Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
The Journal of Systems and Software 68 (2003) 85–102
The Journal of Systems and Software
www.elsevier.com/locate/jss

Techniques for efficiently allocating persistent storage

Arun Iyengar *, Shudong Jin [1], Jim Challenger

IBM Research, T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA

Received 14 July 2001; received in revised form 4 March 2002; accepted 29 March 2002

Abstract

Efficient disk storage is a crucial component for many applications. The commonly used method of storing data on disk using file systems or databases incurs significant overhead which can be a problem for applications which need to frequently access and update a large number of objects. This paper presents efficient algorithms for managing persistent storage which usually only require a single seek for allocations and deallocations and allow the state of the system to be fully recoverable in the event of a failure. We have developed a portable implementation of our algorithms in Java. Results in this paper demonstrate the superiority of our approach over file systems and databases for Web-related workloads. Our system has been a crucial component for persistently storing data at a number of highly accessed Web sites. We describe our experiences from a large real deployment of our system.

1. Introduction

Efficient disk storage is critically important for performance. Conventional methods for storing data persistently include file systems and databases. These methods incur significant overhead which can be a problem for applications which need to frequently access and update a large number of objects. Our work has been motivated by the need to store dynamic data persistently for highly accessed Web sites; file systems and databases offered insufficient performance. File systems and databases also offer insufficient performance for storing Web proxy cache data. Proxy cache data are updated at high rates, and the overhead for file creation and deletion if cached URL's are stored in separate files is significant.

This paper presents algorithms for efficiently storing data on disk. The algorithms preserve the state of the storage manager after failures. They also minimize the number of disk accesses. These algorithms can be used in a standalone storage manager or could be incorporated into a database or file system. We have implemented our algorithms in order to persistently store Web data created by a Web content delivery system. Our algorithms are quite general and can be used for a wide variety of other applications as well.

The storage algorithms we use make use of free lists containing free storage blocks. Main memory storage allocation algorithms also utilize free lists. However, straightforward adaptation of main memory storage allocation algorithms for disk storage generally results in too many disk accesses. It is necessary to resort to other algorithms in order to reduce disk accesses. Disk storage allocation algorithms should also be robust. In the event of a system failure, the disk storage allocator should continue to function with minimal loss of information.

Fast start from a cold state is also important. Naive implementations can result in considerable delays for bringing up a system. Better performance is achieved by storing important data structures on disk before a shut down which can be read in with few disk seeks when the system is restarted. Fast start is also desirable after a system failure.

The storage algorithms we have developed achieve fast steady state performance by minimizing the number of disk seeks during allocations and deallocations. Our algorithms also result in fast start from a cold state after a shut down or a system failure. We have developed a portable industrial strength implementation of our algorithms in Java which has been successfully deployed for storing data at highly accessed Web sites.

* Corresponding author. Tel.: +1-914-784-6468; fax: +1-914-784-7455.
E-mail address: aruni@us.ibm.com (A. Iyengar).
[1] Present address: Department of Computer Science, Boston University, Boston, MA 02215, USA.

0164-1212/$ - see front matter © 2002 Elsevier Inc. All rights reserved.
doi:10.1016/S0164-1212(02)00141-3

1.1. Related work

A number of papers in the literature have examined the effect of disk I/O overhead on Web performance. However, we are not aware of any papers that describe how to manage disk storage in the detail which ours does and that also describe a successful deployment of a system similar to ours at a major Web site. Several studies suggest that disk I/O overhead can be a significant percentage of the overall overhead on Web servers and proxy servers (Kant and Mohapatra, 2000; Mogul, 1999; Maltzahn et al., 1997; Rousskov and Soloviev, 1998). Rousskov and Soloviev (1998) observed that disk delay contributes 30% towards total hit response time. Mogul (1999) suggests that disk I/O overhead of proxy disk caching turns out to be much higher than the latency improvement from cache hits. Kant and Mohapatra (2000) point out that, with network bandwidth increasing much faster than server capacity, more and more bottlenecks will be observed on the server side, among which disk I/O issues arise from the management of large amounts of content. To reduce disk I/O overhead, Soloviev and Yahin (1998) suggest that proxies use several disks to balance the load and each disk have several partitions to avoid long seeks. Maltzahn et al. (1999) propose two methods to reduce the overhead for proxies: the use of a single file to store multiple small objects in order to reduce file operation overhead, and the use of the same directory to store objects from the same server in order to preserve spatial locality. Markatos et al. (1999) propose a similar method, which stores objects of similar sizes in the same directory (called BUDDY). They further develop an efficient method for disk writes (called STREAM) which writes data continuously on the disks. STREAM works in a way similar to log-structured file systems (Rosenblum and Ousterhout, 1992; Matthews et al., 1997; Seltzer et al., 1993). Log-structured file systems write modifications to disk sequentially in a log in order to speed up writing and error recovery. Significant performance improvement results from the fact that disk writes contribute significantly to the workload on caching proxies. Shriver et al. (2001) described the implementation of a lightweight file system library called Hummingbird. Its design is influenced by caching proxy workload characteristics such as file sizes, read/write ratio, and reference locality. Hummingbird packs files into large, fix-sized clusters, which are the unit of disk access. Therefore, disk seek and interrupt processing overhead is reduced.

1.2. Paper outline

The remainder of this paper is structured as follows. Section 2 describes disk storage allocation algorithms we have developed and implemented. In Section 3, we describe the results of extensive performance evaluations on different system platforms to compare our approach with storage management using file systems and databases. We also show the impact of workload characteristics on the performance of our system in Section 3.3. In Section 4, we describe our experiences from deploying our storage system at a major Web site. Finally, Section 5 summarizes our main results and describes future plans for enhancing this work.

2. Disk storage allocation algorithms

Our algorithms can be implemented for allocating storage over raw disk. Alternatively, they can be implemented for allocating blocks stored within a single random access file. The latter implementation choice is simpler, more portable, and still results in good performance. It is often more efficient than storing each block in a different file because file creation and deletion are avoided. It also avoids the proliferation of large numbers of files.

Our disk storage allocation algorithms typically maintain free lists of blocks in main memory in order to allow fast searches. It is not necessary to explicitly store free lists on disk. Instead, enough information on disk can be maintained to allow free lists to be reconstructed in the event of a system failure.

Storage blocks have headers associated with them. The header indicates the size of the block as well as its allocation status (AS). Throughout this paper, we use the convention that positive block sizes indicate allocated blocks while negative block sizes indicate free blocks.

We now describe our storage allocation algorithms. The first three algorithms are general methods for managing storage blocks. The fourth algorithm, persistent multiple free list fit (PMFLF), is an algorithm which segregates free blocks by size for faster allocation. PMFLF can use any of the first three algorithms.

2.1. Storage management method I: the TS method

In this approach, in-memory free lists are maintained for fast allocation and deallocation. These in-memory free lists are searched in order to locate blocks of the right size. Each block maintains the header information on disk shown in Fig. 1. The AS field indicates whether the block is allocated. The size indicates the size of the

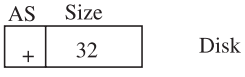

Fig. 1. Fields maintained by the TS method. AS stands for allocation status.

block. We refer to this algorithm as storage management method I or TS for *tagged size*.

In order to allocate a block, the AS field is set to indicate that the block is allocated. If the block does not have to be split, only one disk access is required. In order to deallocate a block, the AS field is set to indicate that the block has been deallocated. If coalescing is not required, only a single disk access is required. [2]

In order to reconstruct in-memory free data structures such as free lists after a system failure, the system examines header blocks starting from the first one within the file. If all blocks are contiguous and header blocks are maintained within the storage blocks themselves, the block size is used to determine how to locate the next header.

An optimization which can reduce the number of disk accesses needed to reconstruct in-memory data structures is to store header information in a contiguous area of the file separate from the blocks themselves. That way, all of the headers can be read in using a small number of disk accesses. Multiple headers can be read in using a single block read. If all of the headers cannot fit in a single contiguous area, a number of contiguous areas containing the headers can be chained together. This approach might consume extra space for block headers. However, unless block sizes are very small, the relative overhead for headers is not significant, and disk space is relatively cheap. This approach is also useful if blocks are not stored contiguously on disk or in a file. Storing header information in a known area of disk in which each header identifies the location of the block allows each block to be located.

If header information is maintained within storage blocks themselves, an optimization which can significantly reduce startup times after normal shutdowns is to output in-memory data structures to contiguous areas of disk just before the system shuts down. During startup, the system obtains the in-memory data structures from the information sent to disk before shutdown instead of from the headers on disk.

During startup, it is not necessary to examine header information for all free blocks. The system should cache information in main memory about enough free blocks to allow efficient allocation for the expected steady state case. If all free blocks contained on in-memory data structures are used up, the system can incrementally obtain information about additional free blocks from headers stored on disk.

[2] In some cases, it may be necessary to read the size of a block being deallocated from disk. If this is the case, then two disk accesses may be required: one to read the block size and a second to mark the block as free.

2.2. Storage management method II: the PL method

In some situations, it may be desirable to actually maintain list structures on disk. For example, different lists might be used for segregating blocks by size. Using storage management method II, one or more lists are maintained containing both free and allocated blocks. This algorithm is known as the PL method since it maintains persistent lists. Fig. 2 shows the header information on disk for blocks when storage management methods II and III are used. Block headers are augmented to contain a pointer to the next block on a list. In addition, a list head pointer for each list is maintained on disk (DLH). A pointer to the head of each disk list is also cached in memory (MLH).

