Efficient Algorithms for Persistent Storage Allocation

Arun Iyengar, Shudong Jin[*] and Jim Challenger
IBM Research
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
{aruni,challngr}@us.ibm.com, jins@cs.bu.edu

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. Our system has been deployed for persistently storing data at the most accessed sport and event Web site hosted by IBM and results in considerable performance improvements over databases and file systems for Web-related workloads.

1 Introduction

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. Our work has been motivated by the need to persistently store dynamic data 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 memory manager after failures. They also minimize the number of disk accesses. These algorithms can be used in a standalone memory 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.

[*]Author's current address: Department of Computer Science, Boston University, 111 Cummington St., Boston, MA, 02215

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. When a system is shut down, it is often desirable to preserve the state of the disk allocator so that after restart, the disk allocator can continue where it left off. Fast start is also desirable after a system failure.

1.1 Related Work

Several studies suggest that disk I/O overhead can be a significant percentage of the overall overhead on Web servers and proxy servers [6, 11, 7, 12]. Rousskov and Soloviev [12] observed that disk delay contributes 30% towards total hit response time. Mogul [11] 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 [6] 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 issue arises from the management of large amounts of content. To reduce disk I/O overhead, Soloviev and Yahin [14] suggest that proxies use several disks to balance the load and each disk have several partitions to avoid long seeks. Maltzahn et al [8] 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 [9] 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 [10, 13]. Significant performance improvement results from the fact that disk writes contribute significantly to the workload on caching proxies.

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. Throughout this paper, we use the convention

that positive block sizes indicate allocated blocks while negative block sizes indicate free blocks.

2.1 Memory Management Method I

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 as shown in Figure 1. The allocation status field (AS) indicates whether the block is allocated. The size indicates the size of the block.

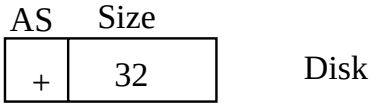

Figure 1: Fields maintained by memory management method I. AS stands for allocation status.

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.[1]

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 memory 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.

[1] 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.

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.

2.2 Memory Management Method II

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 memory management method II, one or more lists are maintained containing both free and allocated blocks. Figure 2 shows the header information on disk for blocks when memory management 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).

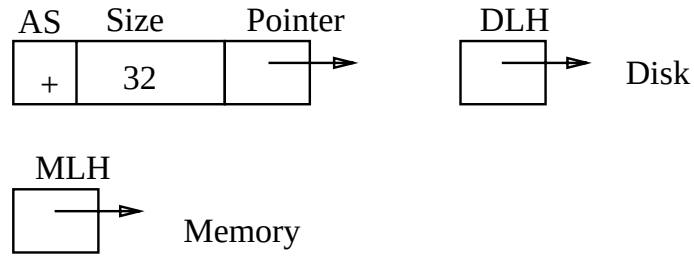

Figure 2: Fields maintained by memory management methods II and III. AS, DLH, and MLH stand for allocation status, disk list head, and memory list head respectively.

The system may also maintain lists of free disk blocks in memory which don't 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 memory management method II provides two methods for locating blocks by examining disk. One method is by examining headers as in memory management method I. 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.

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, memory 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 don't 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 Memory Management Method III

Memory 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 memory management method II. An advantage of method III over II is that disk list traversals don't have to scan through allocated blocks in order to locate free ones. A disadvantage of method III over II 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 method II.

Using memory management method III, 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.

Memory management method III also uses the headers on disk for blocks shown in Figure 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 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. 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.

2.4 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 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 persistent multiple free list fit allocation (PMFLF), can be used in conjunction with any of the previously described memory management methods. We have implemented a disk allocation system using PMFLF and memory management method I which achieves excellent performance (see Section 3).

PMFLF has similarities to MFLF I [5, 4] 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 [15]. 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 $n * g$ where g is a positive integer representing a grain size and n is defined over the interval

$$minQL \leq n \leq maxQL.$$

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

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.

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 (Figure 3). 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 doesn't 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 [3].

In order to achieve better performance for allocating large blocks, PMFLF and MFLF I use multiple misc lists, each for a range of sizes (Figure 3). 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 $(low_i, high_i)$ as a *misc list range.* Misc lists cover disjoint size ranges. If

$$1 \leq i < n,$$

then

$$high_i + g = low_{i+1}.$$

The boundary conditions are

$$low_1 = (maxQL + 1) * g$$

and

$$high_n = max.$$

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 memory management method I in conjunction with PMFLF, although the other two memory management methods could be used as well. The following discussion assumes that memory management method I 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 internal fragmentation.

