests that, in practice, proper selection of the safe buffer size will be more critical under the *LRU Volatile* policy than under *LRU Global*. Too little safe buffer space will result in poor write response times, but excess space will be wasted. In contrast, the *LRU Global* policy can take advantage of additional safe buffer space to reduce read response times. This i+-+-s illustrated more clearly in Fig. 8, which shows read response time as the safe buffer size is increased (for trace "serverA"). The volatile buffer size is fixed at 5% of the reference set size.

B. Effect of the Write Allocation Policy

When the safe buffer is relatively small, the *Write Purge* policy results in a higher read miss ratio than the *Write Update* policy, leading to higher read response times. The effect can be seen most clearly in Figs. 7(b) and 7(c), in which the performance of the two *Write Purge* policies degrades sharply when the safe buffer size falls below 2% of the reference set size.

The *Write Purge* policy performs poorly when the safe buffer is small because of a phenomenon we call *update theft*. When an object residing in the volatile buffer is updated, the *Write Purge* policy causes it to be deleted from the volatile buffer and placed

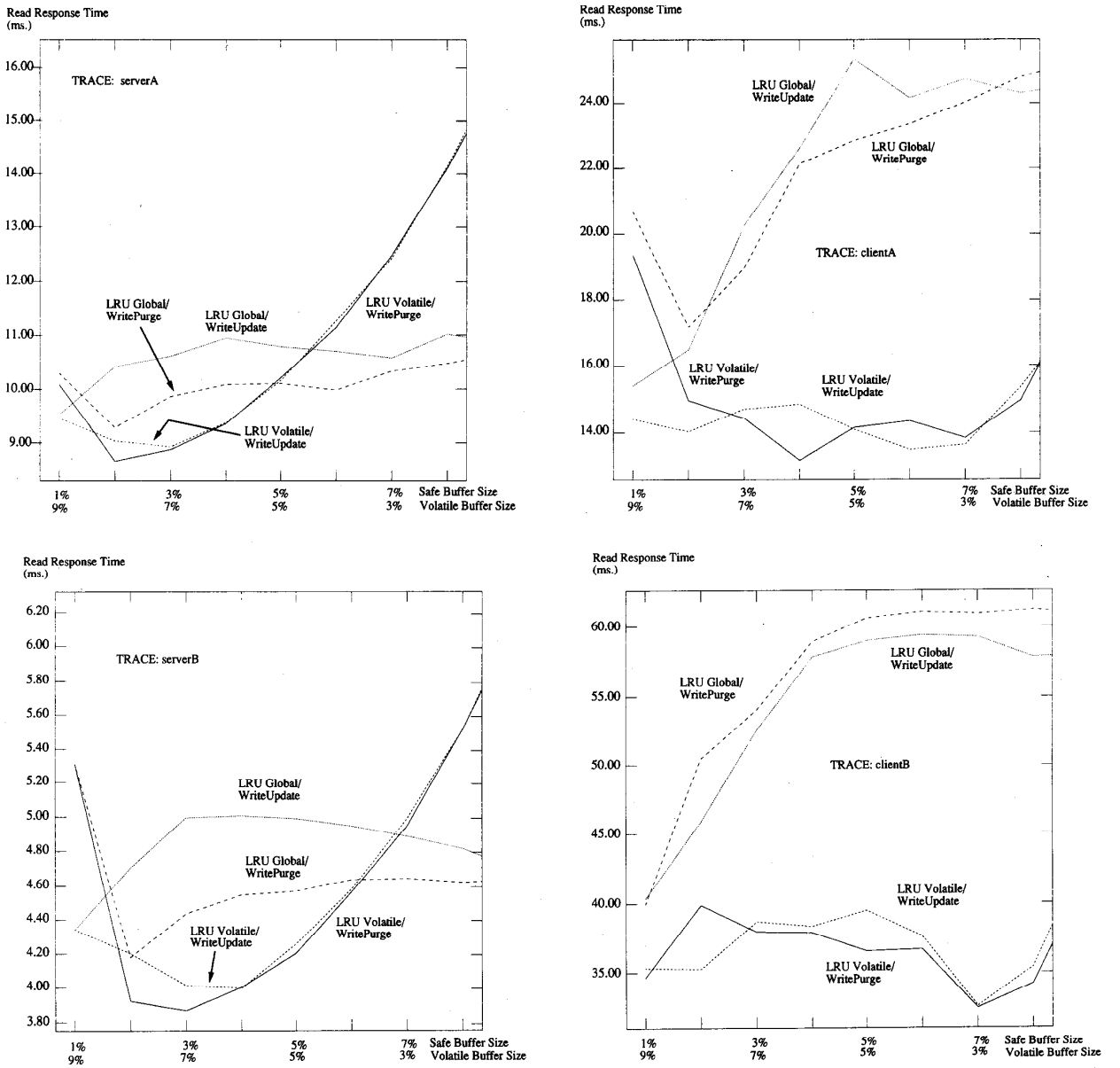


Fig. 7. Mean read response times.

instead in the safe buffer. If the safe buffer is small, this recently read object may be completely eliminated from the buffer much more quickly than it would have been had it remained in the volatile buffer. This problem can be eliminated by making the safe buffer larger. Another possibility is to move data to the volatile buffer when it is replaced in the safe buffer. This latter technique is used in the Sprite studies described in [1].

