NORTH-HOLLAND

Using Asynchronous Writes on Metadata to Improve File System Performance

Li-Chi Feng

Institute of Computer and Information Science, National Chiao Tung University, Hsinshu, Taiwan, Republic of China

Ruei-Chuan Chang

Institute of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, and Institute of Information Science, Academia Sinica, Nankang, Taipei, Republic of China

Due to the increasing gap between CPU and disk I/O speeds, the file system is becoming the performance bottleneck in computer system design. How to improve file system performance is increasingly important. There are two types of entities in a file system: data and metadata. Data mean the actual contents of files. Metadata include access control and other descriptive information about files. Previous research indicates that metadata writes account for 38–40% of disk I/O operations. These large numbers of control request I/Os are usually ignored in traditional file system study, which concentrates only on file-level access patterns. We propose the design and implementation of a metadata-ordering mechanism and its corresponding asynchronous write facility. With such a facility, we can eliminate many synchronous metadata writes, and have the flexibility of choosing a better way to update these metadata modifications to disk asynchronously. Extensive performance evaluation shows that substantial performance improvement can be achieved under various benchmarks. Some other tests are also used to demonstrate the benefits and behaviors of this approach.

1. INTRODUCTION

Recently, technology improvements in CPU speeds, memory sizes, and network bandwidths have changed the characteristics of job execution in computer systems. CPU processing capability is no longer the performance bottleneck, as before; instead, file I/O delays are making the file system the bottleneck. How to improve file system performance is becoming more and more important.

The file system is one of the most visible components of operating systems. Particularly, in the UNIX operating system, it is the most important interface exported to the outside world. Because the UNIX operating system uses file convention to access most available resources, the performance of a file system is a critical component of overall system performance.

If we classify the requests to a file system, we find two kinds of requests. The first kind, called *data request*, mainly operates on that part of the file system storage containing the actual contents of files (the data). For example, read and write system calls belong to this class. The second type, *control request*, accesses these file system storage containing access control and other descriptive information about files (metadata about data). Requests such as file creation, deletion, truncation, and directory manipulation are examples of control requests.

Muller and Pasquale (1991) analyzed the performance of the Berkeley UNIX fast file system (FFS) (McKusick et al., 1984). Using trace data collected at both system call and disk level under intensive disk I/O workloads, they found that data requests only account for 39% of all file system requests, whereas the remaining 61% of requests are control requests (related to metadata including *open*, *close*, *mkdir*, *rmdir*, *create*, *unlink*, *chmod*, *stat*, etc.). At the disk

Address correspondence to Ruei-Chuan Chang, Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, Republic of China. e-mail: rc@iis.sinica.edu.tw

J. SYSTEMS SOFTWARE 1996; 35:43–54

655 Avenue of the Americas, New York, NY 10010

0164-1212/96/$15.00
SSDI 0164-1212(95)00084-E

level, control requests cause 64% of all disk operations, in particular, the write operations coming from these control requests cause 56% of all disk operations, and the rate of disk writes due to control requests is much greater than the rate of disk writes due to write system calls.

Ruemmler and Wilkes (1993) also obtained similar results when they studied the disk access pattern of the HP-UX/4.3BSD FFS. It is reported that user data I/Os represent only 13–41% of the total accesses; the majority of disk accesses (57%) are writes, and 67–78% of the writes are to metadata; 50–75% of disk requests are synchronous. Synchronous request means issuing the I/O request and waiting for its completion before any further processing.

Based on the observations presented in Muller and Pasquale (1991) and Ruemmler and Wilkes (1993), metadata writes account for 38–40% of all disk I/O, and metadata synchronous writes account for 19–33% of total disk I/O operations. These large numbers of metadata synchronous writes are due to the write-through metadata cache on the systems traced. In the UNIX file system, there are many write operations, such as inode and directory updates, which operate synchronously to maintain consistency on the disk. The file system uses synchronous write to ensure an absolute order (McVoy and Kleiman, 1991), but synchronous I/O has many drawbacks, such as forcing the process to endure the I/O delay, reducing concurrency between CPU processing and I/O operations, and issuing many redundant disk operations that frequently rewrite the same location.

In this article, we use the concept of asynchronous I/O to improve file system performance. *Asynchronous I/O* (Buck and Coyne, 1991) means processing the I/O request asynchronously with the computation request. It has the effect of overlapping CPU processing with I/O operation and reducing the delay time of I/O operation from the application's response time. We build a metadata-ordering mechanism to maintain the order of critical metadata writes and then change these metadata updates from synchronous to asynchronous operation. With this approach, we can eliminate many synchronous metadata writes, and we have the flexibility of choosing a better way to update these metadata modifications to disk asynchronously, under the constraint of not compromising file system consistency. Extensive performance evaluation has been done to demonstrate the benefits of this approach.

The rest of the article is organized as follows. In Section 2, we review related work. The concept of asynchronous metadata update is introduced in Section 3. The design and implementation of our metadata asynchronous write facility are described in Section 4. An extensive performance evaluation is given in Section 5. Concluding remarks are given in Section 6.

2. RELATED WORK

File system performance improvement is one of the most important research topics in the operating systems research community. Various approaches have been proposed.

The original UNIX file system is elegant in its simplicity. It has a single block size and a simple-list-based allocation policy. The Berkeley FFS (McKusick et al., 1984) solved many performance problems. It proposed the concept of cylinder group, using a more flexible allocation policy, and provided two block sizes to allow fast access to large files without wasting large amounts of space for small files. File access rates up to 10 times faster than the traditional UNIX file system are experienced. Berkeley's FFS is the predecessor of many file systems developed by various venders. It is also the base of the UNIX File System (UFS) (Leffler et al., 1989; McVoy and Kleiman, 1991).

