Evaluation of Ext4 Write Latency for Real Time Systems

RYO OKABE[†1] MASAKATSU TOYAMA[†1]

Abstract: In some embedded systems, the latency of the file system's `write()` operation is required to be on the order of sub-milliseconds. However, the latency exceeds a millisecond when a storage access occurs inside `write()`. Although a pre-allocation step is known to eliminate these accesses, we need to confirm its effect. Furthermore, we need to determine whether there are other `write()` operations whose latency also exceed a millisecond. We evaluated the latency of the Ext4 `write()` operation using pre-allocation. The results show that pre-allocation is effective in eliminating storage accesses. We also found that the latency exceeded a millisecond due to journaling when another file was operated. Further studies are needed to address this latency.

Keywords: Performance of File System and Journaling, Embedded Systems, Linux, fallocate, NAND Flash, eMMC

1. Introduction

Some embedded systems issue periodic I/O requests. When a fault related to I/O occurs in these systems, a log of periodic I/O data before and after the fault occurrence is useful for analyzing its cause. Recording periodic data in response to a fault can be implemented as follows: a certain amount of periodic data is continually buffered in memory. If a fault occurs, the buffered data are first saved in the storage, and then each periodic data item that occurs after the fault is periodically saved one by one.

When a file system is used, data are saved by the `write()` system call. To ensure that periodic data are recorded without any loss, the latency of `write()` must be lower than the cycle time. However, the latency of `write()` can accumulate in different layers such as file systems, journaling layers and firmware in storage devices. In particular, if a storage access occurs inside `write()`, its latency increases enormously.

In this paper, we evaluate the latency of the Ext4 `write()` operation. A test program which simulates file system workloads of the periodic recording is used in our evaluation. This program adopts a pre-allocation approach so that no storage access occurs inside `write()`. Using this program, we determine whether this approach eliminates any storage access inside `write()`. We also determine whether there are any `write()` operations whose latency exceeds a millisecond due to other causes.

2. Related Work

Schemes to reduce the latency of `fsync()` were proposed for consumer devices [1][2]. The latency of `write()` was found to increase considerably when a kernel thread was performing asynchronous I/Os. Therefore, a scheme for reducing this latency was proposed [3]. However, none of them focused on the latency of `write()` when the pre-allocation approach is adopted.

3. Design Issues and Considerations

Generally, `write()` returns as soon as its data are written into the page cache. However, in some cases, storage accesses occur inside `write()`, and they cause an enormous increase in its latency. The latency will get even worse when using NAND flash storage, which often shows an unpredictable high access latency.

Therefore, we need to design the periodic recording such that no storage access occurs inside `write()`. In particular, block allocation inside `write()` is known to access the storage. This storage access can be eliminated by pre-allocating blocks [4].

The other design issues and considerations derived from the use case of the periodic recording is as follows:

- The period of recording time is variable for each fault event, and the final file size cannot be determined in advance. Therefore, we allocate a predefined number of blocks on the basis of the maximum recording time. Unused blocks are truncated in the post-processing of each recording.
- A new fault event may occur as soon as the previous recording is completed. Even in such a case, a recording for the new event must be started immediately.
- Monotonic increase in main memory usage is not acceptable even when multiple fault events occur successively. Therefore, dirty pages in the page cache must be flushed in parallel with `write()` operations.
- Different fault events are recorded in separate files.

4. Design

Figure 1 shows the main sequence of the periodic recording. The high-priority thread first allocates a predefined number of blocks by `fallocate()` system call, and then it periodically performs `write()`. Each time after a certain number of `write()` is called, a lower-priority thread flushes file data by `sync_file_range()`. As post-processing, unused blocks are truncated by the `ftruncate()` system call. In addition, since `sync_file_range()` leaves metadata unflushed, all dirty metadata are flushed by `fsync()` in the post-processing.

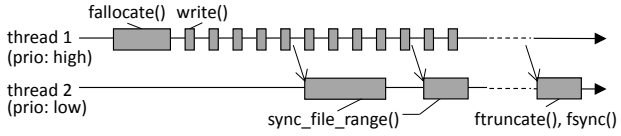

Figure 1: The main sequence of the periodic recording

5. Evaluation

Using a test program based on the design mentioned in Section 4, we evaluated the latency of `write()`. We also evaluated the latency of `fallocate()` because the first calling of `write()` is delayed until `fallocate()` is completed. The purpose of the

†1 Information Technology R&D Center, Mitsubishi Electric Corporation, Kamakura, Japan

evaluation is to confirm whether the latency of these operations can exceed a millisecond due to a storage access or other causes.

The following four tests must be performed. First, the latency of `fallocate()` must be evaluated since it accesses the storage in order to read block group descriptors. Second, the latency of `fallocate()` must be evaluated under a situation where a new recording is started before the post-processing of the previous recording is completed. Therefore, its latency must be evaluated while `ftruncate()` and `fsync()` are called on another file in the background. Third, we must confirm whether the latency of `write()` increases due to calling of `sync_file_range()` on the same file. Finally, the latency of `write()` must be evaluated while `ftruncate()` and `fsync()` are called on another file in the background.