C. Trace Dependencies

One of the most striking features of Fig. 7 is the difference between the client and server traces. For the traces from the server, the *LRU Global* policy is sometimes superior to *LRU Volatile*. However, *LRU Global* performs poorly on the client traces regardless of the safe buffer size. For both the client and the server, there was little difference in performance between *post-buffer* and *pre-buffer* traces.

Further examination of our results indicates that the buffer management policies do exhibit general patterns of behavior that are independent of the traces. However, certain trace characteristics bring out, or magnify, different aspects of the behaviors. To illustrate, consider Fig. 9, which is identical to Fig. 7, except that synchronous read transfer ratios are shown in-

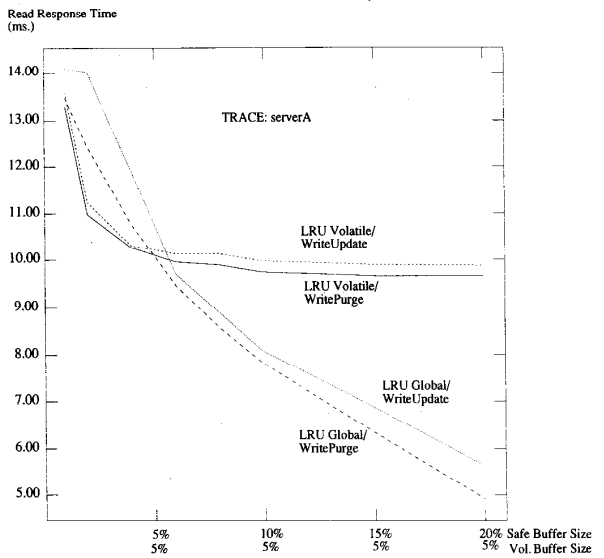


Fig. 8. Read response times vs. safe buffer size. The volatile buffer size is fixed at 5% of the reference set size.

stead of the read response times. The synchronous read transfer ratio describes the average number of disk operations performed per read request. This includes disk reads to bring requested objects into the buffers, as well as the writes needed to copy replaced dirty objects back to the disk.

The effects of the *LRU Volatile* and *Write Purge* polices, which we have already discussed, are clearly evident in both the client and server traces. The *LRU Volatile* policy consistently drives up the transfer ratio as the volatile buffer shrinks. Furthermore, the *Write Purge* policy drives the ratio up when the safe buffer is small. However, this behavior does not translate to consistent response time behavior, as comparison of Figs. 7 and 9 shows.

The reason for the poor read response times of the *LRU Global* policies on the *post-buffer* client trace can be seen from Fig. 9(b). The read transfer ratios for the *LRU Global* polices are elevated because of the very large percentage of write requests in the trace (see Table II). When updates are frequent, read misses become more expensive because replaced objects are more likely to be dirty. (Since updates are channeled through the safe buffer, it is likely that objects replaced from there will be dirty.) In effect, the *LRU Global* policy transfers some of the effort of copying dirty objects back to the disks from write requests to read requests. When write requests are frequent, the impact on read response times is severe. As we will show in the next section, this problem can be reduced significantly by using asynchronous staging.

The *LRU Global* policies have poor read response times on the *pre-buffer* client trace for a different reason. Fig. 10 shows the mean response time (for disk operations) as a function of buffer size for two of the traces. Response times for the *pre-buffer* client trace are several times higher than those of other traces (represented in Fig. 10(a) by the *post-buffer* server trace). Furthermore, use of the *LRU Global* policies increases the already-high response time still further. The high response times are caused by bursts of read requests in the *pre-buffer* client trace, which result in long queues at the disk. This problem is exacerbated by the *LRU Global* policy, since it must sometimes write dirty objects back to the disk to make room for newly read objects. Because of the long queues, these writes have a significant impact on disk response time. (The problem is magnified somewhat because writes, like reads, often arrive in bursts in our traces.) The result is higher response times for read requests, as Fig. 7(b) shows. The long service times also account for the relatively high write response times that we observed for this trace (Fig. 6 and Table III).

In summary, two trace characteristics have been found to have an impact on the relative performance of the buffer management policies. Very bursty read request arrivals tend to drive up response times, regardless of the buffer management policy. However, the problem is exacerbated by the *LRU Global* policy. Since buffers tend to smooth out arrival bursts, this is more likely to be a factor in a main-memory buffer (*pre-buffer* traces) than in a controller buffer (*post-buffer* traces). In addition, a very high percentage of writes in the request stream (such as might be observed at a controller buffer) may be detrimental to read response times when the *LRU Global* read miss policy is used. However, this problem can be alleviated by update staging, as we will show in the next section.