McVoy and Kleiman (1991) measured the existing SUN UFS and showed that about half the processing power of a 12-MIPS CPU was used to get only half of the disk bandwidth. To meet the increasing throughput demands made both by applications and higher performance hardwares, they suggested a file system clustering technique that groups I/O operations into clusters instead of dealing in individual blocks. Based on their reports, a factor of two increased sequential performance was achieved.

Some researchers tried to increase file system performance through improvement of disk subsystems (Staelin and Garcia-Molina, 1991; Akyürek and Salem, 1993), using diverse disk block rearrangement techniques. The disk driver was modified to copy frequently referenced blocks from their original locations to some reserved space near the center of the disk to reduce seek times. Measurements showed that the seek time was reduced by more than half, and the response time was improved significantly.

Another approach adopted by many researchers is to record user-level activities in tracing files and analyze these data to find the file access behaviors of the system (Ousterhout et al., 1985; Floyd and Ellis, 1989; Baker et al., 1991). Through these techniques, much valuable information was derived. For example, they found that most files had very short life-

times (between 65 and 80% live <30 seconds), ~78% of all read-only accesses were sequential whole-file transfers, and >90% of all data were transferred sequentially.

But it is claimed that file-level trace analysis may not be enough (Muller and Pasquale, 1991; Ruemmler and Wilkes, 1993). Because of the existence of the UNIX buffer cache, most file accesses never reach the disk. So these file-level studies cannot exactly model what happens at the disk I/O level. These studies ignore the effects of traffic generated by file systems, such as metadata update and read-ahead.

Several researchers have noticed the importance of metadata-related I/O operations. For example, McVoy and Kleiman (1991) suggested that if there were a way to ensure the order of critical writes, then the file system would be able to do many operations asynchronously. The performance of some metadata-related commands would be improved substantially. There is another related experience mentioned in Peacock (1992). As an experiment, he changed all of the synchronous inode operations to delayed writes to separate them from disk I/Os. It was found that this change improves the I/O performance substantially. But the problem of file system consistency was ignored in his test.

3. ASYNCHRONOUS METADATA UPDATES

As introduced in the first section, asynchronous I/O means processing the I/O requests asynchronously with the computation requests. It is not a new concept, and has been adopted as a common solution to improve system performance in hardware and software. Although many modern operating systems, such as ConvexOS, UNICOS, SunOS 4.1, and USL SVR4 ES/MP, include support for application-level asynchronous I/O facility, none of them recognize the importance of metadata writes. In this section, we introduce the metadata update behaviors of UFS (Leffler et al., 1989; McVoy and Kleiman, 1991) and demonstrate how to change these metadata writes to ordered asynchronous operations.

In UFS, metadata include inode, directory, indirect block, superblock, and other bookkeeping information. UFS uses inode to represent a file. The inode contains various fields to describe the file type, file size, access time, ownership, access permission, number of links to the file, and a file-content map used to record the disk block numbers for the file's contents. The directory is like a regular file; the system treats the data in a directory as a byte stream, but the data contain the names of the files in the directory in a predictable format so that the operating system and programs such as *ls* can discover the files in a directory. Indirect block is a disk block that contains many pointers that link to real data blocks or other indirect blocks. It is useful when the file size grows beyond the limits that can be represented using only normal directly mapped blocks. The superblock contains information about the file system itself, such as the size and status of the file system, the number of free blocks, the number of free inodes, etc. (Bach, 1986; Leffler et al., 1989). The relationships between the inode, indirect block, and data block are shown in Figure 1.

There are many system calls that may change the contents of an inode or a directory. For example, a *read* system call will change the inode's access time; a *create* system call will alter the contents both of its parent's inode and directory and create a new inode to represent the new file; a *mkdir* system call will create a new inode and its directory data block, altering the contents of its parent's inode and directory data.

In the following, we use the *mkdir* operation of UFS to show typical metadata update operations. Let C be the newly created directory, and P be the

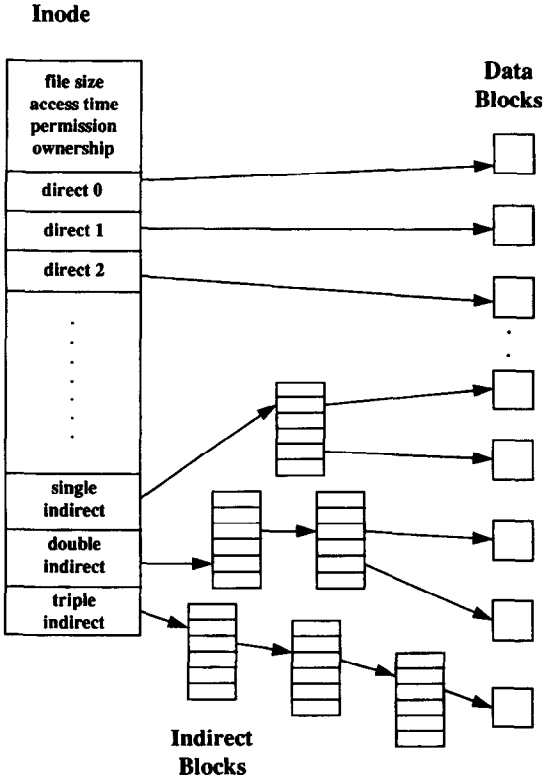

Figure 1. Inode, data blocks, and direct and indirect blocks.