The system may also maintain lists of free disk blocks in memory which do not necessarily correspond to lists on disk. The free lists in memory would typically be searched in order to perform allocations and deallocations. Lists on disk would be examined to populate in-memory free lists either during startup or in situations where the system is populating in-memory free lists incrementally. Note that the PL method provides two methods for locating blocks by examining disk. One method is by examining headers as in the TS method. The second is by examining lists on disk.

When a block is allocated, the AS field is set to allocated. When a block is deallocated, the AS field is set to deallocated. Both operations require a single disk access.

Disk accesses are required to add new blocks to disk lists. When a new block is created, it is placed at the beginning of a disk list. The AS, size, and pointer header fields on disk can be updated using a single block write. The pointer to the head of the disk list also needs to be updated. If the DLH field is updated immediately, an additional disk write is required. In order to eliminate the additional disk write, the DLH field is not updated immediately. Instead, only the MLH field is updated. Periodically, the system checkpoints by copying the MLH to the DLH. The cost for updating the DLH is thus amortized over several operations and can be kept quite low. If there are multiple disk lists, DLH's can be maintained in close proximity to each other on disk so that they can be updated in a single block write. Fig. 3 shows how the PL method works.

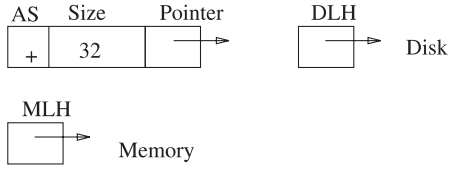

Fig. 2. Fields maintained by storage management methods II and III (the PL and PFL methods). AS, DLH, and MLH stand for allocation status, disk list head, and memory list head respectively.

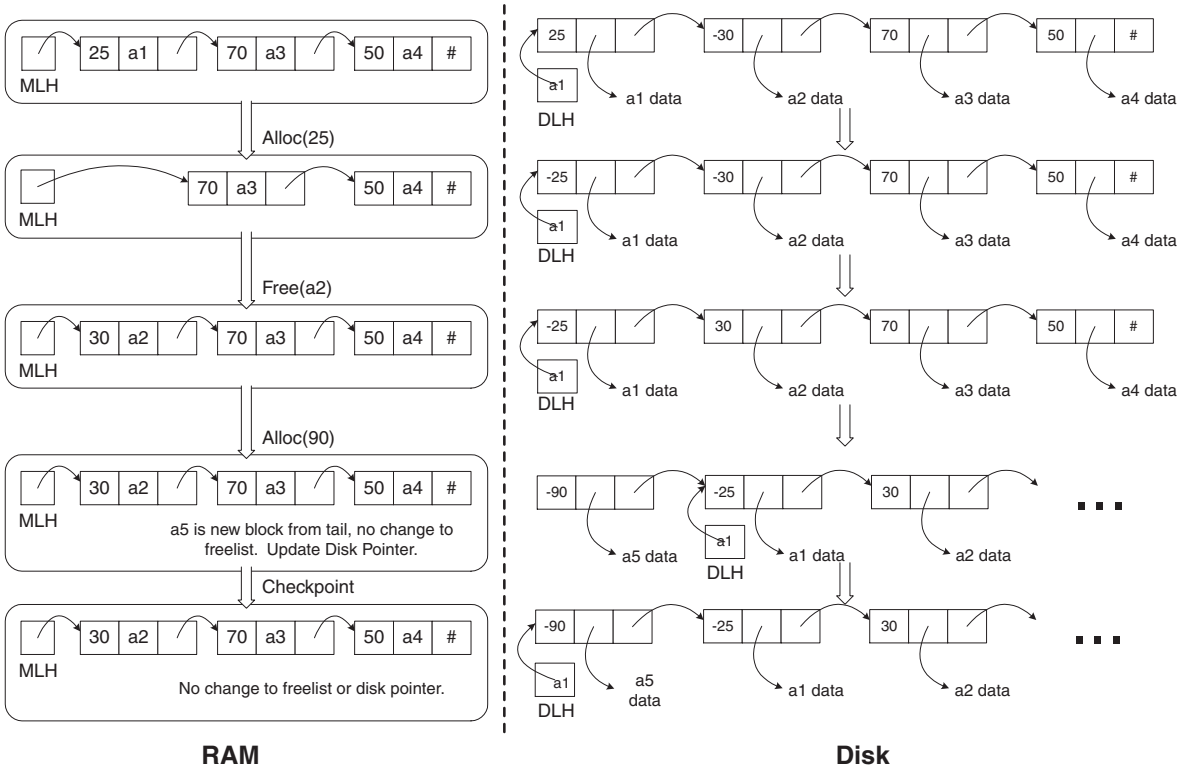

Fig. 3. Allocation and deallocation using the PL method shown as a series of snapshots from top to bottom. Main memory data structures are shown on the left, and disk data structures are shown on the right. Positive sizes indicate free blocks, while negative sizes indicate allocated blocks.

In the event of a failure, the system might lose track of a small number of free blocks. Such blocks would have been created followed by a failure before a checkpoint which would have allowed the blocks to be reachable from DLH's. It is possible to reclaim these blocks if the system periodically scans through block headers to perform administrative operations such as coalescing adjacent free blocks.

If a block is split, it may have to be moved to a different list if, for example, disk lists are determined by size. Moving objects between disk lists requires extra disk accesses. Because of the extra disk accesses required for modifying lists, storage management method I is preferable unless there is a compelling reason for maintaining disk lists. If the lists do not change frequently, however, the extra disk accesses are minimal. Lists do not necessarily have to be updated after allocations and deallocations. They only have to be updated after new blocks need to be added to a list or a block moves to a different list. Some additional space may be required for list pointers. Unless blocks are very small, the relative space overhead for list pointers will not be significant.

2.3. Storage management method III: the PFL method

Storage management method III is for situations where it is desirable to maintain one or more free lists on disk, as opposed to disk lists containing both free and allocated block as in the PL method. Storage management method III is known as the PFL method since it maintains persistent free lists. An advantage of the PFL method over the PL method is that disk list traversals do not have to scan through allocated blocks in order to locate free ones. A disadvantage of the PFL method over the PL method is that the disk lists change more frequently. More I/O is required to update pointers on disk. After a failure, the system must examine free lists in order to remove any allocated blocks at the beginning of the lists. No such procedure is required for the PL method.

Using the PFL method, allocations are made from the beginning of free lists. If, for example, all blocks on a list are known to be the same size, then it is appropriate to always allocate from the beginning of a list. This is the case for many of the free lists for quick fit and the multiple free list fit algorithms described in the next section.

The PFL method also uses the headers on disk for blocks shown in Fig. 2. MLH's are current. DLH's may be slightly obsolete. They are periodically updated from MLH's.

The system caches at least the beginning of each disk list in memory and uses the cached copies for performing allocations and deallocations. When a block is allocated, the AS field is set to allocated. In order to

perform the operation using a single disk access, the DLH field for the list which contained the block is not updated immediately.

When a block is deallocated, the AS field is set to deallocated, and the pointer field is set to point to the previous first block on the list. The AS and pointer fields