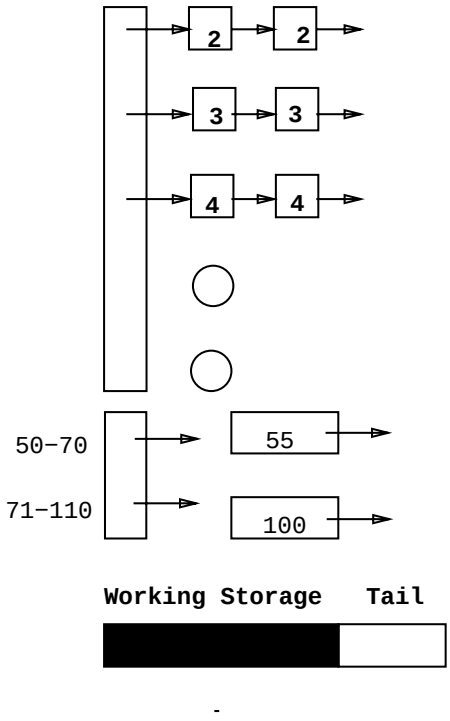

Figure 3: PMFLF and MFLF I use multiple lists organized by size for free blocks.

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 the following strategy to allocate a block of size s where $s \leq low_n$:

1. If the quick list for blocks of size s is nonempty, 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 in memory and go to Step 1.

The following strategy is used for allocating a block of size s where $low_n < s$:

1. Allocate the first block on l_n of size t where $s \leq t \leq s + w$ without splitting.

2. If the previous step fails, allocate the smallest block on l_n 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.

3. If the previous step fails, satisfy the request from the tail.

4. If the previous step fails, coalesce all adjacent free blocks in memory and go to Step 1.

2.4.2 Deallocating Blocks

In order to deallocate a block of size $s \leq maxQL * g$, the block is placed at the head of the appropriate quick list. In order to deallocate a block of size $s > maxQL * g$, the block is placed at the head of misc list i, where

$$low_i \leq s \leq high_i.$$

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 doesn't 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.

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 indicates 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 hasn't 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.

3 Performance

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 memory management method I 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 [1]. 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 HashtableOnDisk 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 an 333MHZ 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 tests used 27,800 items which were the actual objects in use for the most accessed sport and event Web site hosted by IBM [2]. The distribution of object sizes is shown in Figure 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.

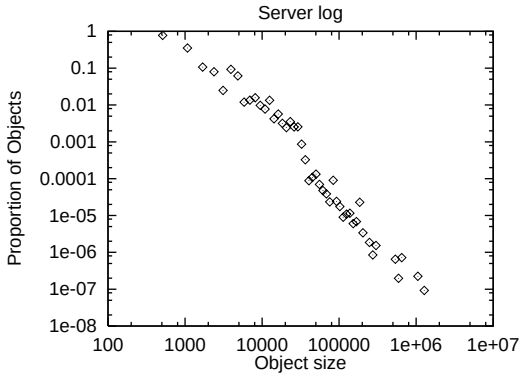

Figure 4: Object size distribution at the most accessed sport and event Web site hosted by IBM. The distribution has Pareto parameter $\alpha = 1.28$, and mean object size 3.7 KB.

Figure 5 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, 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. Figure 6 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.

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 Figure 7.

