2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA) | 978-1-6654-5078-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/NVMSA56066.2022.00016

Polling Based Per-core Workqueue Management in XFS Journaling

Kwangwon Min[§], Dohyun Kim[§], Seungho Lim and Youjip Won
Department of Electrical Engineering, KAIST, Korea

***Abstract*—In this paper, we analyze the scalability of XFS journaling and improve the many-core scalability of XFS journaling. We found that the lock contention in the async and await mechanisms, such as workqueue and waitqueue, is one of the main causes for the scalability failure in XFS journaling. We propose *per-core pool workqueue* and *polling based on-disk logging* to solve these problems. By using per-core pool workqueue, we resolve the lock contention on thread pool used in the workqueue module. By using polling based on-disk logging, XFS waits for the journal thread to finish in polling based mechanism. We resolve the lock contention on global waitqueue and reduce the latency of on-disk logging through polling. We implement these methods based on XFS. In varmail and exim workloads, the proposed techniques improve the benchmark performance by 57% and 28% against XFS, and 9% and 28% against ScaleXFS, respectively.**

***Index Terms*—Operating System, Filesystem, XFS, Journaling**

I. Introduction

The filesystem journaling ensures that the integrity of the filesystem is not compromised in case of the unexpected system failure [1], [2]. The filesystem journaling writes the log data to the journal region first, *journaling*, and later migrates the log data to its original location, *checkpoint*. With the advancement of hardware systems, the journaling of filesystem has emerged as a major reason for hindering the scalability in many-core systems [3], [4]. Many studies are being conducted with the aim of improving the scalability of filesystem journaling [5]–[7]. They separate centralized journaling data structures to per-core basis [7], or separate journal regions into multiple [6] to improve the many-core environments. Most of the existing studies for many-core scalability of filesystem journaling are for EXT4 filesystem [8]. These techniques cannot be readily applied to the other journaling filesystems such as XFS. ScaleXFS [9] addresses the many-core scalability of XFS filesystem journaling. The studies dealing with the scalability of existing filesystem journaling have mainly targeted specific filesystems, especially EXT4 [10]. XFS is popular filesystem, as it is used as the basic filesystem for Red Hat Enterprise Linux(RHEL) [11] and as the default filesystem for Ceph [12], a distributed storage system.

This paper focuses on resolving the scalability bottleneck of XFS journaling. In particular, we focus on resolving the contention generated by the async and await mechanisms used in XFS journaling procedure. ScaleXFS resolves the main bottleneck of XFS journaling [9]. We find that the contention of the locks on the workqueue and the waitqueue in on-disk logging of ScaleXFS is substantial.

We propose two methods to alleviate those contentions. The first one is enabling per-core pool workqueue. XFS uses a workqueue [13] to trigger the journaling operation. Currently, XFS uses a per-NUMA node pool workqueue. In the per-NUMA node pool workqueue, there is a thread pool, which manages the pre-allocated threads, for each NUMA node. Per-core pool workqueue utilizes the existing Linux kernel's workqueue module, allowing XFS to use per-core pool workqueue, not the per-NUMA node pool workqueue. By adopting per-core pool workqueue, we eliminate the lock contention on the thread pool. Second method that we propose is polling based on-disk logging. XFS uses waitqueue [14] mechanism to wait for the completion of the journal thread and commit thread in on-disk logging. In XFS, all the threads calling `fsync()` try to acquire the single global lock that protects the waitqueue whenever they go to sleep or wake up. In many-core environment that many threads can invoke `fsync()` concurrently, the lock contention on the waitqueue becomes severe. By waiting for the completion of the thread that performed on-disk logging in a polling manner, we alleviate the contention on the lock of the waitqueue. We apply per-core pool workqueue and polling based on-disk logging on most recent variant of XFS, ScaleXFS [9]. With the physical experiment on 112 core machine, our methods improve the performance up to by 24% against ScaleXFS on varmail workload [15], and up to by 28% on exim mail server [16].

II. Background

A. XFS journaling

XFS was developed for Terabyte-scale filesystem supporting full 64-bit filesystem [2]. XFS is the journaling filesystem that supports multi-granularity differential logging [17], [18]. The journaling of XFS consists of two phases; *in-memory logging* and *on-disk logging*. In-memory logging is the process that creates a log data, the copy of modified metadata. The log data is inserted into a global list, *Committed Item List*. In on-disk logging, XFS constructs a memory buffer by copying the log data in the committed item list. Then, it writes this buffer in the journal area on the storage device.