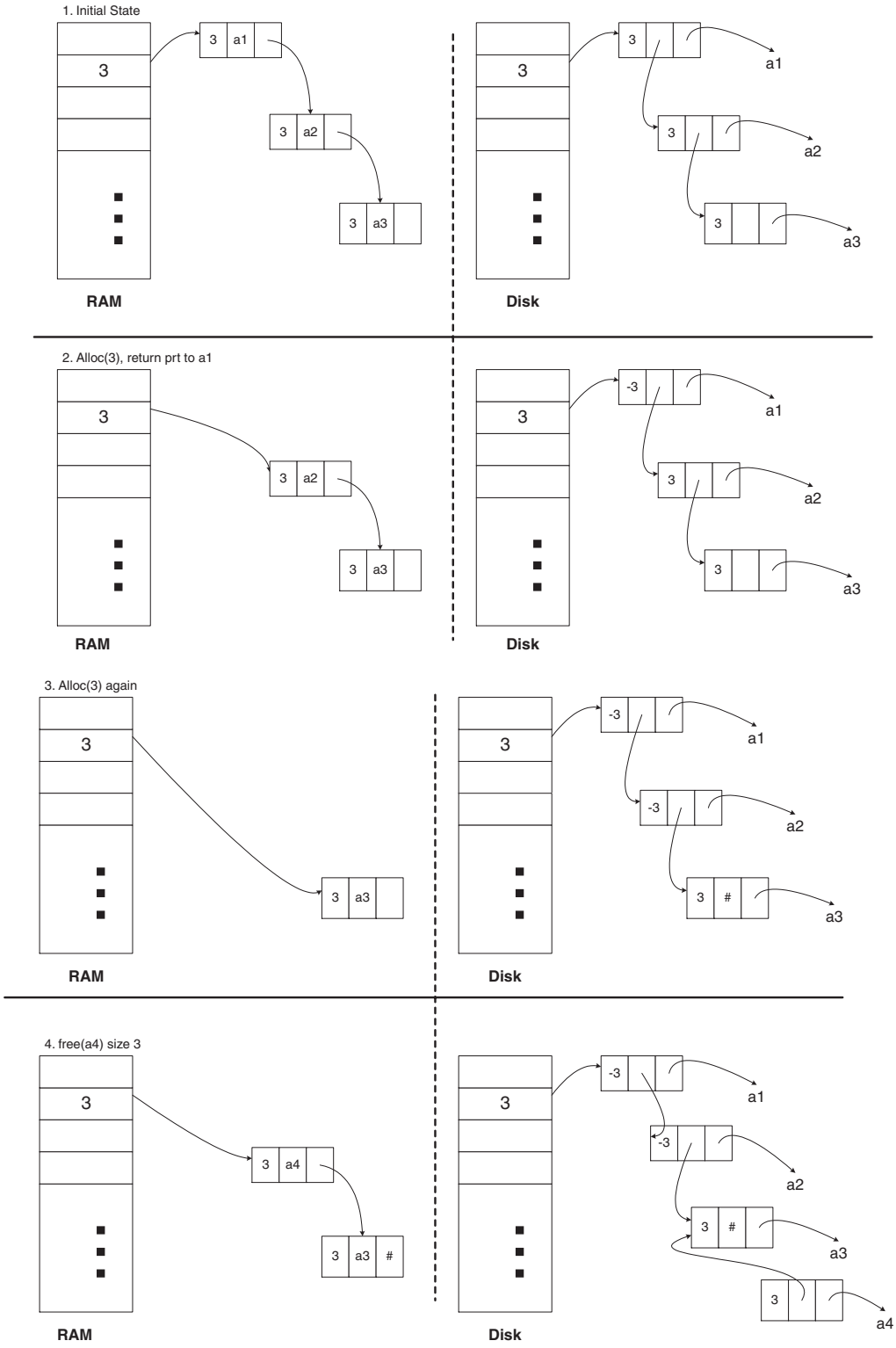

Fig. 4. Allocation and deallocation using the PFL method shown as snapshots from top to bottom. Main memory data structures are shown on the left, and disk data structures are shown on the right. Positive sizes indicate free blocks, while negative sizes indicate allocated blocks.

can be maintained in close proximity to each other on disk, allowing them to be updated in a single disk access. In order to avoid an additional disk access, the DLH field for the list which contained the block is not updated immediately. The system periodically checkpoints MLH's to disk. The PFL method is illustrated in Figs. 4 and 5.

After a system failure, a disk free list may still contain allocated blocks which were allocated since the last checkpoint. All of these allocated blocks would be at the beginning of the free list. In order to fix this problem, the system must examine each free list and remove allocated blocks from the beginning of each list.

In the event of a failure, the system might lose track of one or more free blocks which were freed after the last checkpoint preceding the failure. It is possible to reclaim these blocks if disk blocks are maintained contiguously and the system periodically scans through disk blocks to perform administrative operations such as coalescing adjacent free blocks.

2.3.1. Comparing storage management methods I, II, and III

The three storage management methods offer different advantages for different situations. We have found the TS method to usually be the preferred method, but there are cases where the PL and PFL methods are preferable.

The TS method performs fewer disk writes for allocations and deallocations than the other two methods. It also requires one less word of storage in header blocks since list pointers are not needed. Information on disk is always up-to-date, and checkpointing is not needed.

The PL method allows multiple lists to be maintained on disk which can be used to locate blocks of specific sizes after a failure using fewer disk reads than the TS method. Checkpointing is needed for the PL method, and list heads on disk may not always be current.

The PFL method employs free lists on disk. It has the same advantage over the TS method as the PL method, i.e., using multiple lists segregated by size, blocks of specific sizes can be located using fewer disk reads than the TS method after a failure. Less searching may be required than with the PL method because lists with the PL method contain both free and allocated blocks. Checkpointing is required by the PFL method. The beginning parts of disk lists may need to be scanned and fixed after a failure. This is not the case with the PL method.

2.4. Persistent multiple free list fit allocation

The previous sections described several methods for efficiently allocating disk storage. The methods did not address the problem of how to minimize the amount of searching required to locate free blocks. They also did

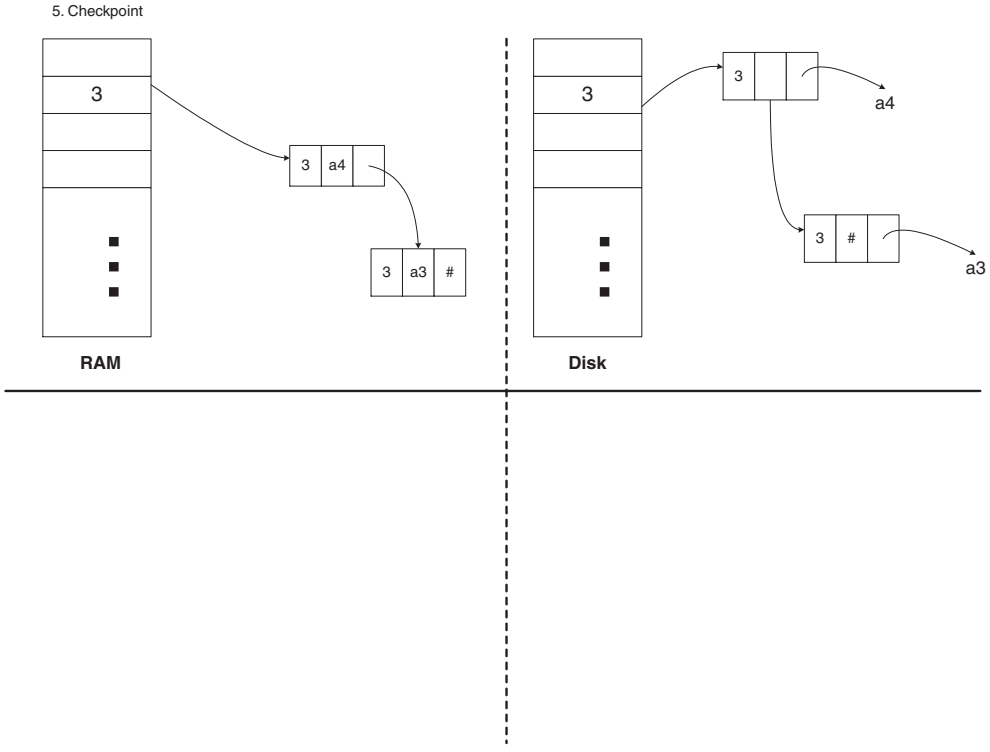

Fig. 5. Allocation and deallocation using the PFL method continued.

not address the problem of how to minimize the number of splits and coalesces, processes which increase the number of disk accesses and are much more costly for disk allocation than for main memory allocation.

We now describe an algorithm for disk allocation which minimizes searching, splits, and coalesces. Our algorithm, known as PMFLF allocation, can be used in conjunction with any of the previously described storage management methods. We have implemented a disk allocation system using PMFLF and the TS method which achieves excellent performance (see Section 3).

PMFLF has similarities to MFLF I (Iyengar, 1996) which is a fast main memory dynamic storage algorithm. PMFLF incorporates key optimizations for reducing disk accesses which are not relevant for main memory allocations. In some cases, PMFLF will perform a bit more searching in allocations in order to reduce disk accesses. In other cases, PMFLF will allocate a slightly larger block than is required in order to avoid splitting the block which would require extra disk accesses.

Both PMFLF and MFLF I use an approach pioneered by Weinstock in his quick fit dynamic storage allocation algorithm (Weinstock, 1976). Several *quick lists* are used for small blocks. Each quick lists contains blocks of the same size. Allocation from a quick list requires few instructions because it is always done from the beginning of the list.

A quick list exists for each block of size ng where g is a positive integer representing a grain size and n is defined over the interval

$$\text{minQL} \leqslant n \leqslant \text{maxQL}.$$

For ease of exposition, we will assume that minQLg is the minimum block size. The optimal value for maxQL depends on the request distribution. A *small block* is a block of size $\leqslant$ maxQLg. A *large block* is a block of size $>$ maxQLg.

This approach has a number of desirable characteristics. Allocation from quick lists is fast; the vast majority of requests can usually be satisfied from quick lists. Deallocation is also fast; newly freed blocks are simply placed at the beginning of the appropriate free list. Segregating free blocks by size also reduces the amount of splitting required during allocations compared with first fit systems, for example. This is a significant advantage for disk allocation because splitting increases the number of disk accesses.

Deferred coalescing is used. This means that adjacent free blocks resulting from a deallocation are not combined immediately. Instead, the system waits until a request cannot be satisfied and then scans through memory or disk and combines all adjacent free blocks. [3]

[3] To limit the disk memory wastage, one can also maintain statistics about the disk fragmentation, and trigger coalescing when fragmentation becomes serious.

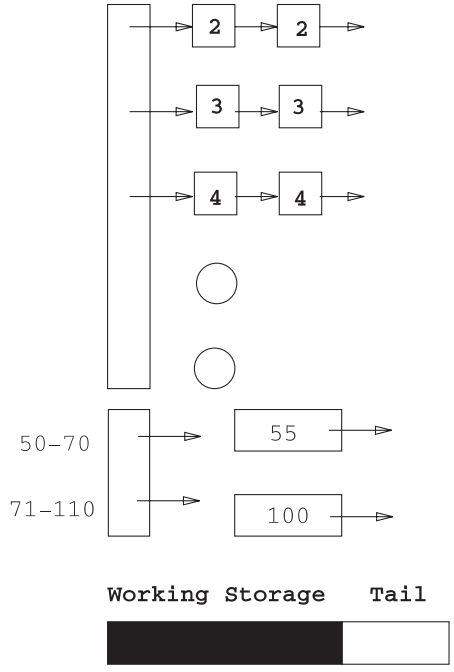

Fig. 6. PMFLF and MFLF I use multiple lists organized by size for free blocks.

Space managed by the system consists of the *tail* and *working storage*. The tail is a contiguous block of free words at one end of the address space which has not been allocated since memory or disk was last coalesced. Working storage consists of blocks which are not part of the tail (Fig. 6). Initially, the tail constitutes all of storage space, and each free list is empty. Blocks are added to free lists during deallocations.