V. Asynchronous Staging

Read and write response times can be reduced by asynchronously staging (copying) dirty pages from the safe buffer to the disk. When a dirty page is staged, it is not removed from the buffer. Its buffer state is simply synchronized with its state on disk. Staging operations are initiated by the buffer manager, allowing many I/O operations to be removed from the critical paths of write (and possibly read) requests. Asynchronous staging from the safe buffer has been suggested in [3] and has been implemented in some systems, including the IBM 3990 storage manager. In the following experiments, we consider the impact of asynchronous staging on response times and show how it affects the performance of the buffer management policies.

Staging can be implemented in a variety of ways. In our simulator, the safe buffer is checked periodically to determine whether it contains any dirty objects. If so, a request to flush the oldest dirty object in the safe buffer is generated. After checking for (and possibly staging) a dirty object, the buffer manager waits C seconds before checking again. (The simulation parameter C is called the minimum staging interval.) When C is set to zero, the buffer manager issues requests to copy objects back to the disk as soon as they are updated. Larger values of C reduce the disk's utilization by reducing the number of write operations that are performed. However, if C is too large, staging may become ineffective.

Staging reduces write response times significantly. Table IV shows how the mean write response time is affected when staging is used. The results show that by staging no faster than one block per second, a safe buffer of size 1% performs at least as well as a safe buffer three times as large that is not staged. When updated pages are staged immediately to the

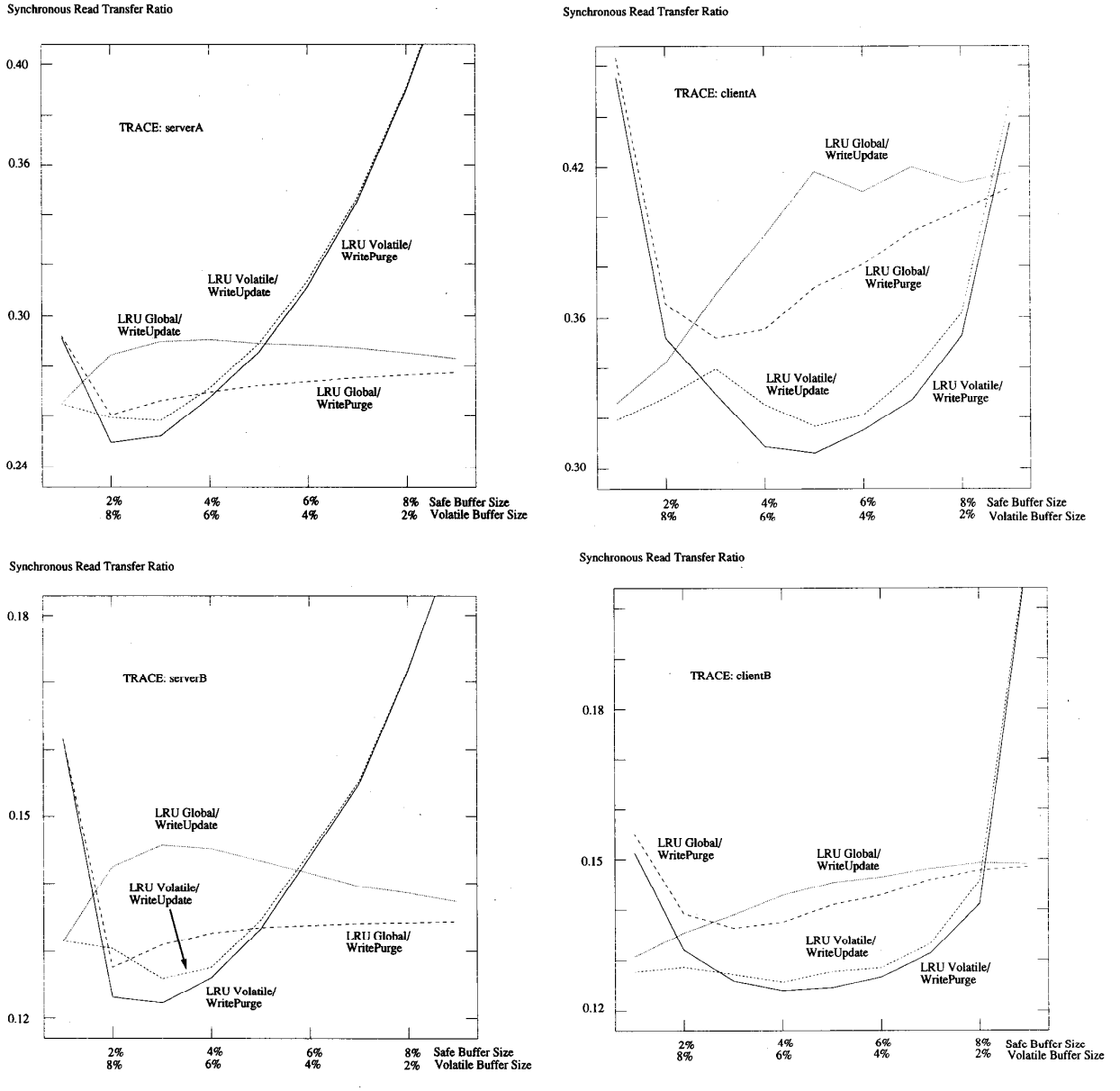


Fig. 9. Synchronous read transfer ratios. The total buffer size is fixed at 10% of the reference set size.