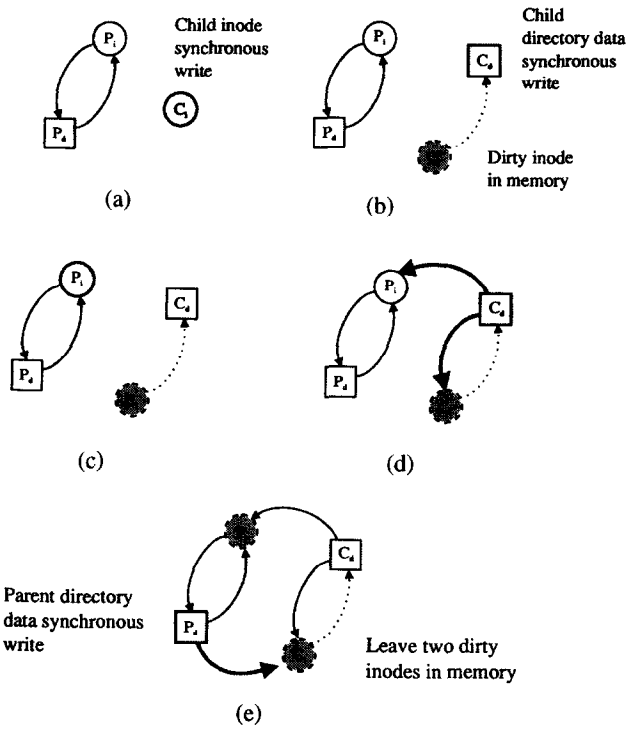

Figure 2. Metadata update in the *mkdir* operation. Let C be the newly created directory and P be the parent directory of C. Let P_i and P_d (C_i and C_d) be the inode and directory data block of P (C), respectively. Solid circles and rectangles are used to represent synchronous writes. A dotted circle represents a dirty inode, and a dotted line means the link is not yet updated to disk.

parent directory of C. Let P_i and P_d (C_i and C_d) be the inode and directory data block of P (C), respectively.

As shown in Figure 2a, the first step of *mkdir C* is to create the new inode C_i, which is filled with initial information such as file mode, link count, user ID, etc. This inode is written to disk synchronously. In the second step, as sketched in Figure 2b, an empty and zero-filled 512-byte child directory data C_d is created and written to disk synchronously, but the dirty inode C_i is left in memory. In the third step, as shown in Figure 2c, a synchronous write is issued to update the link count of the parent inode (increased by 1). Then, in the fourth step, shown in Figure 2d, the system initials the child directory structure, fills two directory entries dot (".") and dot-dot ("..") (IEEE, 1990), and writes the child directory data to disk synchronously. The dirty inode C_i is left in memory. An inode is dirty if its memory contents are modified but not yet updated to disk. Finally, as shown in Figure 2e, a synchronous write is issued to update the parent directory data (this involves the directory entry that points to the child inode). Two dirty inodes P_i and C_i are left in memory.

As illustrated above, there is one synchronous write for C_i, two for C_d, one for P_d, one for P_i, and there are two dirty inodes P_i and C_i in memory that need to be written to disk later. Hence, five synchronous writes and two delayed writes are needed. Synchronous write will block the calling process and increase the response time; however, UFS needs these synchronous writes to keep the file system consistent.

The file system contains many data structures (those metadata described in Section 3) to keep the status information. A file system is in a consistent state if the status information kept in those structures does not conflict. For example, a disk block is either on the free list or assigned to a single inode; if a block number is not on the free list of blocks nor contained in a file, then the file system is inconsistent. To give another example, if an inode number appears in a directory entry but the inode is free, then the file system is also inconsistent because an inode number that appears in a directory entry should be that of an allocated inode (Bach, 1986; Tanenbaum, 1992).

Most file systems read blocks, modify them, and write them out later. Besides, many file operations require multiple phases (suboperations) to be accomplished. If the system crashes before all the modified blocks have been written out, then the file system can be left in an inconsistent state. A utility program called *fsck* is used in most UNIX systems to count, compare, adjust various metadata, and bring the file system back to the consistent state (McKusick and Kowalski, 1985; Tanenbaum, 1992). But if an arbitrary writing sequence is used to complete these operations, then the integrity problem will occur. This situation can be avoided by ordering the write operations properly (Bach, 1986; McKusick and Kowalski, 1985; Tanenbaum, 1992). For example, in the *rename* operation, if we remove its old name from old directory entry first and the system crashes before we add the new name to the new directory entry, the file will be totally lost. But if we write the new directory entry first, at least one version of the file will exist.

Several ordering criteria can assist the processing of the *fsck* utility and are helpful for file system consistency. For example, when removing a file name from its parent directory, the modified directory is written to disk synchronously before it destroys the contents of the file and frees the inode. When removing the contents of a file and clearing its inode, the system frees and writes out the inode first (Bach,

1986). A data structure must exist and be initialized before any pointer is pointed to it. In the *mkdir* operation described above, the synchronous write of C_i in Figure 2a is issued to make sure that the newly created inode goes to disk before directory data and entries point to it. Similarly, the link count of P_i is increased and synchronously written to disk (Figure 2c) before we add the dot-dot ("..") (IEEE, 1990) directory entry into C_d (Figure 2d).

Most UNIX file systems write metadata modifications to disk in a carefully chosen order to minimize file system corruption in the event of system failure. Due to the absence of any ordering mechanism in UFS, these metadata writes as written to disk synchronously as they happen to keep the required ordering (McVoy and Kleiman, 1991).

In this article, we propose a metadata-ordering mechanism and its asynchronous write facility. Once we have such a facility, many synchronous metadata writes become unnecessary (McVoy and Kleiman, 1991). All metadata-related operations are represented as carefully arranged linearly ordered lists. As these file operations are processed, only the corresponding critical writing orders are kept. We reduce the number of these synchronous writes and improve file system performance by changing them to a sequence of ordered asynchronous writes. Because the critical write ordering is kept, *fsck* can be used to return the system to a consistent state when the system crashes before all related operations are accomplished (McKusick and Kowalski, 1985; UNIX, 1990).

For example, we can treat the entire *mkdir* operation as a transaction, and transfer the results to disk. The transfer process (writing the ordered list to disk) must be robust enough to keep the file system consistent even if the system crashes during the transfer process. We consider the direct graph sketched in Figure 2e as a precedence graph and derive a linear order from it to prevent improper write ordering. As shown in Figure 2e, there are three cycles (P_d-P_i-P_d, C_d-C_i-C_d, and P_i-P_d-C_i-C_d-P_i). To get the required linear order, we resketch Figure 2e as the acyclic graph shown in Figure 3.