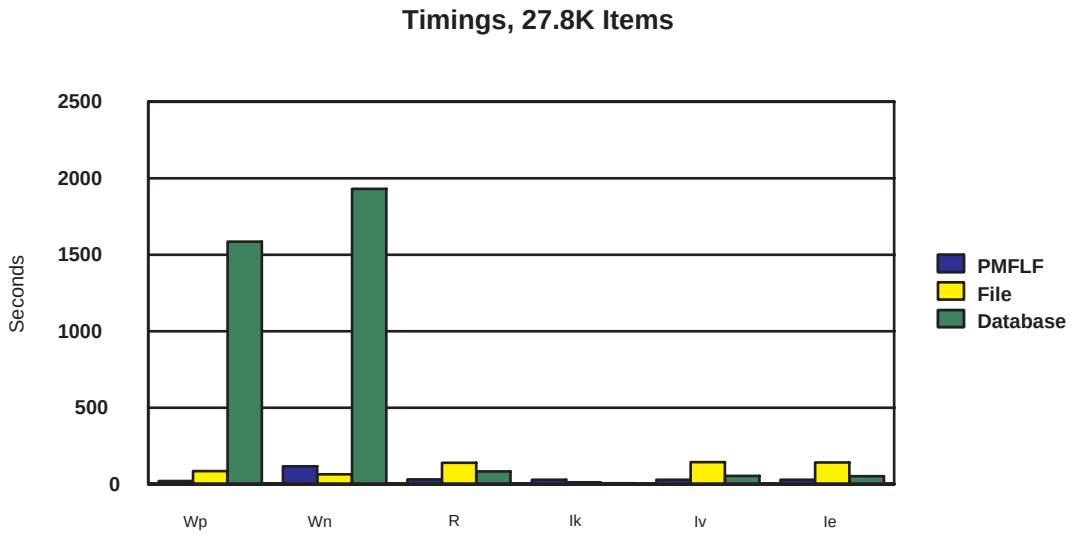

Figure 5: Comparison of PMFLF to Linux file system and a commercial database.

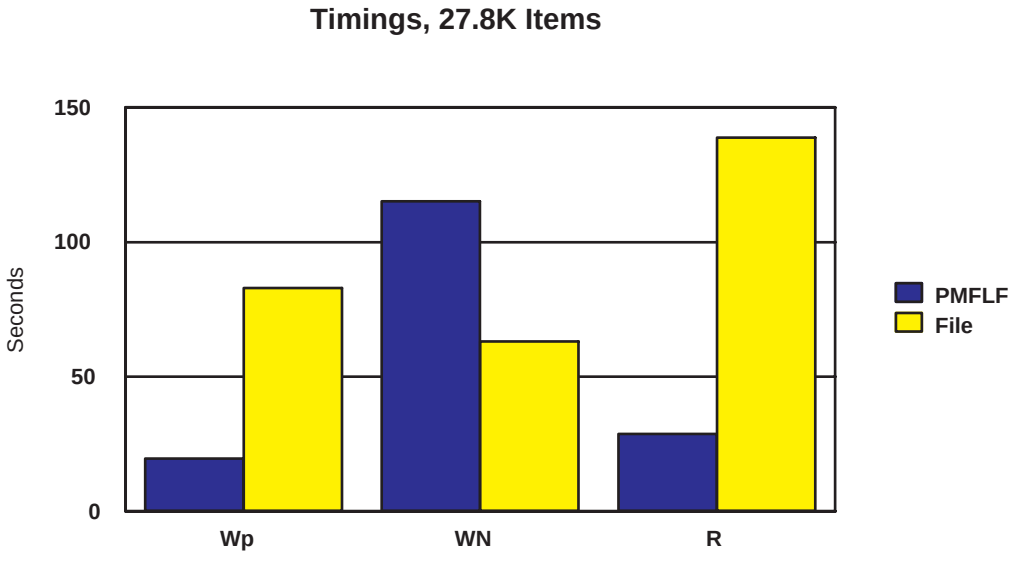

Figure 6: Comparison of PMFLF to Linux file system.

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.

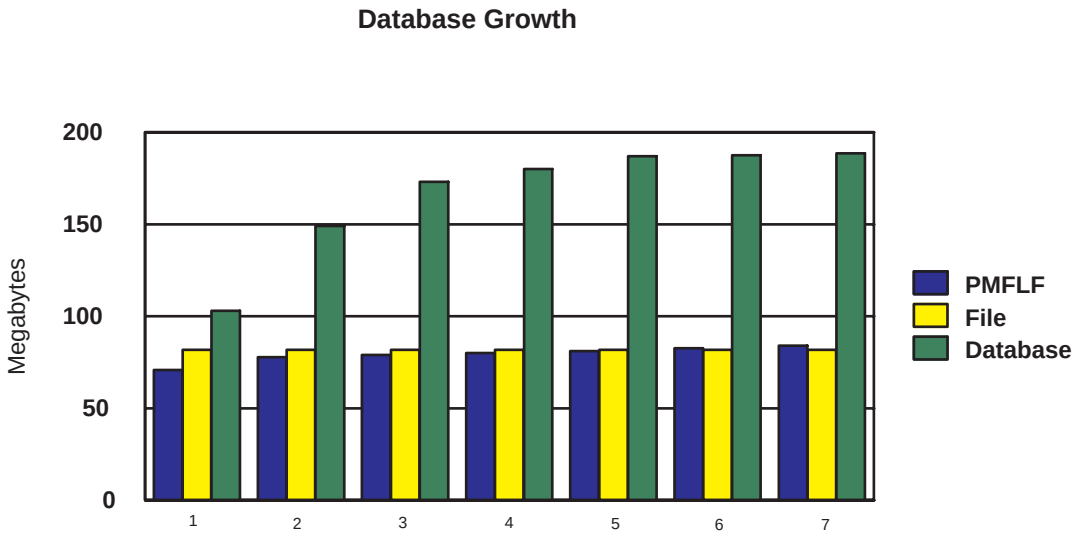

Figure 7: Disk space consumed by the various methods on Linux. Seven cumulative runs were made. The bargraph shows memory consumed after each run.