Quick fit uses a single *misc list* (short for miscellaneous list) for large free blocks which is searched using first fit. Allocating large blocks can consume a significant number of instructions. If the percentage of large blocks is high, quick fit does not perform that well. In the worst case when all requests are for large blocks, quick fit degenerates into a first fit system with deferred coalescing, an algorithm which results in terrible performance.

In order to achieve better performance for allocating large blocks, PMFLF and MFLF I use multiple misc lists, each for a range of sizes (Fig. 6). By contrast, a quick list only stores blocks of a single size. A large free block is placed on an appropriate misc list based on its size. Suppose the system has n misc lists designated by l_1 through l_n. Let max be the size of the largest block which can ever exist in the system, low_i the size of the smallest block which can be stored on l_i, and high_i the size of the largest block which can be stored on l_i. We refer to the pair $(\text{low}_i, \text{high}_i)$ as a *misc list range*. Misc lists cover disjoint size ranges. If

$$1 \leqslant i < n,$$

then

$$\text{high}_i + g = \text{low}_{i+1}.$$

The boundary conditions are

$$\text{low}_1 = (\text{maxQL} + 1)g$$

and

$$\text{high}_n = \max.$$

PMFLF maintains free lists and pointers to the tail in memory. That way, allocations and deallocations can be performed using few disk accesses. It is preferable to use the TS method in conjunction with PMFLF, although the other two storage management methods could be used as well. The following discussion assumes that the TS method is being used.

PMFLF maintains an acceptable wastage parameter, w. In satisfying a request for a block of size s, a block with a maximum size of $s + w$ may be used. If a larger block is found, the block must be split in order to avoid wasting too much storage. Since splitting requires extra disk accesses, it is often desirable to reduce disk accesses by wasting some disk storage. In addition, disk storage is cheaper and more plentiful than memory, so wasting some disk storage is less of a problem than wasting an equivalent amount of main memory.

For main memory storage allocation, splitting is not very expensive, so it is generally desirable to split blocks in order to reduce wasted storage.

In order to manage large blocks, a data structure d associating each misc list range with a free list pointer is searched. One optimization which can reduce this searching is to store free list pointers in an array indexed by block sizes. Some searching of d may still be required to prevent the array from becoming too large.

2.4.1. Allocating blocks

PMFLF uses different allocation strategies depending upon the size of the request. For a request for a small block of size s, quick lists are searched in the following fashion:

1. If the quick list for blocks of size s is non-empty, allocate the first block from the list. This involves modifying in-memory data structures and a single disk access for modifying the AS field for the block.
2. If the previous step fails, satisfy the request from the tail.
3. If the previous step fails, examine lists containing larger blocks until a free block is found. This search is conducted in ascending block size order beginning with the list storing blocks belonging to the next larger block class.
4. If the previous step fails, coalesce all adjacent free blocks and go to Step 1.

The following strategy is used for allocating a large block of size s:

1. Determine the misc list for blocks of size s, l_i.
2. Allocate the first block on l_i of size t where $s \leqslant t \leqslant s + w$ without splitting.
3. If the previous step fails, allocate the smallest block on l_i of size t where $s < t$. Split the block into fragments of size s and $t - s$, and return the fragment of size $t - s$ to an appropriate free list.
4. If the previous step fails, satisfy the request from the tail.
5. If the previous step fails and $i < n$, search misc lists starting with list l_{i+1} in ascending order. Use the methods of steps 2 and 3 to search each list until an appropriate free block is located.
6. If the previous step fails, coalesce all adjacent free blocks and go to Step 1.

2.4.2. Deallocating blocks

In order to deallocate a small block of size s, the block is placed at the head of the appropriate quick list. In order to deallocate a large block of size s, the block is placed at the head of misc list i, where

$$\text{low}_i \leqslant s \leqslant \text{high}_i.$$

Since deferred coalescing is used, adjacent free blocks are not coalesced during a deallocation.

2.4.3. Reducing disk accesses for tail operations

When a block is allocated from the tail, both the AS field for the block and the tail pointer on disk need to be updated. This generally requires two disk accesses. In order to reduce the number of disk accesses for tail allocations to one, the tail pointer does not have to be updated on disk after every tail allocation. An updated copy is maintained in memory which is periodically checkpointed to disk. Typically, one checkpoint would occur for several tail allocations.

A block maintains one or more code bytes in its disk header. Such code bytes are set to a certain value to indicate that the block is allocated. The range of possible values for code bytes is large. That way, if code bytes are set to some value by a previously executing program, there is a very low probability that the code bytes would be set to an allocated status.

When a block is allocated from the tail, both the AS field and code bytes are modified on disk to indicate that the block is allocated. Since the AS field and code bytes are in close proximity to each other, they can be updated in a single disk access (Fig. 7).

After a failure, the system must recalculate the updated tail pointer since the value stored on disk may be obsolete. It does so by assuming that the tail pointer is obsolete and examining the AS field and code bytes for the block beginning at the tail pointer. If this data in-

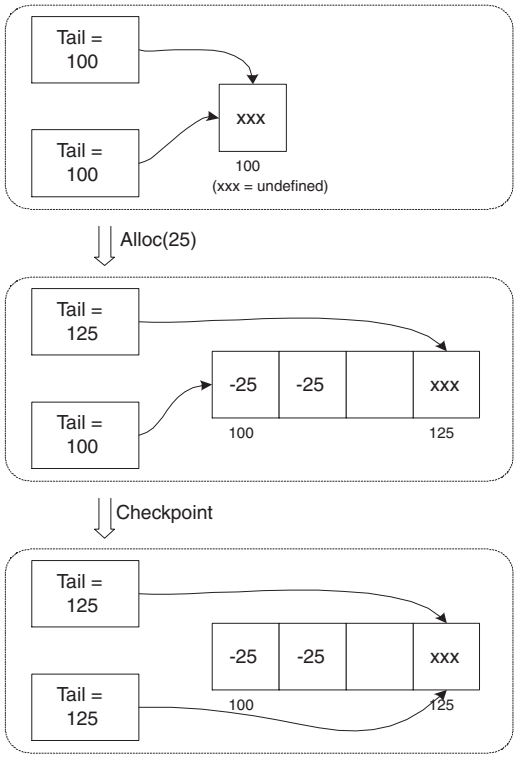

Fig. 7. Reducing disk accesses for tail operations. The top tail pointer in each snapshot is in memory and is always current, while the bottom one is on disk and may be obsolete between checkpoints. Code bytes replicate the tagged sizes of blocks.

Optimized Batch Allocation:

$L_b \leftarrow$ list of free blocks whose size $\geq T$,
$L_r \leftarrow$ sorted requests by size in decreasing order,
$L_a \leftarrow nil$
while $L_r \neq nil$
  do $r \leftarrow head(L_r)$
    $L_r \leftarrow L_r - \{r\}$
    $b \leftarrow BestFit(L_b, r)$, BestFit finds the block of best fit
    if $b = 0$ then fails
    $L_b \leftarrow L_b - \{b\}$
    $(b_1, b_2) \leftarrow b$, split b if necessary
    $L_a \leftarrow L_a + (r, b_1)$
    if $b_2 \neq 0$ then $L_b \leftarrow L_b + b_2$
  od

$L_a \leftarrow$ sorted L_a by block address
while $L_a \neq nil$,
  do $(r, b) \leftarrow head(L_a)$
    $L_a \leftarrow L_a - (r, b)$
    write(r, b)
  od

Fig. 8. The OBA algorithm.

dicates that the block is in fact allocated, the system increments the tail pointer by the value of the size field on disk and continues updating the tail pointer until it locates an AS field or code bytes which indicate that a block has not been allocated.

There is a very small probability that the system will assume that some free storage is actually allocated because AS and code byte fields were set to an allocated status by a previously executing program which modified disk. This probability can be made extremely low by appropriate selection of code bytes. In the unlikely event that this happens, the only penalty is some wasted storage. The system will otherwise function normally.

2.5. Optimizing performance via batched allocation

In some cases, it is possible to optimize performance by performing several allocations at a time and optimizing the disk seek operations for the multiple allocations. We have implemented an algorithm for doing this in our disk storage allocator. We will henceforth refer to this as optimized batch allocation (OBA). In OBA, (see Fig. 8), new objects are not allocated disk space immediately. Instead, they are kept in main memory temporarily. When there are a number of such objects, the algorithm starts to allocate space for them. Blocks smaller than a threshold size T are not used by the allocation algorithm. Allocation works using best fit.

In the beginning, a list of available free blocks, L_b, is obtained and sorted by size in increasing order. Requests are also sorted by size in decreasing order into a list L_r. Then for each request r in L_r, the best fitting free block b in L_b is located. An allocation record is added into list L_a, which is initially empty. If b is split, the remainder is inserted back into L_b. This procedure repeats until all requests are satisfied. Finally, L_a is sorted by block addresses, and blocks on disk are written in address order.

3. Performance

We have tested the performance of our algorithms extensively. This section describes the performance of our algorithms first on Linux then under NT. We then describe some simulations we have done to compare the performance of our algorithms under different workloads.

The PMFLF system just described provides only an allocation mechanism. Most applications will need to

impose some sort of structure upon it. We now show how PMFLF in conjunction with the TS method performs for implementing a hash table. We refer to this structure as a HashtableOnDisk or simply, HTOD. In addition to the disk operations required by PMFLF, writing an item to the hash table requires at least one seek to update the hash index, and one seek to write the data to disk. If the indexed hash bucket is not empty, an extra seek is required per bucket entry. We must search the entire chain of non-empty buckets to determine if the new entry exists, and if so, additional disk operations are required to deallocate the old entry. Choice of a good hash function is especially important with a disk-backed hash table in order to minimize or avoid the additional disk operations. Careful ordering of the operations makes each operation safe against system failures without the need to maintain extra state on disk (and thus avoiding extra disk operations).