Two differences exist between the graphs illustrated in Figures 2e and 3. First, the pointer pointing from P_d to P_i is ignored because it already exists on the disk or in the memory, and we ensure that P_i will appear on the disk before P_d points to it. The second difference is that update of the child inode is divided into two parts, C_i and C_i', to break the cycle. The first synchronous write illustrated in Figure 2a is reserved and named C_i. As a newly created directory inode, C_i is certainly not related to others besides its parent, so this synchronous write does not violate existing linear ordering. The other modifications concerning the child inode, including its file content map, are written by the C_i' operation.

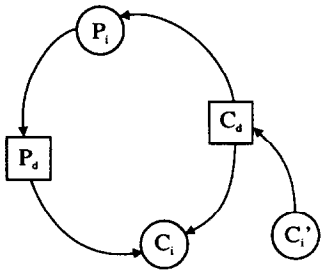

Figure 3. Direct acyclic graph derived from Figure 2e. The directed links represent "depends on " relations (e.g., P_d depends on C_i: C_i must arrive to disk before P_d). Compared with Figure 2e, the pointer pointing from P_d to P_i is ignored, and the child inode's update is divided into two parts C_i and C_i', preventing formation of a cycle.

If we apply a *topological sort* to the graph shown in Figure 3, the required linearly ordered representation illustrated in Figure 4a is derived. The first element in Figure 4a needs to be written to disk synchronously; the other four elements can be done asynchronously. We write the first element C_i in Figure 4a synchronously, because we do not want to maintain two copies of the child inode in memory. This means that C_i and C_i' use the same in-memory data structure (child inode) but describe the state of the child inode at different times. C_i represents the state of the child inode at the stage illustrated in Figure 2a, and C_i' represents the state of the child inode at the stage illustrated in Figure 2e.

With this approach, only one synchronous write and four ordered asynchronous writes are required

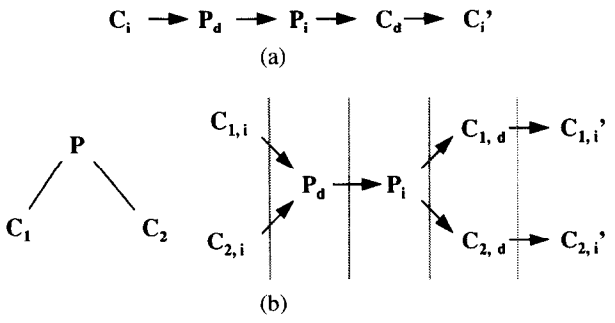

Figure 4. (*a*) A linearly ordered representation of the *mkdir* operation. (*b*) Merging two related *mkdir* operations to reduce the number of disk operations. C_1 and C_2 are two newly created directories, and P is the parent directory of them. $C_{1,i}$ and $C_{1,d}$ ($C_{2,i}$ and $C_{2,d}$) represent the inode and directory data of C_1 (C_2), respectively.

to accomplish the *mkdir* operation. Because the critical write order is kept, this approach does not compromise file system consistency (Bach, 1986; McKusick and Kowalski, 1985; McVoy and Kleiman, 1991; Tanenbaum, 1992). The linearly ordered representation of other operations are derived case by case in a similar way.

We list and compare the synchronous, asynchronous, delayed, and ordered asynchronous writes of several file system operations of UFS and our approach in Table 1. Synchronous write means issuing the write request and waiting for its completion. Asynchronous write means sending the write request to the driver strategy routine and returning immediately. Delayed write means writing the data when the buffer space will be reclaimed by the system for other purposes. Ordered asynchronous write, proposed by us, is a flexible asynchronous write that can be issued at any proper time, but the critical write order is enforced.

Representing metadata update operations as linearly ordered lists yields other benefits besides reducing the number of synchronous writes. For example, as shown in Figure 4b, we make two new directories C_1 and C_2 under an existing parent directory P. If we use the original update policy, we need 10 synchronous disk writes and still leave three dirty inodes P_i, $C_{1,i}$, $C_{2,i}$ in memory. If we apply the asynchronous I/O concept and keep the corresponding linear orders, then we can merge these two *mkdir* operations and have the flexibility of choosing a better time (e.g., low disk load period) for updating. In this way, we only need two synchronous writes and six ordered asynchronous writes that can be accomplished at the appropriate time.

The only potential loss in our approach is that some metadata modifications may come to their disk locations a little later than in the original scheme. But it seems that few meaningful benefits are derived from such urgent synchronous writes. Because the data writes are usually delayed in most popular UNIX operating systems, we can shorten the timing difference of write operations between metadata and data, and so not update these metadata so urgently. Instead, we only keep the meaningful writing orders of these updates and arrange a more effective and reasonable update policy.

4. DESIGN AND IMPLEMENTATION OF THE ORDERED METADATA ASYNCHRONOUS WRITE FACILITY

To implement the ordered metadata asynchronous write, many problems must be solved. First, we need to construct a linearly ordered representation of every file system operation that involves metadata modification. Once we have the linearly ordered representations, the second problem is how to design an ordering mechanism to maintain the order of these disk writes. In particular, it is very important when we want to merge one or more related operations to reduce the number of disk I/O. The last question (update policy) is how and when to accomplish these ordered metadata writes to achieve good performance.

Because of the popularity and attractive features of UFS, we preserve the original interface and existing code as much as possible. This approach has the advantage of maintaining the modularity of UFS and inheriting all of the optimizations that have been done. Based on this decision, we designed and implemented an asynchronous metadata write order manager that keeps the required linear ordering. It is introduced in the following section.