disk ($C = 0$), update time is reduced to near zero for all but the update-intensive trace "clientA". Even for that trace, a 3%-safe buffer coupled with immediate staging reduces the write response time to zero. In qualitative terms, the data in Table IV confirm the predictions of analytic model of [3]: Small, staged safe buffers can reduce write response times to near zero.

It might be expected that asynchronous staging would increase read response times because of contention for the disk. Fig. 11 shows mean read response times for each of the four buffer management algorithms using staging with $C = 0$. This figure should be compared with Fig. 7, which shows read response times without update staging. In absolute terms, the staging did not have a strong impact on read response times. The disk's utilization was low in all of our experiments, and the simulated disk had little difficulty handling the additional traffic caused by staging operations, even when $C = 0$. In the next section we will consider staging to a more heavily utilized disk.

Fig. 11 also shows that staging improves the performance of the *LRU Global* policy relative to the others on the client workstation traces. These were the traces on which the *LRU Global* policy performed poorly without staging (see Fig. 7). Staging is particularly beneficial under the *LRU Global* policy because it cleans dirty objects in the safe buffer. When the

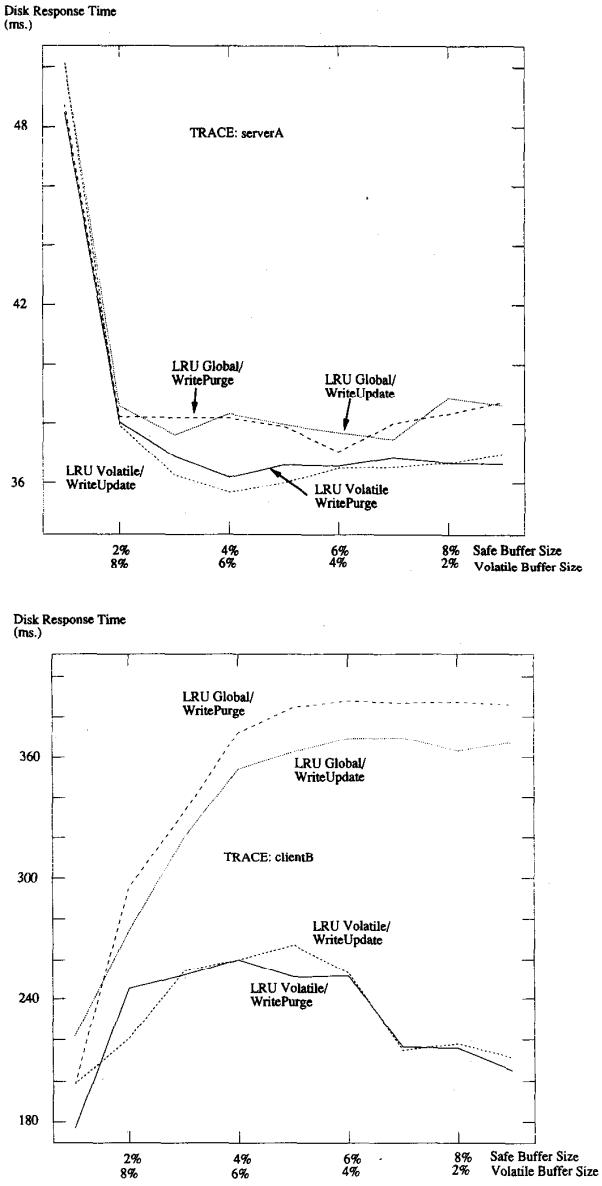


Fig. 10. Average disk response times. The total buffer size is fixed at 10% of the reference set size.

LRU Global policy elects to replace an object in the safe buffer, the object is less likely to be dirty.

VI. Unsafe Updates

For some applications, it is not necessary that all updates be placed into safe memory immediately. Such applications either tolerate some lost updates in the event of a failure, or preserve the durability of their updates using some other mechanism, such as logging.

We have performed experiments to compare a partially safe buffer (with does not lose updates) to a volatile, copy-back buffer, which may. We assume that the copy-back buffer performs periodic synchronization operations to limit the amount of data that might be lost because of a failure. The synchronization operations are initiated at fixed intervals called *synchronization intervals*. Each synchronization operation initiates a batch of disk write operations to copy all dirty objects in volatile memory to the disks.