The HTOD implements hashing-by-doubling (Aho et al., 1974). Once the ratio of number-of-keys to number-of-hash-buckets exceeds a certain ratio, the HTOD allocates a new hash index of double the previous size and rehashes all entries into it. This operation is coded to be restartable, so that if a system failure occurs during doubling, the process can continue where it left off when the system is restarted.

To determine the effectiveness of the PMFLF algorithms with the HTOD, we have executed several performance tests. The goal of these tests is to understand how the PMFLF/HTOD implementation behaves relative to two other common storage mechanisms: file systems and commercial databases. In the file system approach, a separate file is used for each object. In the PMFLF implementation, a single file is used for storing all objects. The use of a file adds some overhead to the PMFLF implementation. Maximum performance would be obtained by a PMFLF implementation which directly rights to raw disk without going through a file system. However, even in the presence of a file system, PMFLF still outperforms databases and the conventional approach of storing each object in a separate file.

3.1. Description of the Linux tests

The first set of tests were run on a 333 MHZ IBM IntelliStation using SCSI disks and running RedHat Linux 6.0 with the kernel upgraded to 2.2.13. Three storage mechanisms were tested: a commercial database, a native file system in which each object is stored in a separate file, and PMFLF managing all objects within a single file. The file system is the Linux ext2 file system. Each test was run six times and the results averaged, to try to minimize local variations caused by caching and system load. For each storage mechanism, the following tests were performed:

Wp Write to uninitialized persistent storage.
Wn Write to already initialized persistent storage. For this test, every item is rewritten once.
R Look up each item by its name and read it (keyed lookup and read).
Ik Look up every key (non-keyed lookup of the keys).
Iv Look up every data value (non-keyed lookup of the data).
Ie Look up every key/value pair (non-keyed lookup key/value pairs).

The database provides multiple options for committing. It is more expensive to commit every write than to write several objects and only then commit the write. The following tests were run against the database to get more competitive results:

Wp,a Write to an uninitialized database, committing every write immediately (autocommit).
Wp,m Write to an uninitialized database, committing only after every item in the test suite is written (manual commit).
Wp,5 Write to an uninitialized database, committing after every 5th update.
Wn,a Write to an initialized database, committing every write immediately (autocommit).
Wn,m Write to an initialized database, committing only after every item is written (manual commit).
Wn,5 Write to an initialized database, committing after every 5th update.

The motivation for performing the extra database tests is to insure that the actual measurements more accurately reflect actual real-world use. Both the file system and PMFLF tests commit after every write. It should be noted, however, that the storage system in real deployments uses application logic to delay the commit of a transaction. Transactions are typically committed after every 5th–10th update.

The tests used 27,800 items which were actual objects in use for the 2000 Sydney Olympics Games Web site (Challenger et al., 2000). The distribution of object sizes is described in detail in Section 4.4. Many of these objects are HTML fragments which are eventually assembled to produce a complete servable HTML page. Therefore, the average request size is lower than would be expected for a set of objects consisting of complete servable entities. For these tests, only Web documents from fragment caches were used; object dependence graph (ODG) data was not used (see Section 4.4).

Fig. 9 shows the basic timings for the various operations. The figure shows that both PMFLF and the file system considerably outperform the commercial database. PMFLF outperforms the file system in all cases except for **Wn** and **Ik**. For **Wn**, old data must be deallocated as part of the rewrite. However, for the file case,

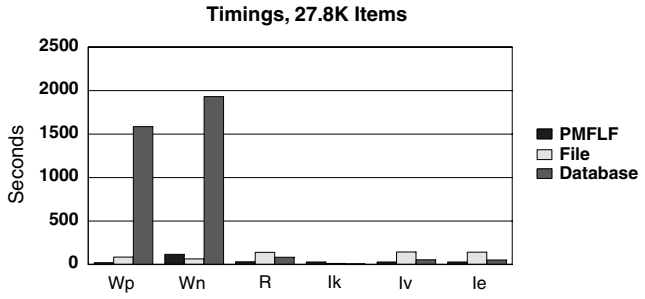

Fig. 9. Comparison of PMFLF to Linux file system and a commercial database.

the directory itself is not updated, saving some disk activity. The same file is used for both the original and updated versions. In addition, the operating system provides a significant boost with its file cache. However, for situations where files are created and/or deleted as in **Wp**, PMFLF considerably outperforms the file system. Fig. 10 shows the same data for the **Wp**, **Wn**, and **R** tests with the database tests removed so we can better compare PMFLF to the conventional file system approach.

The database timings in Fig. 9 are the **Wp,5** and **Wn,5** timings described earlier. Fig. 11 shows how the database performs using different commit options. Performance is poor even when commits are rarely performed.

We also looked at storage requirements for the three different methods. An initialized but empty database used approximately 20–25 MB, while an initialized but empty PMFLF system or file system uses considerably less storage. Disk space consumed for the test with 27,800 objects is shown in Fig. 12.

The file system does not grow or shrink after each run of the file tests. PMFLF initially uses a little less space than the raw file system and grows slightly, reaching a steady state by the end of the six tests. The database initially uses somewhat more space than either of the other two methods but grows rapidly over the first few tests, eventually reaching a steady state.

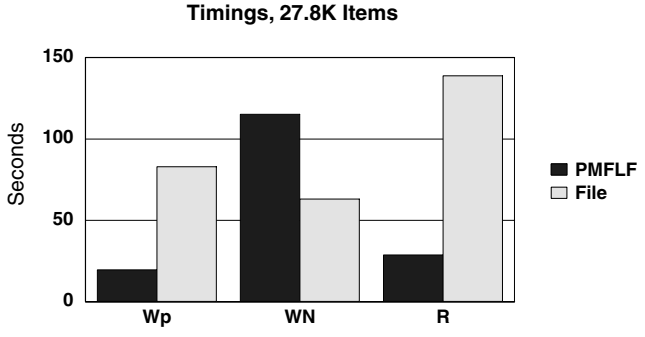

Fig. 10. Comparison of PMFLF to Linux file system.

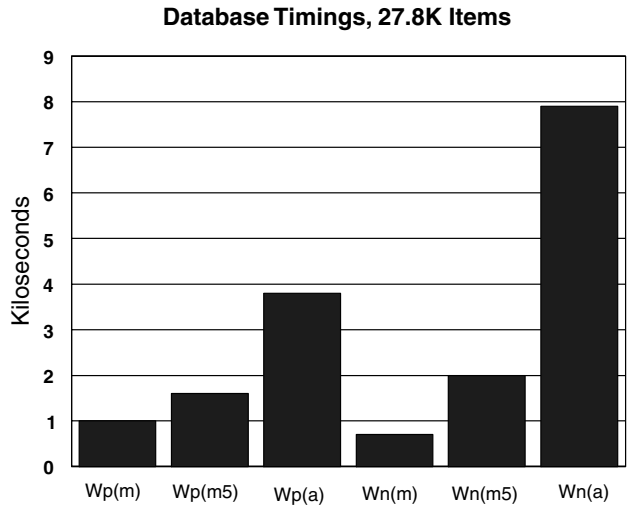

Fig. 11. Database performance using different commit options.

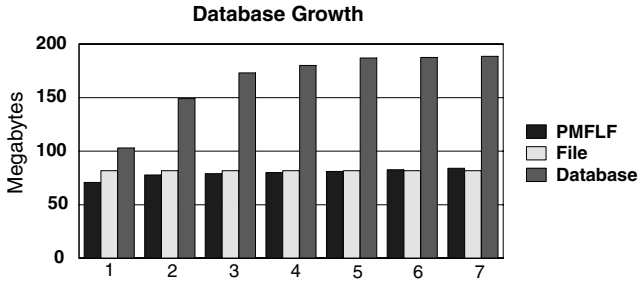

Fig. 12. Disk space consumed by the various methods on Linux. Seven cumulative runs were made. The bargraph shows storage consumed after each run.

3.2. Tests on NT

We also performed a number of tests on a 600 MHz IBM PC300GL running Windows NT 4.0. These tests used the request distribution from the 2000 Olympic Games Web Site as well as a request distribution from a DEC proxy log collected from 8/29/96 to 9/4/96 available publicly from ftp://ftp.digital.com/pub/DEC/traces/proxy/. The DEC proxy log distribution of request sizes is shown in Fig. 13. The DEC proxy log distribution is more heavy-tailed than the Olympic Games distribution, which means it contains a higher concentration of large objects.

We generate streams of allocations and deallocations. Each allocation is followed by a write of the corresponding file or block. The experiment has two phases. In the first phase, we assume storage requirements increase, therefore only allocations are in the request streams. There are 100,000 requests in this phase. In the second phase, we assume the storage requirement is in an equilibrium, and there are 20,000 replacements. Each replacement contains one deallocation and one

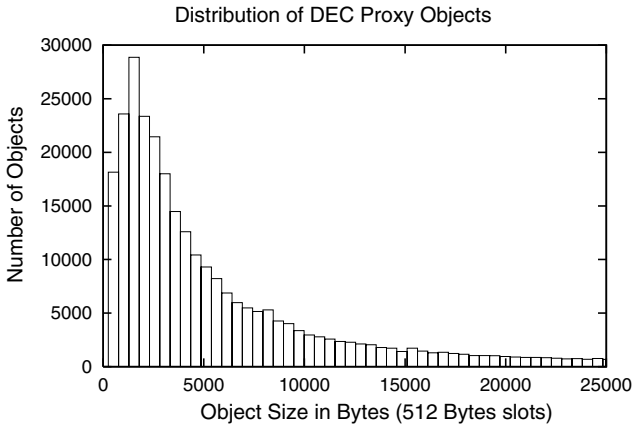

Fig. 13. Distribution of object sizes (in bytes) from a DEC proxy log. The distribution has Pareto parameter $\alpha = 0.91$, and mean object size 15.2 KB.

allocation, followed by write of data. In addition, we randomly choose an existing object for deallocation. The performance is measured from the second phase of the experiment.

