Out-of-Place Journaling

Ping Ge and Saba Sehrish and Jun Wang
University of Central Florida

Data Integrity is an important issue in file systems. The basic claim is that a file system must ensure the integrity of metadata operations[6]. In recent years, stronger integrity semantics are required by file systems themselves and upper layer applications[5, 1, 4]. File systems need to maintain the consistent relationships between data and metadata, for instance no dangling references point to data in metadata, correct file size and last modified time. What's more some applications, such as email server and document processing system, even require the file systems to provide functionality to protect their data integrity. One major solution is to keep a single or multiple writes as an atomic operation. A file system should accomplish *atomic write* in two steps. The first step, called *backup*, is to augment the on-disk state with a backup of updates in stable storage. If the system fails, unfinished file system operations can be completed from the backup or abandoned when the system is rebooted. The second step, call *updating*, is to update the system state. Depending on updating mechanisms, there are two major techniques: write-ahead logging and COW transaction mode. Write-ahead logging updates the *values* of data: it keeps the backups untouched and writes data to their original *addresses*. On the contrary, COW transaction mode updates addresses: it discards the original blocks and updates the system structure to direct all requests to the backup. We classify the whole overhead of atomic write into two types: *backup overhead* and *updating overhead*. Backup procedure is indispensable and the overhead is nearly the same for all mechanisms: file system writes all new data to sequential addresses as backup for recovery. On the other hand, different updating mechanisms result in different updating overhead. Compared with backup overhead which is handled by sequential writes, updating overhead impacts the system performance greatly. Figure 1 demonstrates both overhead in logging and COW.

Logging suffers *in-place updating overhead*. It must write updated blocks twice, although the new data have already reached log in the first step[2, 3]. The conflict between log writes and updates may degrade the system performance greatly if the log stores data and is on the same device with file system. The three different journaling modes of ext3 demonstrate the problem clearly in [7]. Although log can make random writes logically sequential and achieve sequential bandwidth, full journal needs to handle twice as much traffic as other modes and generally the traffic in data fixed locations is not sequential. The results in Table 1 show that the overall performance of ext3 full journal is about 1/2-1/3 of ext2's. Larger journal size and flush timer may alleviate the problem. But it will use more system resources, such as disk and main memory space. COW causes *out-of-place updating overhead*. In this mode, once a block is updated out-of-place, corresponding pointers in other blocks should be changed and then they should also be updated[5]. This procedure is recursive until a root node is updated atomically. Therefore the changes of data addresses may result in tens or hundreds of extra blocks allocation or writing in a file system. Our experiments results in Figure 2 show that the performance of synchronous writes in ZFS can not match that of UFS for database workloads. In addition, the chain effect of COW is unpredictable until the file sytem begins to allocate blocks for the updatings. This will make it hard for the file system to evaluate how many blocks are needed for a write operation.

Both overhead are caused by *the tightly coupled relationship between data and its address*. This means: *if the system makes use of the backup, it needs to change the pointers; if it keeps the pointers intact, the backup can not be used*. An ideal solution to avoid updating overhead should let the file system to use the backup directly without changing anything. In this paper, we present a novel mechanism named Out-of-place journaling to reduce the updating overhead. A mapping layer and a log are introduced to the system to simulate *one-to-two addressing* function which breaks the relationship between data and their addresses. Figure 3 shows the architecture of our design. The log has the similar structure as the journal in ext3 file system. Mapping layer sits between file systems and device drivers. It can take all logical addresses as index and direct read operations to the log. When a write operation reaches the mapping layer, it performs out-of-place updates and saves mapping relationship between logical block numbers and log record numbers. On the other hand, all the logical block numbers used by file system are not changed during the write operations because the updates on disk are transparent to the file system. If the log stores the most recent data, the mapping layer can translate the index into the corresponding log record. Thus the system does not need to update data at the fixed location again. If the system crashes, description records in log can be used to reconstruct the mapping layer, and this procedure can guarantee that file system is in a consistent state. We have implemented a simple prototype based on ext3 file system. Some preliminary results are shown in Figure 4.

References

[1] Btrfs design. http://btrfs.wiki.kernel.org/index.php/Btrfs_ design.

[2] The ext2/ext3 file system. http://e2fsprogs.sourceforge.net.