TABLE IV
MEAN WRITE RESPONSE TIMES USING ASYNCHRONOUS STAGING. RESPONSE TIMES FOR THE PARTIALLY SAFE BUFFERS WERE DETERMINED USING THE *LRU GLOBAL/WRITE UPDATE* BUFFER MANAGEMENT POLICY. HOWEVER, ALL OF THE POLICIES PRODUCED SIMILAR WRITE RESPONSE TIMES. ALL TIMES ARE REPORTED IN MILLISECONDS.

Trace	3%-safe 7%-volatile no staging	1%-safe 9%-volatile $C = 1$	1%-safe 9%-volatile $C = 0$	3%-safe 7%-volatile $C = 0$
serverA	1.79	1.22	0.00	0.00
clientA	16.40	14.92	6.97	0.00
serverB	1.51	1.49	0.02	0.00
clientB	8.27	7.47	0.59	0.00

TABLE V
UNSAFE UPDATE COMPARISON — READ RESPONSE TIMES. THE VOLATILE BUFFER WAS MANAGED USING LRU REPLACEMENT AND A SYNCHRONIZATION INTERVAL OF 10 SECONDS. THE DATA FOR THE PARTIALLY SAFE BUFFER WERE OBTAINED USING THE *LRU VOLATILE/WRITE UPDATE* POLICY, WITH AND WITHOUT ASYNCHRONOUS STAGING. ALL TIMES ARE IN MILLISECONDS.

Trace	0%-safe 10%-vol. write-through	0%-safe 10%-vol. copy-back	3%-safe 7%-vol. no staging	3%-safe 7%-vol. staging ($C = 1$)	3%-safe 7%-vol. staging ($C = 0$)
serverA	10.71	9.13	8.91	9.34	9.41
clientA	14.20	15.25	14.69	14.78	17.09
serverB	5.11	4.09	4.02	4.18	4.15
clientB	32.03	30.11	38.70	32.06	28.47

Tables V and VI compare read and write response times from a volatile copy-back buffer with those of a partially safe buffer. Both buffers provide comparable read response times and very low write response times. Write response times for the volatile buffer are somewhat better than those of the partially safe buffer without staging. When blocks are staged from the safe buffer, write response times are comparable.

The data in Tables V and VI show that volatile copy-back buffers and partially safe buffers can provide comparable buffer performance, i.e., they can be viewed as alternatives. However, partially safe buffers have several advantages. No updates are lost from a partially safe buffer in the event of a failure. This property also eliminates the need to implement periodic synchronization. In transaction processing systems, which use an additional mechanism (such as REDO logging and checkpointing) to guarantee