Table 1. The Synchronous, Asynchronous, Delayed, and Ordered Asynchronous Writes of Several Metadata-Related File System Operations in UFS and Our Approach

	UFS			Our Approach			
Operations	Synchronous write	Asynchronous write	Delayed write	Synchronous write	Asynchronous write	Ordered Asynchronous write	Delayed write
mkdir	5		2	1		4	
close		1	1			1	1
create	1		2			1	2
link	1		1			1	1
link	2		1			2	1
unlink	1		2			1	2
rmdir	1		2			1	2
rmdir	2		2			2	2
chmod			1				1
chown			1				1

4.1 Asynchronous Metadata Write Order Manager

There are two main components of the *asynchronous metadata write order manager* (AMOM). The upper half is an order keeper, and its function is to manage those ordered update lists that come from the file system. It intercepts the incoming metadata write requests and merges them into the already existing ordered relations. We add several flags in the inode structure to assist the merging decision directly. For independent operation (not related to any previous unaccomplished operation), the overhead of this check is negligible. Certainly, we also need to eliminate any unnecessary metadata synchronous write in UFS and generate the corresponding linearly ordered lists for those metadata-related operations. The lower half belongs to the *asynchronous metadata update mechanism*, which is discussed later. Its function is to write the next batch of ordered asynchronous metadata writes to the device driver interface (DDI). The relationship between AMOM and UFS is shown in Figure 5.

Two data structures are used in AMOM: asynchronous write element and write ring buffer (Figure 6). Each asynchronous write element represents either inode or directory data. For inode, we simply keep the inode pointer. For directory data, we need to record the vnode pointer and the directory's length and offset in this vnode.

The asynchronous write ring buffer uses different entries to represent different orders of precedence. The entries near the ring head take precedence over other entries. For those unrelated metadata updates, they can appear anywhere in the asynchronous write ring buffer. For example, they can appear in the same linked list under the first ring buffer entry in order to be updated to disk quickly. But for those related metadata updates, we must keep proper write order. This means that the predecessor must be in some linked list closer to the current ring buffer head than the successor. They cannot appear in the same linked list, because we don't know which one will be completed sooner.

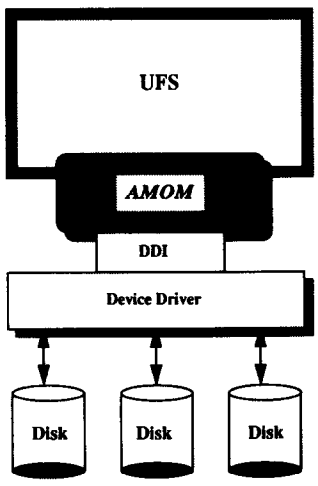

Figure 5. AMOM.

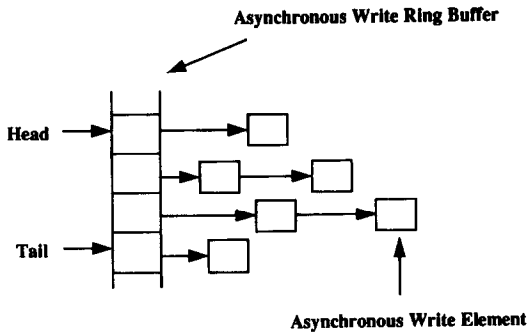

Figure 6. The data structure used by the AMOM.

We still use the *mkdir* operation sketched in Figure 2 as an example. After the completion of a successful *mkdir* operation, the file system will pass a four-element ordered list to the asynchronous metadata write order manager (Figure 7a). Assuming the original ring buffer is empty, the order manager will keep this request, as illustrated in Figure 7b.

Figure 7c illustrates a situation in which we make two new directories C and E under the same parent P. When the second ordered list corresponding to the *mkdir* E comes to the order manager, the previous operation *mkdir* C has not been completed. Thus, we can merge these two ordered lists and save two disk write operations. For those overlapped elements P_d and P_i, we say that they have a hit. Figure 7d illustrates another situation in which we make a new directory C under P, then make a new directory E under C. In this case we hit both C_d and C_i.

Compared with *mkdir*, other metadata-related operations are easier. In most cases, the result is a one-element list indicating that the content of the inode is changed. For example, file size change will generate an inode synchronous write in UFS.

4.2 Ordered Metadata Asynchronous Write

We have introduced the ordering behaviors of AMOM and its relationship with UFS. The second problem is how and when to write those ordered metadata write requests to disk.