In-memory logging. When a application thread calls the filesystem operation that modifies the metadata, this thread performs in-memory logging. First, it acquires the lock on a metadata and modifies the metadata. Then, it creates the

[§]Both of the authors equally contributed to this work.

978-1-6654-5078-2/22/$31.00 ©2022 IEEE
DOI 10.1109/NVMSA56066.2022.00016

committed item list's entry, which is called *log item*, for each modified metadata. The log item contains the type of metadata and the start address of log data. After creating the log item, the application thread creates the log data by copying the updated region of metadata. For the metadata smaller than 4 Kbyte, the log data becomes the copy of full metadata. For the metadata which is equal to or larger than 4 Kbyte, XFS copies the updated region in the units of 128 bytes.

If the same metadata has been updated more than once before triggering on-disk logging, the log item would be already in the committed item list. The race condition can occur when in-memory logging and on-disk logging access the same log item in this list. To avoid it, before storing the start address of log data, XFS acquires a shared lock (read-write semaphore) to allow other in-memory loggings and block on-disk logging. Then, XFS inserts the log item into the committed item list. The application thread needs to acquire the exclusive lock (spin lock) on the committed item list when it inserts the log item to the committed item list.

On-disk logging. On-disk logging is triggered periodically, when `fsync()` is invoked, or when the log data in the committed item list exceeds 1/8 of the journal region. Default interval for triggering the on-disk logging is 30 sec. XFS executes on-disk logging asynchronously by using the workqueue [13]. On-disk logging is performed on two different threads. We call them *journal thread* and *commit thread*.

The journal thread is executed asynchronously through the workqueue when on-disk logging is triggered. It first acquires an exclusive lock (read-write semahpore) on the committed item list to block in-memory loggings and other on-disk loggings. Subsequently, it creates the on-disk logging context that contains the information of current on-disk logging process. The on-disk logging context has its ID and the list of log data. The context ID is assigned to a value that is one greater than the context ID of the most recently initiated on-disk logging. The log data of log item inserted in committed item list is migrated to the log data list in this context. Then, the current on-disk logging context is inserted into a global list, *Committing List*. The committing list maintains the contexts of ongoing on-disk loggings. The journal thread inserts the created on-disk logging context to the committing list and releases the exclusive lock (read-write semaphore).

The journal thread traverses the log data list in the context and copies all the log data to the in-memory buffer, which is called *Log Buffer*. XFS maintains eight log buffers. Each log buffer has its own waitqueue. Only one of these buffer can be active state and the journal thread only can copy the data to active log buffer. When the active log buffer is full, XFS chooses the new active log buffer in a round-robin manner. Once it finishes copying the log data, the commit record that represents the last log data have to be copied to the log buffer. The commit record has the context ID. In XFS journaling, the commit record has to be copied to the log buffer in the order in which the context ID increases. The on-disk logging context has the indicator of whether the commit record is copied or not. The journal thread checks the indicator of each context in the committing list. If the journal threads needs to wait for the preceding journal thread, it inserts itself to a global waitqueue. The committing list and this waitqueue are protected by a global exclusive lock. After copying all the log data including the commit record, the journal thread dispatches the data in log buffer to the journal region on the storage device with `PRE_FLUSH` and `FUA` flags. Then, it updates the indicator of on-disk logging context and wakes up all the threads inserted in the global waitqueue. The journal thread terminates without waiting completion of the log buffer write.

When the completion interrupt of log buffer write occurs, the commit thread is executed asynchronously through the workqueue. It first acquires the two locks on the log buffer and the committing list, respectively. Then, it cleans up all the data in the log buffer and wakes up the threads in the waitqueue of log buffer. The context of on-disk logging, where the written log buffer contains the commit record, is removed from the committing list. Lastly, the locks are released and the commit thread terminates.

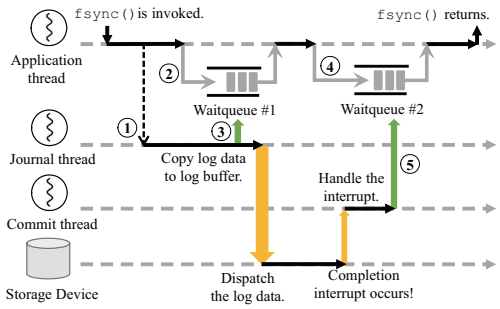


Fig. 1: `fsync()` in XFS.