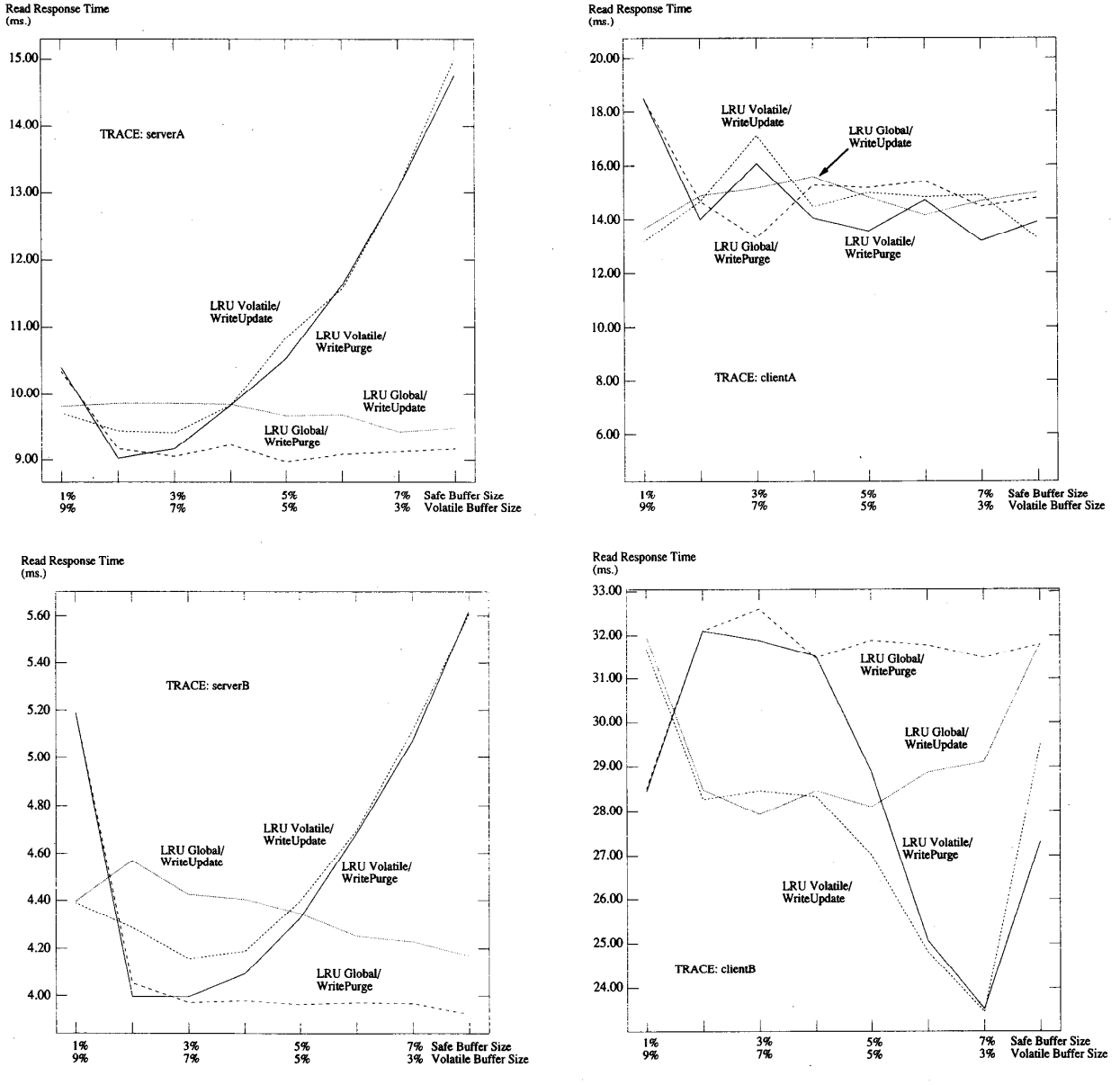


Fig. 11. Mean read response times using asynchronous staging.

the update safety, the need for the additional mechanism is eliminated.[6]

We experimented with other synchronization intervals and found that the copy-back buffer performed best with shorter intervals. (Ten seconds was the shortest interval we tried.) Similarly, the partially safe buffer performed best with short staging intervals. Since the utilization of the disks is low for all of the traces, the extra disk operations generated at the synchronization points do not contribute to significant increases in response times for concurrent read or write operations.

For both the partially safe buffer and the volatile buffer, write response times are near zero. This indicates that synchronous I/O is rarely required to satisfy write requests, i.e., that the blocks replaced by newly written blocks tend to be clean. In a more heavily loaded system, this may not be true.

To test this hypothesis, we increased the request load offered to the simulated system by multiplying the request interarrival times (determined from the traces) by a scaling factor.

[6] A log may be used for other purposes as well, such as to record the commit decisions for each transaction. However, such a log can be much more compact than a REDO log [18].

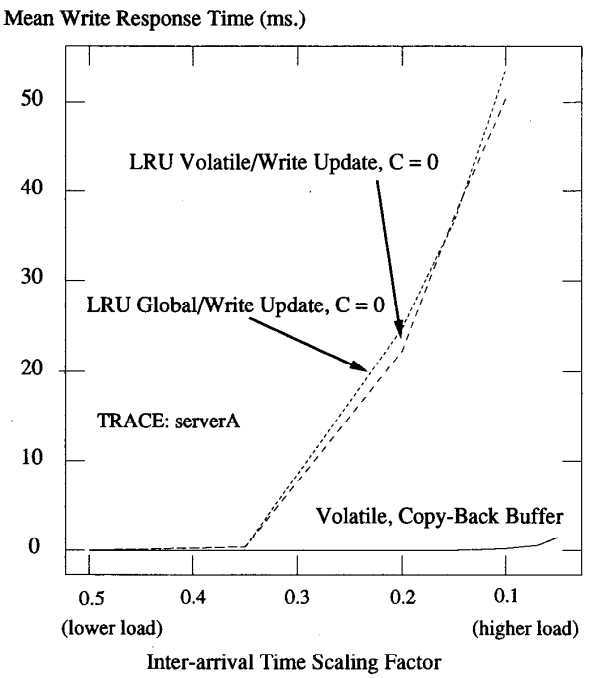
TABLE VI
UNSAFE UPDATE COMPARISON — WRITE RESPONSE TIMES. THE VOLATILE BUFFER WAS MANAGED USING LRU REPLACEMENT AND A SYNCHRONIZATION INTERVAL OF 10 SECONDS. THE DATA FOR THE PARTIALLY SAFE BUFFER WERE OBTAINED USING THE *LRU VOLATILE/WRITE UPDATE* POLICY, WITH AND WITHOUT ASYNCHRONOUS STAGING. NOTE THAT THE HIGH WRITE RESPONSE TIMES FOR THE WRITE-THROUGH ALL-VOLATILE BUFFER ARE AN ARTIFACT OF THE BURSTS OF WRITE REQUESTS IN THE REFERENCE TRACES. ALL TIMES ARE IN MILLISECONDS.