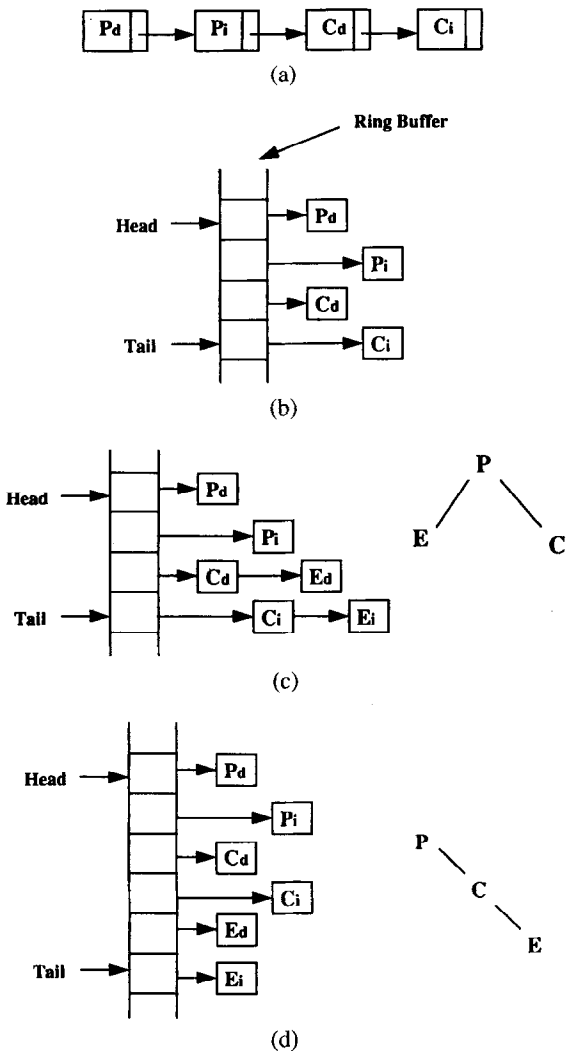

Figure 7. The ordering behavior of the AMOM. (*a*) An ordered list passed from the file system *mkdir* operation. (*b*) Data structure used in AMOM and the internal representation of AMOM after inserting the list sketched in *a*. (*c*) Make two directories C and E under the same parent P; two requests P_d and P_i are overlapped. (*d*) Make a new directory C under P, then make a new directory E under C. Two requests C_d and C_i are overlapped, but in a different way than in *c*.

Two critical components are required in our ordered metadata asynchronous write mechanism. First, an *asynchronous write demon* is necessary to trigger the lower half of AMOM to write the requests linked on the list under the current head entry of the ring buffer asynchronously. Another important component is a *condition sensor*. Its function is to wake up the asynchronous write demon to write the next batch of jobs asynchronously when the previous batch of ordered asynchronous metadata writes have been accomplished.

The asynchronous write demon is designed as a system process. Its main body is an infinite loop that writes the next batch of jobs on the ring buffer to disk asynchronously (sending them to DDI and returning immediately), and then it goes to sleep again. The demon process will be created as an asynchronous UFS (AUFS) file system is mounted.

For the condition sensor, the problem is more complicated. As mentioned above, there are two kinds of asynchronous write elements. For an inode element, only one disk write will be generated. For a directory element, one or more disk writes may be generated depending on its range and contents. We cannot determine the actual number of disk operations before searching the mapped memory region belonging to this vnode and finding out which pages are dirty. To make sure that all the work issued by the previous batch of the ordered asynchronous writes is accomplished, some bookkeeping mechanisms are required to identify which write requests have not yet been completed.

To reduce implementation overhead, we take advantage of the disk-scheduling algorithm. The SCSI disk driver is SVR4/MP uses an elevator algorithm to schedule disk requests. It uses two queuing lists, *next* and *batch*, both of which are sorted according to the requests' disk addresses. The disk services those requests linked on the *next* list one by one. New requests are added to the *batch* list. When all of the jobs in the *next* list have been serviced, it changes head movement direction, moves all jobs in the *batch* list to the *next* list, and begins its service again. Now empty, the *batch* list begins collecting new incoming requests as usual. So it is safe to trigger a new batch of ordered asynchronous writes when the disk arms change direction, because any requests coming from this new batch only go to the *batch* list, and will not arrive at disk after all requests on the *next* list, including those jobs written in the previous batch, are accomplished. Therefore the critical writing orders are maintained. We add one line of code in the disk driver to wake up the asynchronous write demon when the disk service changes direction.

Now we switch to the problem of update policy. The flexibility to choose any appropriate update policy to meet different user requirements is one of the most important advantages of our design. Those users who view reliability as the most important issue can adjust system parameters to write all metadata updates as soon as possible, or write them synchronously. Those users who view performance as an important issue can get more file system performance by adopting delayed or periodic update

policies when the system writes their metadata updates to disk.

Besides, the asynchronous write demon uses some system parameters, such as CPU load and disk queue length, to make decisions. This improves overall system performance.

5. PERFORMANCE EVALUATION

To understand the benefits of our mechanism and the performance behaviors of various update policies, extensive performance evaluation was performed. The evaluation consisted of two parts. In the first part, we compared file system performance between the UFS and our modified UFS, which we named AUFS. Then, we used various update policies and evaluated their effects.

5.1 Evaluation Environment

Our testbed consisted of an AcerFrame 3000/MP computer with a 50 Mhz Intel 486 processor. It contained 32 megabytes of main memory and a Seagate ST2383 SCSI disk. The 317-megabyte SCSI disk had two partitions. The first was a root partition that contained the operating system, and the other was an 80-megabyte test partition mounted on the directory */home*2. The SCSI host adaptor was an Adaptec 1740 in standard mode. The system was running the USL SVR4/MP version 2 operating system, and all measurements were taken without network attachments. During each test, we *umount* and *remount* the test partition to eliminate the effects of memory caching. The root file system was running UFS, and the test partition was running either AUFS or UFS, according to the requirements of our experiments. The block size was 4 kilobytes in all file systems.

5.2 Benchmarks

Two benchmarks, the Andrew and Connectathon test suites, were used to evaluate our work. The Andrew benchmark (Howard et al., 1988) contains five phases, which include the various file system–related operations described as follows.

- Phase 1: many subdirectories are recursively created.
- Phase 2: stresses the file system's ability to transfer large amounts of data by copying files.
- Phase 3: recursively examines the status of every file under the testing directory, without actually examining the data in the files.
- Phase 4: examines every byte of every file under the testing directory.
- Phase 5: is computationally intensive through the compilation of a large amount of files.

The Connection test suites from SunSoft test the functionality of UNIX file systems by exercising all the file system–related system calls. It consists of nine tests:

- Test 1: file and directory creation test.
- Test 2: file and directory removal test.
- Test 3: does a sequence of *getwd* and *stat* on the test directory.
- Test 4: executes a sequence of *chmod* and *stat* on a group of files.
- Test 5: read and write test. It writes a 1-megabyte file sequentially and then reads it.
- Test 6: reads entries in a directory.
- Test 7: calls *rename* and *link* on a group of files.
- Test 8: calls 400 *symlink* and *readlink* on 10 files.
- Test 9: calls 1,500 *statvfs* on the test directory.