`fsync()` in XFS. XFS handles `fsync()` in two stages. XFS leaves the processing of journaling to the journal thread and commit thread. The application thread calling `fsync()` waits for the completion of each thread at each step.

First, ① the application thread initiates the on-disk logging by using the workqueue. Then, it waits for the initiated journal thread to copy the commit record to the log buffer. To do this, the application thread first finds the context of on-disk logging by traversing the committing list. Then, ② it inserts itself into a global waitqueue. To avoid race condition, the application thread acquires and releases the lock that protects the committing list and the waitqueue when it accesses the list and waitqueue. Once the journal thread copies the commit record, ③ it wakes up all the threads waiting in the waitqueue.

Second, the application thread waits for the completion of on-disk logging. Through the first step, it can the log buffer in which the commit record of on-disk logging is copied. The start address of log buffer is stored at the context of on-disk logging. ④ The application thread inserts itself to the waitqueue of log buffer. After the data in log buffer is persisted to the storage, ⑤ the commit thread wakes up all the threads waiting in the waitqueue of log buffer.

B. Workqueue

Workqueue is the queue that maintains the functions to execute. It manages the multiple pre-created threads and allocates the queued function to one of those threads. It is often used when asynchronous execution of processes or interrupt handling are required [13]. The workqueue consists of the three main components; work item, thread pool, and pool workqueue. First, the work item is the unit of execution request. It holds the pointer of function to be executed. If a thread needs to execute the function asynchronously, it should first create the work item which has the function pointer to be executed. The thread pool is a pool of multiple threads that waits for the work item to be allocated. These threads are waiting in an IDLE state. Each thread pool has its own queue, *pool workqueue*, which maintains the queued work items. The workqueue keep checking the status of pool workqueue. Once the new work item is inserted into the pool workqueue, the workqueue wakes up one of the waiting threads in the thread pool that owns the pool workqueue. The awaken thread becomes running state and removes the work item from the pool workqueue. Then, it calls the function stored in the work item.

C. Waitqueue

Waitqueue is the queue that maintains the contexts of threads waiting for a certain condition to be true [14]. When a thread needs to sleep until some threads finished their activities, it inserts its thread context to the waitqueue. Then, it yields the CPU. The threads waiting in waitqueue are set to be uninterruptible to be never woken up by CPU scheduling. To wake up the waiting threads, an another thread removes the context in the waitqueue and sets it to be running. Then, waiting threads in waitqueue are woken up by CPU scheduling. Usually, the waitqueue and the variables that can identify the condition are shared across the multiple threads. To prevent race condition among these threads, the other methods, such as a lock, that control the access of multiple threads is required.

III. Motivation

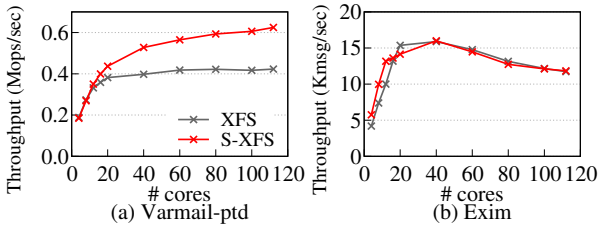


Fig. 2: Throughput of varmail-ptd and exim on XFS and ScaleXFS.

To analyze the many-core scalability of XFS journaling, we conduct an experiment with varying the number of cores. We use 112 core machine (HPE ProLiant DL580, 28 core/socket, 4 sockets, Intel Xeon Platinum 8276) with 512 GByte DRAM. We examine XFS and ScaleXFS [9] on Intel Optane 905p [19] with Filebench varmail [15] and Exim [16] workload. We modify the varmail workload so that each thread operates on its own directory, not the shared one, to eliminate the contention on the shared directory. The exim workload is configured to call `fsync()` for all message deliveries. In Fig. 2, XFS fails to scale when the number of cores is greater than 20. The throughput of XFS saturates or even drops on both workloads. ScaleXFS greatly improves the performance on varmail-ptd workload. However, in exim, ScaleXFS performance drops as XFS does.

TABLE I: Top 5 hottest locks and waitqueues in ScaleXFS.