Trace	0%-safe 10%-vol. write-through	0%-safe 10%-vol. copy-back	3%-safe 7%-vol. no staging	3%-safe 7%-vol. staging ($C = 1$)	3%-safe 7%-vol. staging ($C = 0$)
serverA	288.89	0.00	2.24	0.68	0.00
clientA	77.89	0.01	16.08	6.58	0.00
serverB	415.23	0.00	1.52	0.00	0.00
clientB	66.43	1.31	8.82	4.93	0.00

For example, by choosing a scaling factor of 0.5, we divide the actual traced interarrival times in half before supplying them to the simulator. Disk utilization increases from about 8%[7] (for trace "serverA") without scaling to over 60% with a scaling factor of 0.05.[8] The scaled traces provide a somewhat more artificial workload than the unscaled ones. However, scaling is a simple way to get a rough idea of performance under higher utilizations in the absence actual traces of heavier workloads.

Fig. 12 shows the mean write response time as a function of the scaling factor (the offered load) for the two server traces. (Similar behavior was observed for the client traces.) To produce these traces, the synchronization interval was fixed at 10 seconds for the volatile buffer, and the staging interval set to $C = 0$ (immediate staging) in the partially safe buffer. In practice, the selection of an optimal staging or synchronization interval becomes a complex problem at higher loads. (For low loads, short intervals are almost always best.)

The figures show that as the offered load increases, the volatile, copy-back buffer can maintain low write response times at higher loads than the partially safe buffer. The reason for this is that the partially safe buffer channels all update requests through its safe buffer (3% of the reference set size), whereas updates can be buffered anywhere in the volatile buffer (10% of the reference set size). At high loads, the smaller safe buffer tends to fill up with unstaged updates. When this occurs, update requests are likely to encounter a delay while a replacement is performed in the safe buffer. Of course, the improved performance of the volatile buffer at high loads comes at the expense of additional data loss in the event of a failure, since many unsafe updates reside in the volatile buffer.



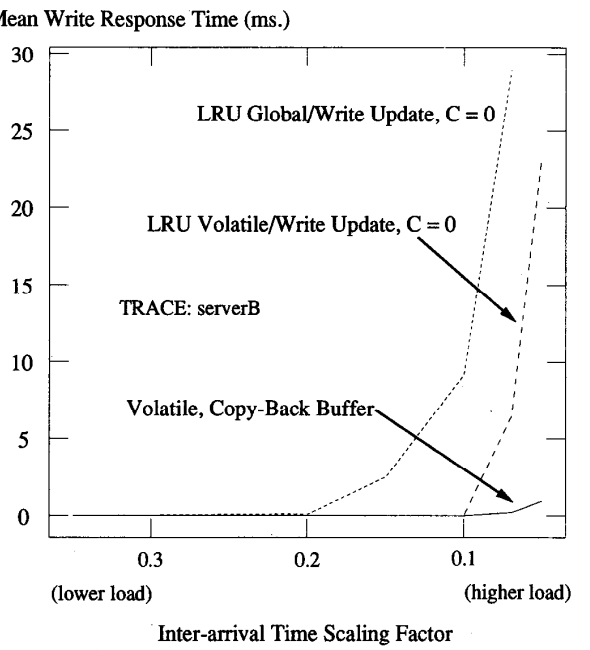


Fig. 12. Mean write response times vs. inter-arrival time scaling factor.

VII. DISCUSSION AND CONCLUSION

We have considered the problem of managing partially safe buffers. Our experiments support a number of conclusions about their use:

- Only a small safe buffer is necessary. For our traces, a few megabytes was always sufficient without staging. If staging is used, even smaller safe buffers will still provide good performance.

[7] Even at low utilizations, requests experience significant average waiting times at the disk server because of the bursty arrival rate.

[8] Utilization does increase at the same rate as the scaling factor because many requests have interarrival times of zero.

- If the *LRU Volatile* technique is used, excess space in the safe buffer will be wasted. The *LRU Global* policy is more flexible and can take advantage of the additional safe buffer space to reduce read response times.
- When the read reference pattern is very bursty, the *LRU Global* policy may result in poor read response times. The same is true for very update-intensive workloads. The *LRU Global* policy exacerbates the problem of read request bursts when it elects to replace dirty objects in the safe buffer, creating additional work for the disks at the wrong time. These effects can be reduced by asynchronously staging dirty objects to the disks.
- The *Write Update* policy is preferred to the *Write Purge* policy because of the poor performance of *Write Purge* when the safe buffer is very small. The small buffer case is important because small buffers are usually sufficient to eliminate most or all of the write response time. For larger safe buffers, the distinction is not significant.
- Asynchronous staging of dirty objects from the safe buffer reduces already-low write response times even further. Staging also improves read-response times when the *LRU Global* policy is used. For workloads such as ours, it is best to perform staging operations without delay. If the disks are more heavily utilized, it may be appropriate to use less bandwidth for staging. Alternatively, staging operations could be performed at lower priority than synchronous (request-initiated) disk operations.
- For lightly loaded disks, the performance of a partially safe buffer is comparable to that of an all-volatile copy-back buffer. By using the partially safe buffer, the need for periodic synchronization of the buffer is eliminated, and no updates will be lost in the event of a failure.

