

A Caching Mechanism for FAT File System in Low-performance Embedded System

Jeaho Hwang, Kangheui Won

*RTST, 218 Gajeong-ro, Yuseong-gu, Daejeon, Republic of Korea

jhhwang@rtst.co.kr, khwon@rtst.co.kr

Abstract— FAT file system is one of the most common file systems on various targets and operating systems. However, it has a performance issue on sequential writing to low-bandwidth storage due to reading FAT area and directory entries. In this paper, we design and implement an efficient FAT caching mechanism for sequential writing to low-bandwidth. The mechanism consists two parts: metadata cache and whole FAT read-ahead cache. As a result, we show 20x improvement for sequential writing to low bandwidth storage with our caching mechanism.

Keywords— file system, FAT, cache, embedded system, operating system

I. INTRODUCTION

File Allocation Table(FAT) file system has been widely used since it is developed in 1977. Despite this classic file system has several disadvantages in comparison with modern file systems as NTFS and ext4 it is still used for such as USB memory sticks due to its simplicity and compatibility. Consequently, FAT file system could be considered preferentially to design embedded systems. However, its mechanism could cause catastrophic performance problem if it would run on low-speed storage devices[1].

In order to cover latency of low-speed devices, buffering and asynchronous I/O are commonly used. With these schemes, write commands return just after the data is stored in a buffer entry. The real write operation into the storage devices would be done asynchronously. This approach could increase performance of consequent write operations until the buffer would be full.

On the other hand, since a read operation should be done synchronously it should wait until data would be sent from the device. Above all, the waiting delay consists not only the reading time from device but also writing time of all writing operations in the buffer. Therefore, a read operation between large number of write operations interferes with the performance gain by buffering. In order to maximize buffering effect for large size writing, read operation should be minimized.

FAT file system, however, should make several read operations even on a sequential writing. A sequential writing file system looks the directory entry of the target file to update its size and the FAT sectors to allocate a new cluster to expand the file. As a result, FAT file system mechanisms causes performance drop for sequential write operations[2].

In this paper, we introduce a caching mechanism that eliminates read operation from the device between sequential writing in order to maximize the performance of sequential writing.

II. DESIGN

A. Environment

In this paper, we assume an environment for an embedded system board which is connected with an storage which has 80KBps(KiloByte per second) bandwidth. The embedded board periodically writes large sized log into the storage. Thus sequential writing performance is the most important. The storage has a hardware queue as a command buffer. It stores the command and return if a write command is issued, and stores and wait until the command is processed if a read command is issued.

In order to simulate the environment that we assumed we used P2020 RDB ppc architecture board as a embedded board and RTWORKS, which is a real-time operating system developed by RTST[3], as an operating system. We simulate a low-bandwidth storage using RAM. It is a 400MB virtual storage and formatted as FAT32 file system.

In order to simulate low bandwidth of the storage, we made 12ms delay per both read and write operation. For write operations, *Estimated Time of Completion(ETC)* is calculated. If a write operation is issued after the last ETC, ETC is reset to 12ms after current time. On the contrary, ETC is postponed to 12ms if a write operation is issued before ETC. We can expect that the current buffer would be flushed at ETC. more frequent write operation makes ETC farther. We assume no buffer overflow; buffer size is infinite.

For read operations, since it should wait until the buffer is flushed, it sleeps until ETC and additional 12ms as its own delay. By this approach we can simulate the performance penalty of read operation in sequential writing.

B. Metadata Cache

The primitive approach to prevent passing read operations to device we design a write-through cache of metadata sectors such as FAT area, directory entry. Since sequential writing in FAT file system reads only those metadata, data sectors need not be cached. We implement a prototype LRU cache using 10 entry linked list. When a metadata sector should be read, it first

looks up the cache and insert the target sector into the head of the list.

Since the required cache size is small, a simple linked list could show negligible lookup cost. By applying the cache, hardware reads happen only for file creation. These read cost can be negligible because they happen before mass data writing. After starting sequential write, almost all read by write, sync, close and reopen operations is hit by cache. When the file size become larger, we observed periodic hardware read operation. In order to eliminate those HW read, we should design second approach.

C. Whole FAT Read-ahead Cache

FAT file system manages space as cluster, a set of several sectors. FAT area is a map between files and clusters. If a file expands over the size of a cluster, file system allocates a new empty cluster by looking up the FAT area. Since FAT area also consists of sectors, looking up the FAT area needs read operations.