Table 1 gives the throughput in number of replacements per second. PMFLF obtains performance improvement of a factor of about 2.5, with a little difference between the two size distributions. The improvement is mainly due to reduction of file operation overhead. File operations require additional disk I/O; the CPU overhead is also high. Since the overwhelming majority of the files are not very large, the file operation overhead dominates the overall overhead as opposed to operations for reading and writing data.

The experiments resulting in Table 1 do not contain any read operations. We design an additional phase after the second phase consisting of 100,000 read requests. Each request randomly chooses an object. Table 2 gives the throughput in number of reads per second. We find that the performance improvement of PMFLF is even larger, nearly a factor of 5. We suspect this improvement is not only because of the high file operation overhead. Therefore, we compared the disk space usage of the two implementations. Table 3 shows that the file system implementation uses much more disk space than

Table 1
Throughput, number of replacements per second

Implementation	Olympic Games	DEC proxy
File system	21.6	18.5
PMFLF	55.6	42.3

Table 2
Throughput, number of reads per second

Implementation	Olympic Games	DEC proxy
File system	20.2	15.6
PMFLF	96.9	69.2

Table 3
Disk space usage

Implementation	Olympic Games	DEC proxy
File systems	643 MB	1731 MB
PMFLF	370 MB	1521 MB

PMFLF. The difference is more obvious for the Olympic Games workload whose mean object size is much smaller (3.7 KB compared to 15.2 KB of the DEC log sizes). The disk space usage affects the performance of read requests in at least two ways. First, the more space occupied, the higher the miss rate from main memory caching. Second, average disk seek distance increases.

3.3. Effect of workload characteristics on disk space usage

We now show how disk space usage of PMFLF varies with the request distribution. Studies have shown that a Pareto distribution models object size distribution well. The cumulative distribution function of a Pareto distribution is $F(x) = 1 - (k/x)^{\alpha}$, where $\alpha, k > 0$, $x \geqslant k$. Parameter α represents whether the distribution is heavy-tailed, and how heavy it is. If $1 < \alpha < 2$, the distribution has finite mean and infinite variance. If $\alpha \leqslant 1$, it has infinite mean and variance. For practical reasons, truncated Pareto distributions are typically used, in which a cutoff C value is another parameter. It means C is the maximum possible object size. The PDF is $f(x) = ((\alpha C^{\alpha} k^{\alpha})/(C^{\alpha} - k^{\alpha}))x^{-\alpha-1}$. The use of a cutoff object size in our experiments is reasonable, since objects whose sizes are beyond the cutoff size can be stored separately. In addition, parameter k represents the scale of a distribution; we define it to be 512 bytes, which is the grain size of PMFLF.

We then design a process to produce requests whose sizes follow the truncated Pareto distribution. We use an inverse method to assign a size to each newly created object. We calculate the inverse function $F^{-1} = k(1 - x(1 - (C/k)^{-\alpha}))^{-1/\alpha}$. Then we apply it to a uniformly distributed variable on the interval $(0, 1)$. That is, a random number $r \in (0, 1)$ is generated, and the size is the integer nearest to $k(1 - r(1 - (C/k)^{-\alpha}))^{-1/\alpha}$.

Finally, the request stream contains object operations, i.e., creations, writes, deletions, and reads. For the experiments in this section, we do not consider read operations. Instead, we design the process to first produce n creations, then produce N replacements. The creation of each new object is followed by a write to the allocated disk space. Thus, α and C represent the size distributions; n and N represent the scale of the storage system.

3.3.1. Effect of distribution parameters

We first study whether the parameters in the object size distribution will affect the performance of the stor-

age allocation algorithm. We design an experiment by setting the number of objects in equilibrium $n = 100{,}000$, and setting the number of replacements $N = 500{,}000$. We then change parameters α and C of the truncated Pareto distribution.

Storage is wasted due to *internal fragmentation* and *external fragmentation.* Internal fragmentation is storage wasted by satisfying a request with a larger block than is needed. We define internal fragmentation at any time t as: $f_i = a/b$, where a is the amount of storage allocated at time t and b is the amount of storage which would have been allocated in a system with no internal fragmentation.

External fragmentation occurs when free blocks are interleaved with allocated blocks. If m is the maximum number of consecutive free bytes of storage, then a request for a block of size $> m$ cannot be satisfied, even if the total number of free bytes is substantially larger than m. External fragmentation is given by the formula: $f_e = M/a$, where M is the maximum storage actually required and a is the storage which would be required in a system with no external fragmentation.

Total fragmentation is storage lost to either internal or external fragmentation. We define total fragmentation quantitatively as the product of internal and external fragmentation: $f_t = f_i f_e$.

Figs. 14 and 15 give the external and internal fragmentation respectively after the system was tested with different distribution parameters. Internal fragmentation decreases as α decreases or C increases because the mean object size increases. Fig. 16 gives the total fragmentation.

Typically, external fragmentation increases as the cutoff size C increases, but not necessarily as α decreases (i.e., more heavy-tailed size distribution). The most striking result is the highest external fragementation when $\alpha = 1.0$ and C is very large. This is due to the high probability of very large requests. In general, when C is not very large, fragmentation never causes serious problems. The minor difference suggests the kind of distribution is not the deciding factor. The cutoff object size might be more important. Nevertheless, it is trivial to set an adequate cutoff object size in PMFLF, for example 2^{10} grains.

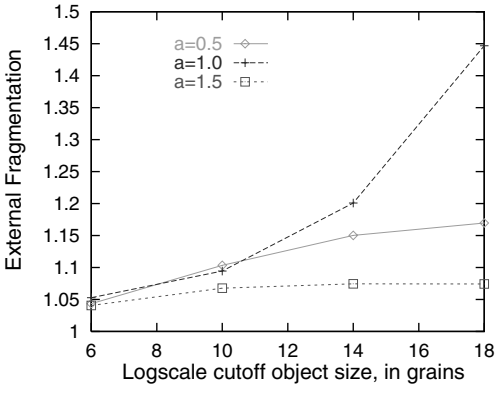

Fig. 14. External fragmentation as α and C change. The x-axis is the base-2 log-value of C (the maximum object size) in grains, e.g., 10 means $C = 2^{10}$ grains.

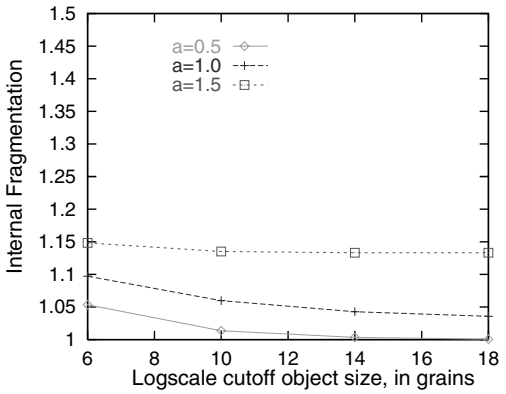

Fig. 15. Internal fragmentation as α and C change. The x-axis is the base-2 log-value of C (the maximum object size) in grains, e.g., 10 means $C = 2^{10}$ grains.

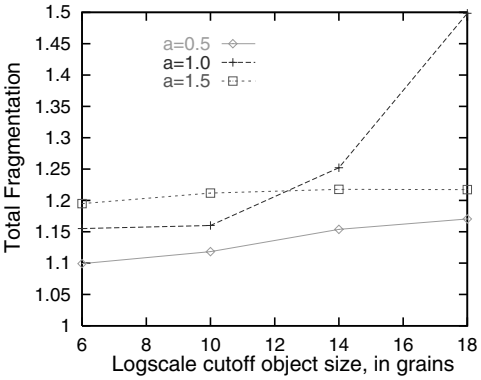

Fig. 16. Total fragmentation as α and C change. The x-axis is the base-2 log-value of C (the maximum object size) in grains, e.g., 10 means $C = 2^{10}$ grains.

3.3.2. Effect of storage system scale

We now show how the storage system scales, as a function of the number of objects. We set $\alpha = 1.0$ and $C = 2^{10}$ grains. Fig. 17 shows the external fragmentation as the number of objects and the number of replacements change. Fig. 18 gives the total fragmentation. Internal fragmentation did not vary and was approximately 1.07. Results indicate that when the number of objects in equilibrium decreases, fragmentation increases. However, it is reasonable to only consider the case when the number of objects is large, since it is unlikely disk I/O overhead dominates in a system which stores very few objects. When the number of objects is large, fragmentation is not a serious problem.

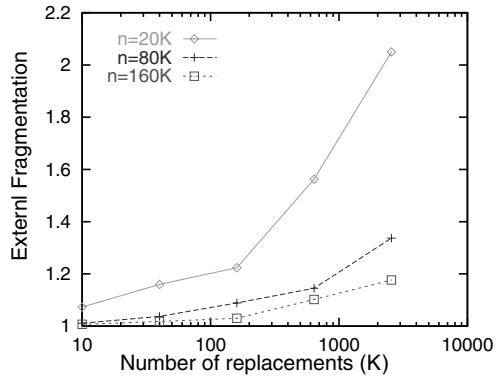

Fig. 17. External fragmentation as the storage system scales.