5.3 Performance Results

The result of the Andrew benchmark is illustrated in Table 2 and Figure 8. Our AUFS outperforms UFS in phases 1, 2, 3, and 5. In phase 1, the primary operation is *mkdir*, which we have reduced from five synchronous operations to one. So, we achieved the maximum enhancement (70% improvement). Although the dominant operations in phase 3 are

Table 2. Andrew Benchmark Results

	Phase 1: Create directories	Phase 2: Copy files	Phase 3: Stat touch inode	Phase 4: Grep touch byte	Phase 5: compile	Total
UFS	2.57	8.14	4.86	6	47.14	68.71
AUFS	0.71	7.71	4	6.14	44.57	63.13

This table shows the elapsed time (seconds) for each phase of the benchmark on each file system. Reported times are the average across 10 iterations.

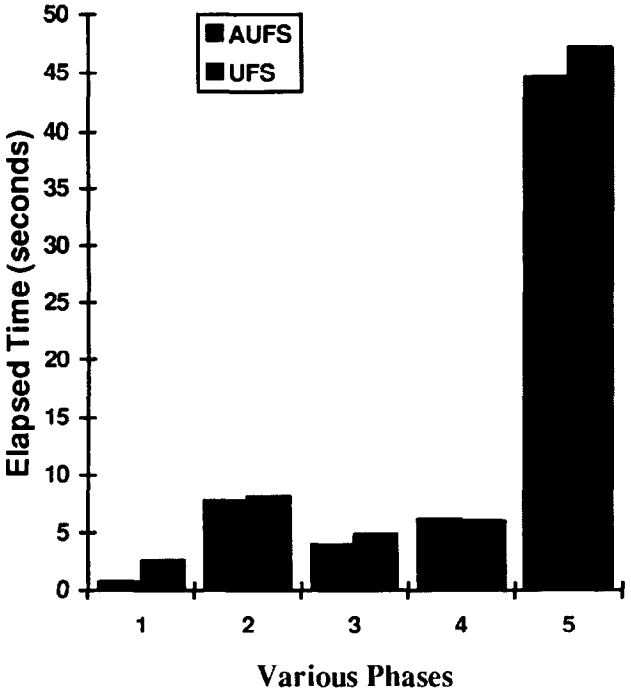

Figure 8. The performance result of the Andrew benchmark.

metadata related, no synchronous writes are issued in either UFS or AUFS. Because the dominant operations in all other phases are not concerned with metadata, the improvement of AUFS is small. In sum, our scheme yields an 8% improvement over UFS.

The results of the Connectathon test suite are illustrated in Table 3. We divide the nine tests into two groups as in Chutani et al. (1992). The first groups are metadata-related operations. The AUFS is better than UFS for all metadata-related operations except test 4. No synchronous writes are generated by *chmod* and *stat* in either UFS or AUFS. Because of the overhead of the asynchronous write facility, the performance of AUFS is lower than UFS in this test. Test 7, *rename* and *link* on a group of files, has many synchronous writes in original UFS; our approach achieves maximum enhancement in this test. Although test 6 is classified in the Other group, there are 200 *create* and 200 *rmdir* calls before and after the *readdir* test. So, we achieved performance improvement. In sum, AUFS demonstrates 60% better performance than UFS.

5.4 Update Policy

We evaluate several update policies. The first policy writes the metadata to disk asynchronously but as soon as possible. Our experience indicates that in the worst case, our scheme accomplishes those metadata update requests just slightly later than UFS does. It is the least successful aspect of AUFS performance, but still meets the requirements of users who view reliability as the most important issue. Besides, various delayed and periodic update policies are also used in our experiments.

We use the Connectathon test suites to evaluate various update policies. The performance results are summarized in Table 4 and Figure 9.

Scheme 1 (immediate update policy) writes the metadata to disk asynchronously but as soon as possible. As mentioned above, under this scheme, AUFS accomplishes jobs just slightly later than UFS does. This version can satisfy those users who view reliability as the most important issue. Schemes 2 and 3 are delayed update policies. Scheme 2 (3) wakes up our asynchronous write demon after a 100-(300-) tick timing delay after we detect a situation in which the previous batch of ordered asynchronous writes have all been accomplished. Schemes 4–6 are periodic update policies. Scheme 5 (4, 6) wakes up the asynchronous write demon every 300 (4,500) ticks (~ 3 seconds).

Based on these test results, the performance of AUFS is better than UFS in most tests, especially those tests emphasizing synchronous metadata write operations. Even the immediate policy outperforms UFS. As is illustrated in Table 4 and Figure 9, when the timing restriction is released, the performance is further improved.

Table 3. Connectathon Test Suite Results

	Metadata Updates					Others			
	Test 1	Test 2	Test 4	Test 7	Test 8	Test 3	Test 5	Test 6	Test 9
UFS	10.425	5.65	0.373	8.82	12.795	1.13	4.295	3.665	0.275
AUFS	4.32	0.19	0.4	0.42	3.808	1.16	3.998	0.933	0.28

This table shows the elapsed time (seconds) for each test of the benchmark on each file system. Reported times are the average across several iterations.

Table 4. Connectathon Test Suite Results of Various Update Policies

Scheme	Metadata Updates					Other			
	Test 1	Test 2	Test 4	Test 7	Test 8	Test 3	Test 5	Test 6	Test 9
UFS	10.246	5.926	0.388	6.66	10.182	1.142	3.976	3.74	0.264
1	8.818	0.393	0.435	1.87	3.648	1.228	2.843	1.4	0.308
2	4.727	0.197	0.408	0.442	4.134	1.22	3.966	1.122	0.284
3	4.65	0.19	0.392	0.44	3.845	1.207	3.918	0.94	0.278
4	8.47	0.245	0.4	1.13	3.645	1.205	3.99	0.945	0.275
5	4.514	0.19	0.414	0.43	4.056	1.172	3.984	0.95	0.282
6	4.526	0.19	0.39	0.42	4.283	1.138	3.948	0.934	0.268

This table shows the elapsed time (seconds) for each test of the benchmark on each file system. Reported times are the average across several iterations. Scheme 1 accomplishes these metadata updates as soon as possible. Schemes 2 and 3 are delayed update policies with 100- and 300-tick timing delays respectively. Schemes 4–6 are periodic update policies. Scheme 4 wakes up the asynchronous write demon every 4 ticks. Scheme 5 (6) wakes up the demon every 300 (500) ticks.