For example, while a large size of sequential write is being done, file size exceeds current allocated cluster size, and there is no more empty cluster in the current read sector of FAT area. In this case, a new sector must be read, and it could make cache miss so that hardware read which makes performance drop would happen. A common read-ahead approach also cannot guarantee to solve this problem since we cannot predict which sector would be read.

In order to completely eliminate the read operation of FAT area, we designed *Whole FAT Read-ahead Cache*. It reads all sectors of FAT area into RAM on mount time. It requires about 500KB for 2GB FAT32 storage, and this requirement is reasonable cost considering RAM size of recent embedded hardware. When a read operation of FAT area is issued, it reads from the FAT cache instead of the LRU cache nor the device. Like the LRU cache, this cache also operates as a write-through cache. By using this approach, we guarantee the performance of sequential write since completely no hardware read between sequential write happens.

III. EVALUATION

A. Experiment Design

For the evaluation we run periodical sequential write on the test environment which is described in the chapter II. The virtual storage is formatted as FAT32 and mounted in boot time. After it is mounted, whole FAT cache is filled and then the test file is created. The test process writes 50KB sequentially for each 5 seconds, 50 times. Since virtual storage has about 80KB/s speed, each period does not affect other periods. We evaluated the write time of each period.

B. Result

Figure 1 describes the result of the evaluation which is shown above. First of all, without any scheme, the write time shows about 353-462ms. With both metadata and whole FAT cache, however, the write time is reduced to minimum 15ms; about 20x faster. This result proves that the cache maximizes the performance of low-bandwidth storage.

On the other hand, only with the LRU metadata cache, it shows almost same performance with both caches except periods which makes a new FAT sector read. Despite whole FAT read-ahead cache takes large memory size and long read time on boot, they are reasonable trade-offs with the performance gain which could guarantee real-timeness.

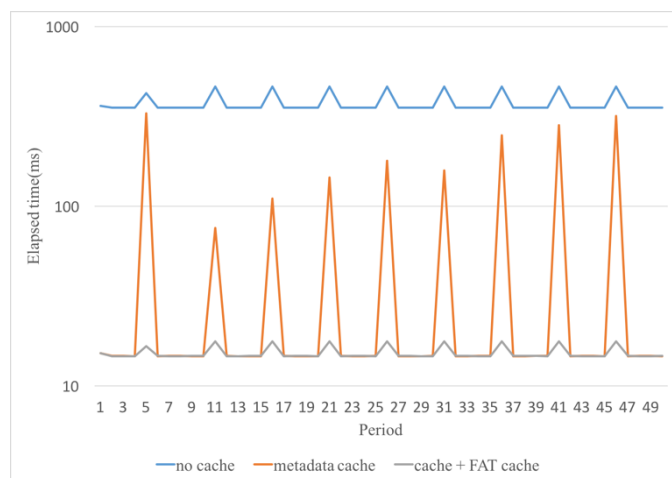


Figure 1. write time of each period

IV. CONCLUSIONS

In this paper we maximize the buffer scheme for low-bandwidth storage by eliminating hardware read between large sequential writes in order to increase performance and guarantee real-timeness. We implement two types of metadata caching approaches, a LRU cache and a whole FAT read-ahead cache, to prevent any cache-misses between sequential writing and show about 23 times of performance improvement.

Hardware and software must be mutually complementary in embedded systems. This work can be one of the example of co-design that software helps a low-performance hardware in order to improve total performance.

Even though the trade-offs for whole FAT read-ahead buffer is reasonable, there is some room of improvement. We read whole FAT area on boot since we cannot predict FAT sectors to be read. If we could predict it, we can reduce the cache size so that we can apply it to embedded systems with less RAM size.

ACKNOWLEDGMENT

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP)(No.R0190-15-2071, Open PNP platform for diversity of autonomous vehicle based on cloud map).

REFERENCES

- [1] "FAT File System (Windows Embedded CE 6.0)", Microsoft. 2010-01-06.
- [2] W. Vogels, "File system usage in Windows NT 4.0". Proceedings of the seventeenth ACM symposium on Operating systems principles, pp 93-109. ACM New York, 1999.
- [3] (2015) RTST website. [Online]. Available: <http://www.rtst.com/>



Jeaho Hwang (BS'07-M'09) received his BS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2007, and MS degree in computer science from KAIST in 2009. He is currently a PhD candidate in the Computer Science Department, KAIST. He is also working as a senior researcher in RTST, which is a startup company developing a real-time operating system. His current research interests include operating systems, system reliability, virtualization and embedded systems.



Kangheui Won (BS'14) received his BS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2014. He worked in start-ups for 3 years before joining as a researcher in RTST, which is a startup company developing a real-time operating system. His current research interests