Variable name	Wait time	P_{all}	P_{lock}
`iclog->ic_force_wait`	437 s	20.3 %	-
`cil->xc_push_lock`	333 s	15.5 %	23.7 %
`cil->xc_commit_wait`	307 s	14.3 %	-
`log->l_iclog_lock`	137 s	6.4 %	9.8 %
`pool->lock`	111 s	5.2 %	8.0 %

To identify the cause for scalability failure in XFS and ScaleXFS, we examine the wait time of various locks and waitqueues in the filesystem by using `lockstat` [20]. The wait time is the time for acquiring the lock or sleeping in the waitqueue. The P_{all} and P_{lock} columns show the percentage of the wait time of single object in the wait time of all objects or all locks, respectively. Table. I shows the top five XFS-related locks and waitqueues with respect to the wait time. We use 112 core machine with Optane SSD. We use modified varmail workload where each thread works on its own directory.

The top five locks and waitqueues on the table are as follows. First, `iclog->ic_force_wait` represents the waitqueue which waits for the log data in the log buffer to be written to the journal region on the storage device. Most of the wait time in this waitqueue is caused by disk I/O to finish. The lock `cil->xc_push_lock` protects the committing list and the waitqueue, `cil->xc_commit_wait`. The third one, `cil->xc_commit_wait`, represents the waitqueue used to wait for the journal thread to copy the commit record. Then, the lock `log->l_iclog_lock` protects log buffer and its waitqueue, `iclog->ic_force_wait`. Lastly, `pool->lock` is the lock besides in the workqueue. It controls the access of thread pool and pool workqueue.

The main lock contention overhead is from the locks that protect the waitqueues. If we do not consider the wait time of waitqueue, which is not the lock overhead, the wait time of `cil->xc_push_lock` and `log->l_iclog_lock` becomes 33.5% of all lock's wait time. When the journal thread or the commit thread terminates, it wakes up all the threads in the waitqueue at a time. In many-core environment and for the journaling-intensive workloads that frequently call `fsync()`, multiple threads try to acquire the same lock on the waitqueue at the same time. Due to this, the lock contention on the waitqueue becomes severe.

The wait time of `pool->lock` is 8.0% of the wait time of all locks. It is similar with the one of the locks for waitqueue. Currently, XFS uses an unbound workqueue that maintains the thread pool for each NUMA node. In this design, the

threads operated on the same NUMA node access the same thread pool to insert the work item. Since multiple threads try to acquire the lock on the thread pool, the lock contention becomes noticeable in XFS journaling.

IV. Design

We identify two lock contentions in XFS journaling. One is the contention among the multiple threads that try to acquire the lock on the thread pool and pool workqueue in the workqueue. The other is caused by waiting for the completion of on-disk logging in the waitqueue. To resolve the overhead caused by workqueue, we propose to use *per-core pool workqueue*. To resolve the overhead caused by waiting in the waitqueue, we propose *polling based on-disk logging*.

A. Per-core pool workqueue

Once a XFS filesystem is mounted, XFS creates the workqueues used for on-disk logging. The workqueues are generated with assigning WQ_UNBOUND flag. If the WQ_UNBOUND flag is assigned, the workqueue creates the thread pools in a per-NUMA node basis. When a thread tries to execute the work asynchronously, it inserts the work item into the pool workqueue located in the NUMA node where it is operating. When the number of cores per NUMA node increases, such as in a server environment, multiple cores access a single pool workqueue. This leads to a situation where threads running on multiple cores compete with each other for acquiring the lock of pool workqueue, `pool->lock`.

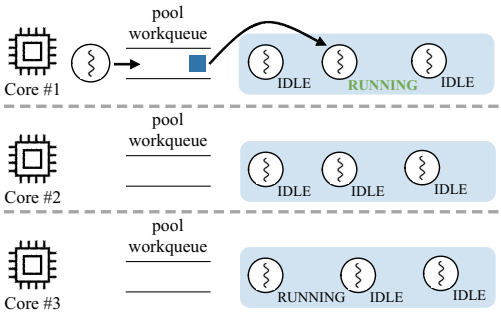


Fig. 3: Structure of per-core pool workqueue.