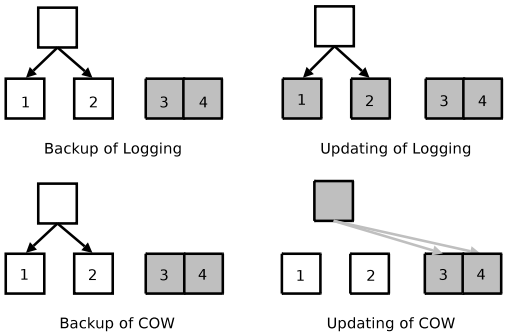

Figure 1: This figure demonstrates the overhead of logging and COW mechanisms. Gray blocks represent updated blocks. The first step of two different modes is the same. Both of them write a backup of new data to stable storage sequentially. In the second step, logging performs in-place updates while COW employs out-of-place updates.

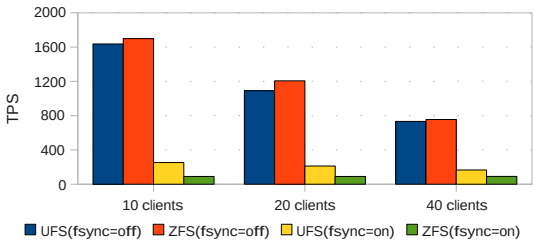

Figure 2: The performance of ZFS and UFS with non-sync and sync operations. Tests run on a Dell Precision 690 workstation with Postgresql and Solaris 10 distribution. We run pgbench, a TPC-B based benchmark shipped with Postgresql, to evaluate the performance of ZFS and UFS.

Table 1: Performance of three different ext3 journal modes. The experiments are done on Dell precision 690 workstation with PostMark. All the default configurations of the benchmark are used except setting transaction number to 1000.

	Ext2	Write Back	Ordered	Full
Time	17.9sec	18.7sec	21.9sec	42.0sec

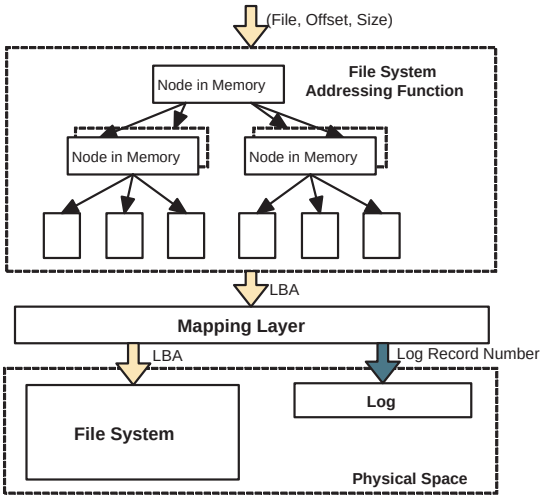

Figure 3: The architecture of out-of-place journaling mechanism.

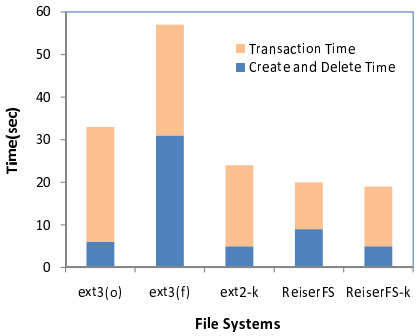

Figure 4: The graph displays the performance of the overall tests of PostMark. We have run ext2 and ReiserFS on our mapping layer. Both of them can provide the same integrity semantic as full journal mode because both data and meta are recorded in the log. These two cases are called ext2-k and ReiserFS-k. Ext3(o) and ext3(f) represents ext3 file system with ordered mode and full mode. We configured PostMark to create 1000 files ranging in size from 512B to 1MB. Other configurations include performing 1000 transactions and setting read or create bias parameter to 5.

[3] Jfs overview: how the journaled file system cuts system restart times to the quick. http://www.ibm.com/developworks/linux/library/l-jfs.html.

[4] Transactional ntfs. http://msdn.microsoft.com/en-us/library/bb968806(VS.85).aspx.

[5] Zfs: Last word in file systems. opensolaris.org/os/community/zfs/docs/zfs_ last.pdf.

[6] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft updates: a solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst. 18*, 2 (2000), 127–153.

[7] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 8–8.