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 most accessed sport and event Web site hosted by IBM as well as a request distribution from a DEC proxy log collected from 8/29/96-9/4/96 available publicly from ftp://ftp.digital.com/pub/DEC/traces/proxy/. The DEC proxy log distribution of request sizes is shown in Figure 8. The DEC proxy log distribution is more heavy-tailed than the IBM 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 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.

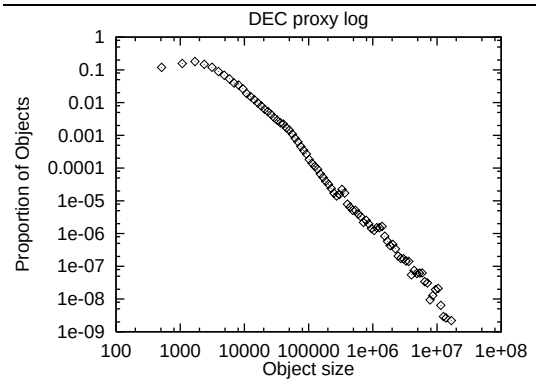

Figure 8: Request size distribution from a DEC proxy log. The distribution has Pareto parameter $\alpha = 0.91$, and mean object size 15.2KB.

Table 1: **Throughput, number of replacements per second.**

Implementation	IBM Server	DEC proxy
File system	21.6	18.5
PMFLF	55.6	42.3

Table 1 gives the throughput in number of replacements per second. Our storage manager 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.

Table 2: **Throughput, number of reads per second.**

Implementation	IBM Server	DEC proxy
File system	20.2	15.6
PMFLF	96.9	69.2

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 our storage manager 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 our storage manager. The difference is more obvious for the IBM workload whose mean object size is

Table 3: **Disk Space Usage.**

Implementation	IBM log sizes	DEC proxy sizes
File systems	643 MB	1,731 MB
Our storage manager	370 MB	1,521 MB

much smaller (3.7KB compared to 15.2KB 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.

Our PMFLF system has been used to store data at the most accessed sport and event Web site hosted by IBM. Due to the difficulties in obtaining performance data of the storage system from this deployment, we instead presented performance data of the storage system running in isolation.

4 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 of memory 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 implemented our algorithms for managing multiple objects within a single file and deployed our storage system for persistently storing data at the most accessed sport and event Web site hosted by IBM. Our system results in considerable performance improvements over databases and file systems for Web-related workloads.

References

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

[2] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A Publishing System for Efficiently Creating Dynamic Web Content. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[3] A. K. Iyengar. Dynamic Storage Allocation on a Multiprocessor. Technical Report MIT/LCS/TR-560, MIT Laboratory for Computer Science, Cambridge, MA, December 1992. PhD Thesis.

[4] A. K. Iyengar. Parallel Dynamic Storage Allocation Algorithms. In *Proceedings*

of the Fifth IEEE Symposium on Parallel and Distributed Processing, pages 82–91, December 1993.

[5] A. K. Iyengar. Scalability of Dynamic Storage Allocation Algorithms. In *Proceedings of the Sixth IEEE Symposium on the Frontiers of Massively Parallel Computation*, pages 223–232, October 1996.

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

[7] C. Maltzahn, K. Richardson, and D. Grunwald. Performance issues of enterprise level Web proxies. In *Proceedings of ACM SIGMETRICS*, 1997.

[8] C. Maltzahn, K. Richardson, and D. Grunwald. Reducing the disk I/O of Web proxy server caches. In *Proceedings of the USENIX COnference*, June 1999.

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

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

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

[12] A. Rousskov and V. Soloviev. On performance of caching proxies. In *Proceedings of ACM SIGMETRICS*, June 1998.

[13] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. A Log-Structured File System for UNIX. In *Proceedings of the Winter USENIX Conference*, January 1993.

[14] V. Soloviev and A. Yahin. File placement in a Web cache server. In *Proceedings of ACM SPAA*, July 1998.

[15] C. B. Weinstock. *Dynamic Storage Allocation Techniques*. PhD thesis, Carnegie-Mellon University, 1976.