To alleviate the lock contention on workqueue, we propose to use *Per-core pool workqueue*. If a workqueue is generated without assigning WQ_UNBOUND flag, the workqueue creates the thread pools in a per-core basis. Since each thread pool has its own pool workqueue, there is a pool workqueue for each core. We call this workqueue *per-core pool workqueue*. In per-core pool workqueue, a thread pool and its workqueue is located on each core. When thread try to execute a function asynchronously , it creates and inserts the work item with the function pointer into the pool workqueue located in the core where it is operating. Since the two or more threads cannot access the same pool workqueue at the same time, the lock contention on the workqueue disappears. Also, per-core management for workqueue no longer determines the core to be execute the work item according to workqueue's properties [21]. The procedure of mapping between the work and thread can be passed so per-core pool workqueue executes the work faster than per-NUMA node workqueue.

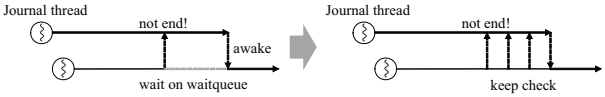


Fig. 4: Structure of polling based on-disk logging.

B. Polling based on-disk logging

XFS uses the waitqueues to block the thread that calls `fsync()` until on-disk logging is done. If the multiple threads share the same waitqueue, the lock contention for accessing the waitqueue may become scalability issue. To avoid this problem, the polling based method can be applied. With the polling mechanism, the waitqueue and associated exclusive lock are no longer needed. However, the polling based method uses busy-wait that occupies a CPU core. To well optimize the await mechanism, both lock contention and wasted CPU cycles should be considered.

To optimize the await mechnism in XFS journaling, we propose *Polling based on-disk logging* that uses the polling based method to wait for the completion of the journal thread. In XFS, the execution of `fsync()` consists of two parts. Each part waits for the completion of each thread's activity in on-disk logging by using the waitqueue. The first part is waiting for the commit record to be copied to the log buffer. Another part is waiting for the data in log buffer to be persisted in the journal region. The latter case is a wait for the completion of the I/O requests to the storage device. Polling based I/O may incur the waste of CPU cycle [22]. Accordingly, in this case, busy-waiting is not appropriate to optimize the mechanism.

For the first part, busy-waiting would waste fewer CPU cycles than the latter part. The journal thread spends most of its time copying the log data to the log buffer. Copying the log data relatively takes small amount of time compared with writing it to the storage device. In practice, the wait time of waitqueue in the first part, `cil->xc_commit_wait`, is less than that of latter part, `iclog->ic_force_wait`, as shown in Table. I. Also, Adopting polling to the first part resolves more significant lock contention and can lead to higher performance benefits. In Table. I, the lock contention on waitqueue of the first part, `cil->xc_push_lock`, is more severs than that of latter part, `log->l_iclog_lock`.

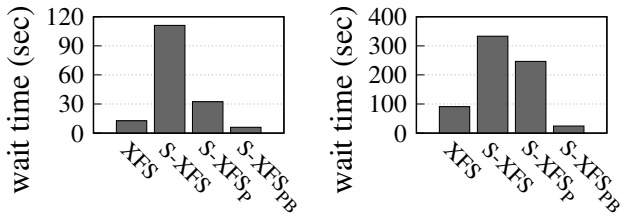
To apply the polling based method, the threads that call `fsync()` keep checking whether the commit record is copied to the log buffer or not. The journal thread also keeps checking the preceding journal thread to copy the commit record. The waitqueue that is required to wait for the journal thread is completely removed. The completion of the commit record copy can be detected by reading the indicator in the context of on-disk logging. The journal thread updates this indicator right after it copies the commit record to the log buffer. The threads

that call `fsync()` and the journal thread should traverse the committing list to find the context to be checked. To avoid the race condition, the exclusive lock of committing list is required. This lock is the same one that protects the waitqueue used to wait for the journal thread. Although polling based on-disk logging cannot eliminate all the contention of lock `cil->xc_push_lock`, it can reduce the lock contention overhead caused by the lock acquisition for the waitqueue.

V. Experiment

A. Experiment Setup

We perform the physical experiment to examine the performance effect of the proposed techniques. For the experiments, we use a 112-core server (HPE ProLiant DL580, 28 core/socket, 4 sockets, Intel Xeon Platinum 8276) with 512 GByte DRAM. We use CentOS 7.4 operating system. We use kernel version 5.8.5. We use Intel Optane 905p NVMe SSD [19] and 256 GByte Ramdisk. We compose four different versions of XFS; XFS(stock XFS), ScaleXFS [2], ScaleXFS$_{P}$ (ScaleXFS with per-core pool workqueue) and ScaleXFS$_{PB}$ (ScaleXFS with per-core pool workqueue and polling based on-disk logging).