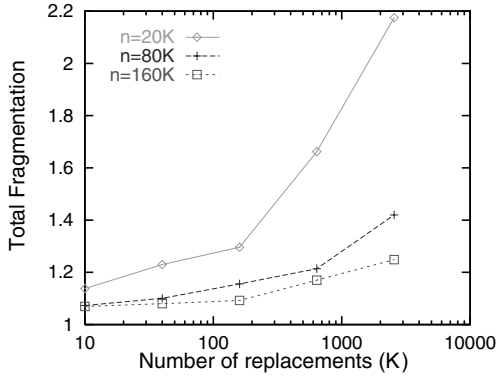

Fig. 18. Total fragmentation as the storage system scales.

3.4. Performance of optimized batch allocation

In this set of experiments, we use the truncated Pareto distribution where $\alpha = 1.0$, $k = 512$ bytes (grain size), and $C = 2^{10}$ grains, resulting in a mean size of 3.8 KB. The acceptable waste of disk space T varies from 1 to 128 grains. The request stream contains 40,000 allocations in the first phase and 40,000 replacements in the second. Fig. 19 shows the throughput in number of replacements per second. It indicates that throughput increases with T.

Figs. 20 and 21 show the external and total fragmentation respectively as a function of T and the number of replacements. Internal fragmentation is constant at about 7.8%. It indicates that when T is small, i.e., less than 4 KB, fragmentation is only a few percentage points of the disk space. Even when the system has run for a long time, the fragmentation does not increase much and reaches a steady state. However, if the acceptable waste increases to above 16 KB, then we find the disk space requirement increases very fast, and eventually, the system needs more than twice the requested disk space to store the objects.

Observing these figures, we find the best operating point might be where T is between 4 and 16 KB. In this range, throughput is close to the maximum possible value while disk fragmentation is not very serious. The throughput is high because the majority of the objects are small, and they can be allocated together in continuous blocks.

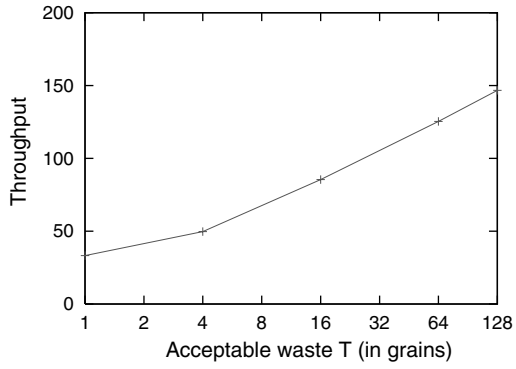

Fig. 19. Throughput of OBA increases as the acceptable waste parameter T increases.

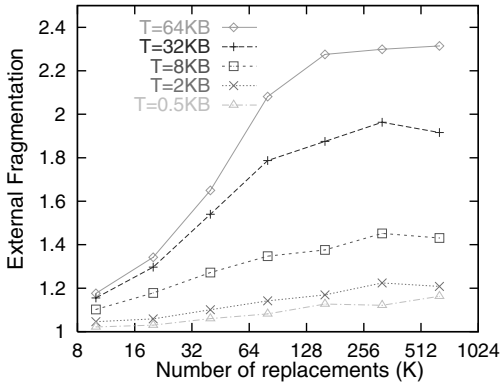

Fig. 20. External fragmentation for OBA increases as the acceptable waste parameter T increases.

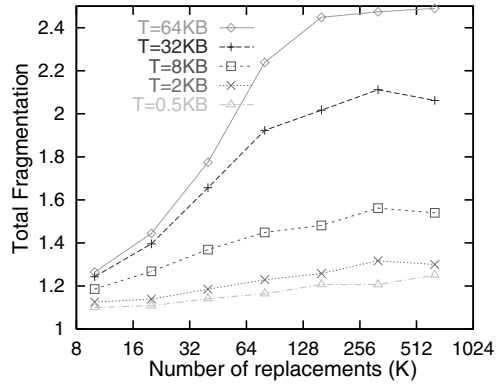

Fig. 21. Total fragmentation for OBA increases as the acceptable waste parameter T increases.

4. Experiences from a real deployment

We now describe our experiences in using our persistent storage manager at a major Web site. [4] It was used for several functions:

- The ODG (described in Section 4.1).
- The fragment caches.
- Temporary storage during page assembly.
- Miscellaneous structures such as page expiry tables.

We will focus attention here on the most important applications of the storage manager: the ODG and the fragment caches.

4.1. ODG overview

All pages for the games were constructed from subsets of HTML pages called *fragments*. When a game event occurred, the publishing system created a set of fragments and issued a *publish event* to a process which then constructed all relevant pages and forwarded the fresh pages to the HTTP servers. The interrelationships of fragments and the pages built from them were maintained in a structure known as an object dependence graph (ODG). By the use of this graph, it was possible to quickly retrieve the list of pages that required rebuilding, given a list of newly published fragments. This process is described in detail in Challenger et al. (2000).

4.2. Fragment cache overview

There were two fragment caches for each instance of the publishing system. One cache, known as the *source* repository, contained an instance of every published fragment as received from the publishing system. The other cache, known as the *assembled* repository, contained pre-assembled versions of all composite fragments. The pre-assembled fragments in the *assembled* repository were a by-product of page assembly and were used to optimize the assembly process. We will subsequently refer to these caches as *cache1* and *cache2*.

4.3. Role of the storage manager

The storage manager implemented the PMFLF algorithm on top of a file system. Each file contained multiple persistent objects. Layered on top of the storage manager was a hash table implementation which provided a simple and effective directory mechanism for locating objects within a PMFLF-managed file.

[4] The 2000 Olympic Games Web site.

The storage manager played two important but distinct roles:

- provide persistence for large, complex data structures, and
- provide an alternative to the traditional method of managing each object in a different file, an approach which does not scale well for a large number of objects.

The ODG and page expiry tables are examples of persistent data structures. In theory, both structures could be initialized from the publishing databases when the systems were booted, and both could be checkpointed regularly. However, the cost of doing so was high. Experiments showed that without persistence, three to four hours were required to construct the ODG from its underlying data, and up to 30 min were needed to checkpoint it. The requirements on the system were to be initialized and ready to publish within two or three minutes of system boot, even in the face of most failures, and to be able to shutdown cleanly in seconds. Clearly an efficient persistence mechanism was required; our PMFLF system provided the performance and functionality needed for this Web site.

The ODG is implemented as a directed acyclic graph. Each node can be connected with an arbitrary number of other nodes as long as a cycle is not formed. By implementing both the node and the edge objects as hash tables, we had a simple migration scheme from a pure-memory structure to a disk-backed structure using hash tables built upon the PMFLF disk manager. Not only did this relieve the memory constraints (see Table 5—the ODG grew to about 800 MB), it provided exactly the level of persistence we needed.

The fragment caches are examples of alternative file systems. Each fragment was addressed by its URL (minus the protocol, host, and port). This type of namespace does not map well onto a normal file system because it tends to be shallow and wide; that is, each "directory" level contains a very large number of files. Normal file systems are not designed to handle this situation gracefully. In addition to anticipated performance problems, we had the requirement of being able to easily and rapidly move and copy the caches to other systems. It was thus felt that maintaining the caches as collections of a large number of discrete files was not practical.

4.4. Analysis of the storage manager files

All instances of the storage manager were configured identically as shown in Table 4. The configurations were based on our best guess of the characteristics of the system based on the pre-games development activities. Since no realistic load could be placed on the systems until the games started, we chose what appeared to be

Table 4
Storage manager configuration

Grain size (g)	Number of buckets (maxQL)	Acceptable waste (w)
512	75	2000

the most robust compromise between performance, disk usage, and ease of maintenance.

Table 5 shows a number of statistics gathered from the system after the games. It should be pointed out that the ODG consisted of multiple PMFLF hash table files:

1. one file contained all graph nodes and edge objects where the edge count was less than 50,
2. a separate file was used for each edge object with an edge count greater than 50. There were approximately 500 edge files for each ODG instance by the end of the games.

The summary in Table 5 shows one entry for the ODG node file, and a consolidated entry for all of the edge files combined. In this table,

1. *File type* indicates the purpose of the file (ODG, ODG edge, fragment caches),
2. *Filesize* shows the final size of the file on disk,
3. *Object count* shows the number of allocated objects within the file,
4. *Bytes requested* shows the total number of bytes that were requested for allocation within the file,
5. *Bytes allocated* shows the actual number of bytes that were allocated for all requests,
6. *Unallocated objects* shows the number of objects that were allocated, freed, and remained free after the end of the Olympic Games, and
7. *Unallocated bytes* shows the number of bytes represented by the unallocated objects.

We were somewhat surprised by the large amount of storage represented by the unallocated objects within the fragment caches. We discovered that near the end of the games, the files were starting to approach 2 GB in size. Although we had written the code using long integers as pointers, the version of Java in use had not yet been updated to include large file support. Therefore, the site managers deleted several hundred megabytes of objects from within the PMFLF files to insure they did not grow to the point where the system would crash.

Table 6
Disk wastage

File type	Internal fragmentation (f)
ODG nodes	1.31
ODG edge files	4.17
Cache 1	1.05
Cache 2	1.03

Table 6 shows the internal fragmentation for each of the disk manager instances. Recall that we define internal fragmentation as $f_i = a/b$ where a is the amount of storage allocated and b is the amount of storage requested. Table 6 shows that *ODG edge files* have significant internal fragmentation. To try to understand what is occurring, we examine the object distributions.