One extension of this work is its application to transaction processing systems, or other systems that use logging to guarantee the durability of updates. If write response times can be made sufficiently small by introducing safe buffers, then one of the principal motivations for logging will have been removed. (One version of this idea is currently being used in the POSTGRES storage system [17].) Aside from the elimination of the complexities of logging, an advantage of the safe-buffer approach is that failure recovery is very fast, since there is no need to reconstruct the state of the database from the log.

It may be possible to further enhance the performance of partially safe buffers by taking advantage of buffered updates to reduce the cost of disk update operations. Piggy-backed updates, as suggested in [18], can be used, or the buffer manager can attempt to delay updates until the disk is idle. We expect that such enhancements would be most beneficial when the disk is heavily loaded, which was not the case in our study.

REFERENCES

[1] M. Baker, S. Asami, E. Deprit, and J. Ousterhout, "Non-volatile Memory for fast reliable file systems," *Proc. Int'l Conf. Architectural Support Programming Languages and Operating Systems*, pp. 10-22, Oct. 1992.

[2] M. Baker and M. Sullivan, "The recovery box: Using fast recovery to provide high availability in the Unix environment," *Proc. Usenix Technical Conf.*, pp. 31-44, June 1992.

[3] G. Copeland, R. Krishnamurty, and M. Smith, "The case for safe RAM," *Proc. 15th VLDB Conf.*, pp. 327-336, Amsterdam, 1989.

[4] M.H. Eich, "A classification and comparison of main memory database recovery techniques," *Proc. Int'l Conf. Data Eng.*, IEEE, pp. 332-339, Feb. 1987.

[5] J. Gait, "Phoenix: A safe in-memory file system," *Comm. ACM*, vol. 33, no. 1, pp. 81-86, Jan. 1990.

[6] H. Garcia-Molina and K. Salem, "High performance transaction processing with memory-resident data," *Proc. Int'l Symp. High Performance Computer Systems*, Paris, Dec. 1987, North-Holland, 1988.

[7] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, no. 4, pp. 287-317, Dec. 1983.

[8] *IBM 3990 Storage Control Introduction*, IBM, manual no. GA37-0098-0, 1987.

[9] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman, *Design and Implementation of the 4.3BSD Unix Operating System*, Addison-Wesley, 1989.

[10] T.J. Lehman and M.J. Carey, "A recovery algorithm for a high-performance memory-resident database system," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 104-117, San Francisco, 1987.

[11] S.W. Ng, "Improving disk performance via latency reduction," *IEEE Trans. Computers*, vol. 40, no. 1, pp. 22-30, Jan. 1991.

[12] E. Rahm, "Performance evaluation of extended storage architectures for transaction processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 308-317, 1992.

[13] C. Ruemmler and J. Wilkes, "Unix disk access patterns," *Usenix Conf. Proc.*, pp. 405-420, Jan. 1993.

[14] H. Schwetman, "CSIM Reference Manual (Rev. 14)," MCC Technical Report No. ACT-ST-252-87, MCC, Austin, Texas, Mar. 1990.

[15] A.J. Smith, "Disk cache — Miss ratio analysis and design considerations," *ACM Trans. Computer Systems*, vol. 3, no. 3, pp. 161-203, Aug. 1985.

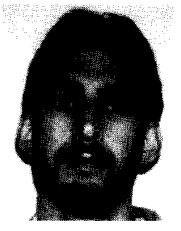
[16] J.A. Solworth and C.U. Orji, "Write-only disk caches," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 123-132, May 1990.

[17] M. Stonebraker, "Design of the POSTGRES storage sysytem," *Proc. 13th VLDB Conf.*, pp. 289-300, Brighton, UK, 1987.

[18] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "Design of XPRS," *Proc. 14th VLDB Conf.*, pp. 318-330, Los Angeles, 1988.

Sedat Akyürek received the BS degree in computer engineering from the Middle East Technical University, Ankara, Turkey, in 1988 and the MS degree in computer science from the University of Maryland, College Park, in 1991, and is currently a PhD candidate in the Department of Computer Science at the University of Maryland, College Park.

Akyürek's research interests include disk and I/O systems, operating systems, databases, and distributed systems. He is currently working on adaptive data storage techniques for disk systems.



Kenneth Salem received the BS degree in electrical engineering and applied mathematics from Carnegie Mellon University in 1983 and the PhD degree in computer science from Princeton University in 1989.

Dr. Salem is an assistant professor in the Department of Computer Science at the University of Maryland, College Park. His research interests include database and operating systems, and transaction processing. He is a member of the ACM and the IEEE Computer Society.