We conducted our evaluation using an embedded board equipped with a dual-core processor running at 1.0 GHz, 1 GB SDRAM, eMMC 4.5 host controller and 8 GB eMMC 5.0 flash storage. Linux 4.4.0 was run on the board. We chose the Ext4 file system since it is used as the file system for NAND flash storage in many embedded systems. The file system was mounted with the *ordered* journaling mode. The *noatime* and *lazytime* mount options were turned on in order to reduce the amount of metadata updates. Furthermore, we assume that `fsync()` is appropriately called by application programs to flush dirty pages. Therefore, we configured the Procfs parameters of virtual memory subsystem to prevent the kernel thread from flushing dirty pages.

We measured the latency of `fallocate()` and `write()` as follows. All dirty pages were flushed at the beginning of each test.

- `fallocate()`: We measured the latency of `fallocate()` 8000 times to calculate its average and worst latency. Each calling of `fallocate()` allocated 100 MB of storage space. We measured the latency with and without the background workload, where `ftruncate()` and `fsync()` were repeatedly called on another file.
- `write()`: We measured the latency of sequential writes on a pre-allocated file. Each sequential `write()` was called every 5 milliseconds, and 16 KB of data were written each time. In addition, `sync_file_range()` was called by a lower-priority thread each time after the written size reached 2 MB. After calling `write()` 8000 times, we calculated its average and worst latency. We measured the latency with and without the background workload, where `ftruncate()` and `fsync()` were repeatedly called on another file.

The measurement results of `fallocate()` are shown in Table 1 and Figure 2. Under no background workload, the latency was 141 microseconds on average and 4299 microseconds at maximum. The worst latency seemed to be caused when the file system read a block group descriptor stored in the storage in order to construct its cache. On the other hand, `fallocate()` took up to 208 milliseconds under the background workload. Although further analyses are yet to be done, the cause of this latency is presumed as follows: metadata accessed by `fsync()` and that accessed by `fallocate()` were stored on the same page. Thereby, `fallocate()` was blocked while accessing the metadata until `fsync()` has completed flushing that page.

The measurement results of `write()` are shown in Table 1 and Figure 3. Under no background workload, no latency spike was observed even when `sync_file_range()` was executed in parallel on the same file. However, under the background workload, `write()` took up to 1043 microseconds. This latency was caused when the `write()` operation was blocked while obtaining a journal handle. The journal handle was presumed to be held by `fsync()`. We had expected that by pre-allocating blocks and withholding timestamp updates with the *lazytime* and *noatime* mount options, any metadata update inside `write()` would be prevented. However, since `write()` tried to obtain a journal handle, some metadata were presumed to be updated inside this operation. Further analyses are necessary to determine whether this latency can be reduced.

Table 1: The latency of the system calls

Background	`fallocate()`		`write()`	
	Avg	Max	Avg	Max
NONE	141µs	4299µs	178µs	359µs
Ftruncate + Fsync	813µs	208440µs	181µs	1043µs

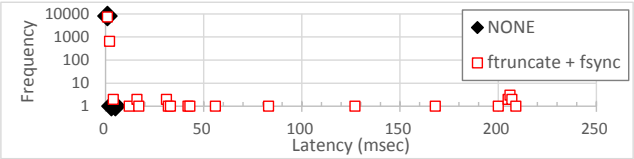

Figure 2: The latencies of the `fallocate()` system call

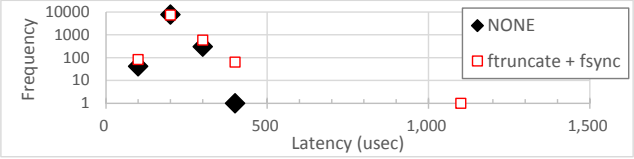

Figure 3: The latencies of the `write()` system call

6. Conclusion and Future Work

In this paper, we evaluated the latency of the Ext4 `write()` system call based on a specific use case. Our results show that by adopting the pre-allocation approach, latency increase due to storage accesses inside `write()` does not occur. We also found that the latency of `write()` exceeded a millisecond due to journaling when `ftruncate()` and `fsync()` were executed on another file. In addition, `fallocate()`, which is necessary for subsequent `write()` calls, showed an enormous increase in its latency when `ftruncate()` and `fsync()` were executed on another file.

Our future work includes reduction of these latencies and evaluation of other file systems.

Reference

[1] Hankeun Son, et al, “Coarse-grained mtime Update for Better fsync() Performance,” SAC '17 Proceedings of the Symposium on Applied Computing, Pages 1534-1541, Apr. 2017.

[2] Yunji Kang, et al, “Per-Block-Group Journaling for Improving Fsync Response Time,” The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014), Jun. 2014.

[3] Daeho Jeong, et al, “Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices,” FAST'15 Proceedings of the 13th USENIX Conference on File and Storage Technologies, Pages 191-202, Feb. 2015.

[4] Ext4: Slow performance on first write after mount. https://www.spinics.net/lists/linux-ext4/msg38336.html (accessed Aug. 18, 2018).