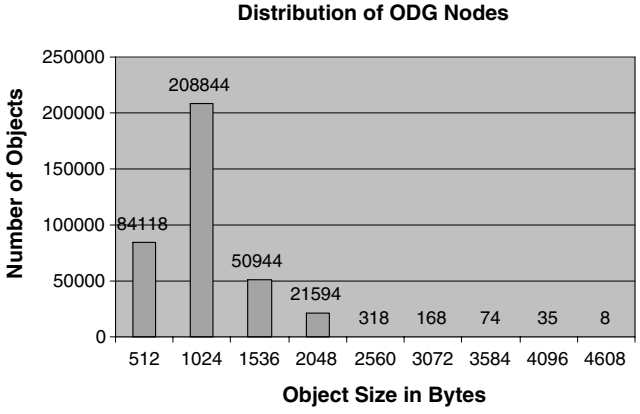

Fig. 22. Object size distribution for ODG nodes.

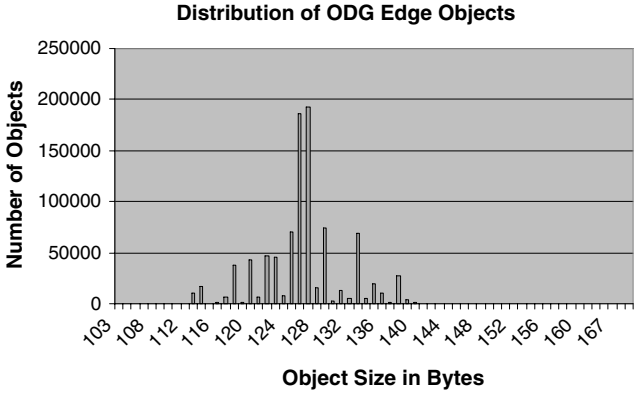

Fig. 23. Object size distribution for ODG edge files.

Table 5
Summary of storage manager file characteristics

File type	Filesize	Object count	Bytes requested	Bytes allocated	Unallocated objects	Unallocated bytes
ODG nodes	407 MB	366 K	295 MB	381 MB	29 K	21 MB
ODG edge files	504 MB	925 K	117 MB	473 MB	10 K	820 K
Cache 1	1.8 GB	276 K	1.5 GB	1.6 GB	44 K	186 MB
Cache 2	1.3 GB	79 K	708 GB	729 GB	48 K	600 MB

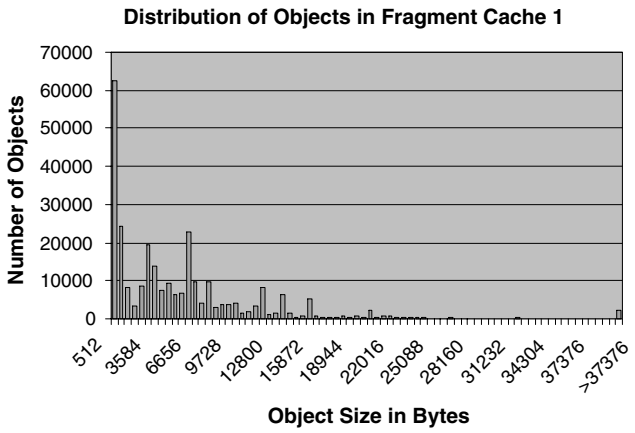

Fig. 24. Object size distribution for fragment cache 1.

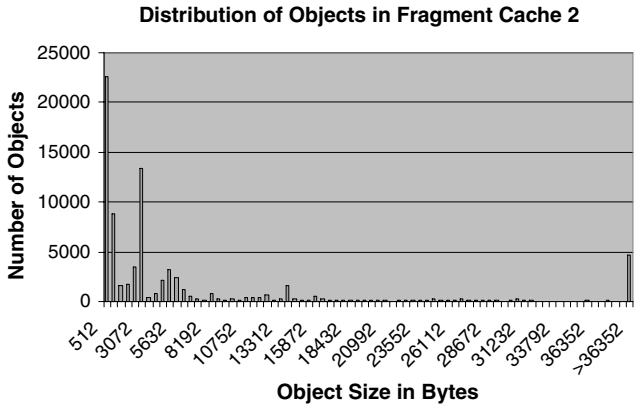

Fig. 25. Object size distribution for fragment cache 2.

Figs. 22–25 show the distributions by size of the objects in the various files. It is immediately clear from these distributions that the grain size for the ODG edge files is not optimal. Most objects were under 132 bytes, and all objects were smaller than 150 bytes. Since the grain size was 512 bytes, about three quarters of the space allocated is lost to internal fragmentation. The low internal fragmentation for the caches indicates that the grain size of 512 bytes was well matched for data consisting of HTML fragments.

5. Summary and conclusion

This paper has presented new algorithms for managing persistent storage which minimize the number of disk seeks for both allocations and deallocations. Our disk allocator can usually allocate or deallocate a block using a single disk seek and still allow the state of the system to be fully recovered in the event of a failure.

Our algorithms can be used for managing multiple objects within a single file or for managing multiple objects over raw disk. The latter implementation would offer the best performance but is more work and less portable. We have developed a portable Java implementation of our algorithms for managing multiple objects within a single file. We described our experiences from deploying our system for persistently storing data at a highly accessed Web site. Our system results in considerable performance improvements over databases and file systems for Web-related workloads.

There are a number of extensions to this work which we are currently engaged in. One extension is to develop a similar sort of disk storage allocator which would be efficient for storing objects smaller than one sector. Such a disk storage allocator would use intelligent techniques for clustering related objects on the same sector to minimize I/O overhead.

Another extension is to characterize robustness and fault tolerance of our storage algorithms along with file systems and databases. The reliability of our storage system has been sufficiently high for our purposes, and no significant data has yet been lost in real deployments due to the failure of our system. However, most commercial databases have more sophisticated features to protect against data being lost than our system, and this adds overhead. What is needed is a quantitative elucidation of the failure probabilities and the overhead of the various storage allocation techniques so that it would be possible to pick the right storage system based on the needs of the application.

Acknowledgements

Several people contributed to the deployment of our storage manager described in this paper including Paul Dantzig, Cameron Ferstat, Ed Geraghty, Paul Reed, Jerry Spivak, and Karen Witting.

References

Aho, A., Hopcroft, J., Ullman, J., 1974. The Design and Analysis of Computer Algorithms. Addison-Wesley.

Challenger, J., Iyengar, A., Witting, K., Ferstat, C., Reed, P., 2000. A publishing system for efficiently creating dynamic Web content. In: Proceedings of IEEE INFOCOM 2000, March.

Iyengar, A.K., 1996. Scalability of dynamic storage allocation algorithms. In: Proceedings of the Sixth IEEE Symposium on the Frontiers of Massively Parallel Computation, October, pp. 223–232.

Kant, K., Mohapatra, P., 2000. Scalable Internet servers: Issues and challenges. Performance Evaluation Review.

Maltzahn, C., Richardson, K., Grunwald., D., 1997. Performance issues of enterprise level Web proxies. In: Proceedings of ACM SIGMETRICS.

Maltzahn, C., Richardson, K., Grunwald, D., 1999. Reducing the disk I/O of Web proxy server caches. In: Proceedings of the USENIX Conference, June.

Markatos, E., Katevenis, M., Pnevmatikatos, D., Flouris, M., 1999. Secondary storage management for Web proxies. In: Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, October.

Matthews, J.N., Roselli, D., Costello, A.M., Wang, R., Anderson, T., 1997. Improving the performance of log-structured file systems with adaptive methods. In: Proceedings of ACM SOSP, October.

Mogul, J., 1999. Speedier squid: A case study of an Internet server performance problem. ; *login*; 24 (1), 50–58.

Rosenblum, M., Ousterhout, J., 1992. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems 10 (1), 26–52.

Rousskov, A., Soloviev, V, 1998. On performance of caching proxies. In: Proceedings of ACM SIGMETRICS, June.

Seltzer, M., Bostic, K., McKusick, M., Staelin, C., 1993. A log-structured file system for UNIX. In: Proceedings of the Winter USENIX Conference, January.

Shriver, E., Grabber, E., Huang, L., Stein, C., 2001. Storage management for Web proxies. In: Proceedings of the 2001 USENIX Annual Technical Conference, June.

Soloviev, V., Yahin, A., 1998. File placement in a Web cache server. In: Proceedings of ACM SPAA, July.

Weinstock, C.B., 1976. Dynamic Storage Allocation Techniques. Ph.D. Thesis, Carnegie-Mellon University.

Arun Iyengar does research into Web performance, caching, and parallel and distributed computing at IBM's T.J. Watson Research Center. He has a PhD in Computer Science from the Massachusetts Institute of Technology. He is the National Delegate representing the IEEE Computer Society to IFIP Technical Committee 6 on Communication Systems, the Chair of IFIP Working Group 6.4 on Internet Applications Engineering, and the Chair of the IEEE Computer Society's Technical Committee on the Internet. He is also an IBM Master Inventor.

Shudong Jin is a PhD candidate in Computer Science at Boston University, where he conducts research in Web and Streaming access workload characterization, caching algorithms, performance evaluation of large-scale content delivery techniques, and Internet measurement and modeling. Mr. Jin obtained his BS and MS degrees in Computer Science in 1991 and 1994, respectively, from Huazhong University of Science and Technology, China.

Jim Challenger is a senior programmer at IBM's T.J. Watson Research Center. His research interests include development and deployment of highly scaled distributed computing systems, caching, and highly scaled Web servers. He has a MS in computer science and BS in mathematics from the University of New Mexico.