6. CONCLUSIONS

We have designed and implemented an ordered metadata asynchronous write facility in a UNIX system. Many synchronous metadata writes are eliminated in our approach, and a sequence of ordered asynchronous writes is issued. Under the results of the Connectathon test suite, 60% performance improvements are experienced. As demonstrated in the previous section, flexibility is one of the most important advantages of our asynchronous write facility. We can adjust system behavior to meet uses' requirements. If urgent synchronous writes are not necessary, then we can trade them for further performance improvement. Because we choose the writing order carefully when we write these ordered asynchronous metadata operations to disk, it does not compromise file system consistency.

Some issues need to be studied further, for example, the behavior of the mechanism under various update policies. This can help us to control the mechanism more exactly and construct a high-performance adaptable update policy. Another interesting area is how to use this facility to get more benefits, especially by applying it to other related subsystems to achieve even higher performance.

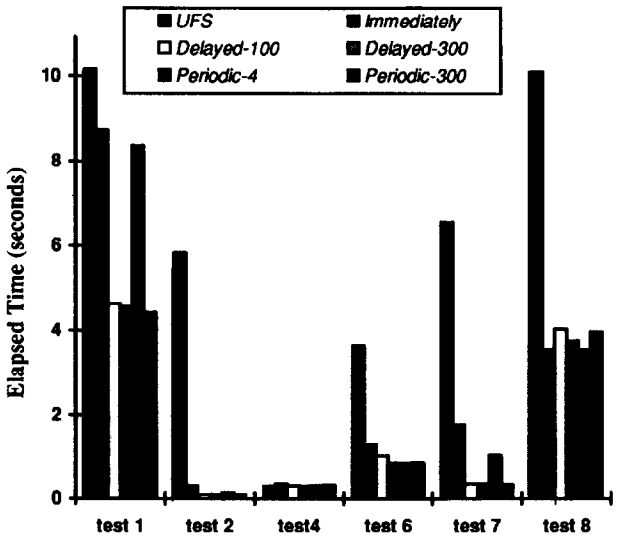

Figure 9. Results of metadata-related tests of Connectathon test suites under various update policies. *Immediately* means to accomplish these metadata updates as soon as possible. *Delayed*, delayed update policy. *Periodic*, periodic update policy.

ACKNOWLEDGMENTS

This research was partially supported by the National Science Council of the Republic of China under grant NSC-0408-E009-055.

REFERENCES

Akyürek, S., and Salem, K., Adaptive block rearrangement under UNIX, in *Proceedings of the 1993 Summer USENIX Conference*, 1993, pp. 307–321.

Bach, M., *The Design of the UNIX Operating System*, Prentice-Hall, 1986, pp. 60–89, 133–140.

Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K., Measurements of a distributed file system, in *Proceedings of the 13th ACM Symposium on Operating System Principles*, 1991, pp. 198–212.

Buck, A. L., and Coyne, R. A., An experimental implementation of draft POSIX asynchronous I/O, in *Proceedings of the Winter 1991 USENIX Conference*, 1991, pp. 289–306.

Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A., and Sidebotham, R. N., Transarc Corporation, The episode file system, in *Proceedings of the Winter 1992 USENIX Conference*, 1992, pp. 43–60.

Floyd, R. A., and Ellis, C. S., Directory Reference Patterns in Hierarchical File System, *IEEE Trans. Knowl. Data Eng.* 1, 238–247, (1989).

Howard, J. H., Kazar, M. L., Nichols, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West,

M. J., Scale and Performance in a Distributed File System, *ACM Trans. Comp. Sys.* 6, 51–81, (1988).

IEEE, *POSIX-Part 1: System Application Program Interface (API) [C language]*, Standard 1003.1, IEEE, 1990.

Leffler, S., McKusick, M., Karels, M., and Quarterman, J., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

McKusick, M. K., and Kowalski, T. J., *Fsck—the UNIX file system check program*, in *UNIX System Manager's Manual*, 1985, pp. 5:1–5:22.

McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S., A Fast File System for UNIX, *ACM Trans. Comp. Syst.* 2, 181–197 (1984).

McVoy, L. W., and Kleiman, S. R., Extent-like performance from a UNIX file system, in *Proceedings of the Winter 1991 USENIX Conference*, 1991, pp. 33–43.

Muller, K., and Pasquale, J., A high performance multi-structured file system design, in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991, pp. 56–67.

Ousterhout, J., K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G., A trace-driven analysis of the UNIX 4.2 BSD file system, in *Proceedings of the 10th ACM Symposium on Operating System Principles*, 1985, pp. 15–23.

Peacock, J. K., File system multithreading in system V release 4 MP, in *Proceedings of the 1992 Summer USENIX Conference*, 1992, pp. 19–29.

Ruemmler, C., and Wilkes, J., UNIX disk access patterns, in *Proceedings of the 1993 Winter USENIX Conference*, 1993, pp. 405–420.

Staelin, C., and Garcia-Molina, H., Smart file systems, in *Proceedings of the Winter 1991 USENIX Conference*, 1991, pp. 45–51.

Tanenbaum, A., S., *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992, pp. 175–179.

UNIX Software Operation, *UNIX System V Release 4: System Administrator's Guide*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990, pp. 5:77–5:118.