(a) `pool->lock` (b) `cil->xc_push_lock`

Fig. 5: Wait time for acquiring the lock.

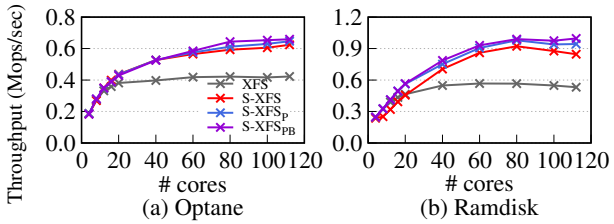
B. Lock Contention

To analyze the effect of per-core pool workqueue and polling based on-disk logging, we conduct an experiment with varmail-ptd workload that each thread operates on its own directories. We modify the varmail workload so each threads works on its own separate directory. We call this modified variant of varmail workload *varmail-ptd*.

Fig. 5a and Fig. 5b shows the wait time of `pool->lock` (pool lock wait time) and `cil->xc_push_lock` (push lock wait time) with 112 threads on 112 cores and Intel Optane SSD, respectively. The wait time of each lock increases in ScaleXFS compared to XFS. The pool lock wait time of ScaleXFS is 8.7× higher than that of XFS. The push lock wait time of ScaleXFS is 3.6× higher than that of XFS. ScaleXFS resolves the main cause of the scalability failure; the contention of locks on committed item list [9]. In ScaleXFS, the overhead of lock contention on waitqueue and workqueue still remains and even worse than XFS.

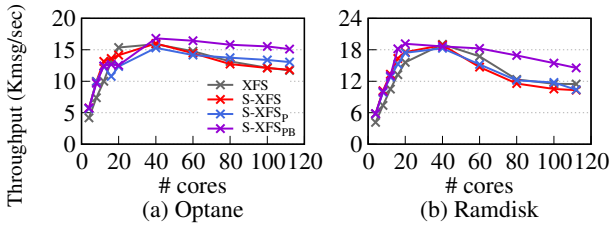
The experiment result shows that the wait time of each lock decreases significantly, compared to XFS and ScaleXFS with our techniques. By applying our techniques, the pool lock wait time decreased by 53% and 95% compared with XFS and ScaleXFS, respectively. Also, the push lock wait time decreased by 73% and 93% compared with XFS and ScaleXFS, respectively. Per-core pool workqueue resolves the contention on the lock of thread pool, `pool->lock`, and polling based on-disk logging resolves the contention on the waitqueue, `cil->xc_push_lock`. Per-core pool workqueues allow on-disk logging to be triggered by leveraging distributed thread pools, indicating that threads running on multiple cores rarely access to the same thread pool at the same time. At the same time, polling based on-disk logging will no longer use waitqueue to wait for the previous journal thread to complete. It eliminates the need for accessing the single global waitqueue and acquire the lock that protects the waitqueue.

C. Benchmark Performance



(a) Optane (b) Ramdisk

Fig. 6: Throughput of varmail-ptd workload.



(a) Optane (b) Ramdisk

Fig. 7: Throughput of Exim workload.

We measure the benchmark performance of four filesystems. Fig. 6 shows the result with varying number of cores. In Optane, ScaleXFS$_{P}$ shows the performance improvement of up to about 4% compared to ScaleXFS. In ScaleXFS$_{PB}$, it shows up to about 9% improvement in performance compared to ScaleXFS and about 6% improvement in 112 cores. When using Ramdisk, the performance increase becomes more significant. For ScaleXFS$_{P}$, there is a performance improvement of up to about 23% against ScaleXFS and about 12% on 112 cores. ScaleXFS$_{PB}$ improves performance up to about 24% compared to ScaleXFS and about 18% on 112 cores.

Fig. 7 shows the results about an exim workload [16], a type of mail server. Exim consists of different types of filesystem operations such as creating, deleting, and renaming small files. The operation of the exim workload consists of the following: receive mail using the SMTP connection, and put it in the shared spool directory one after another. After that, attach it to the per-user mail file, delete the mail, and record this delivery in the shared log file. We use the version of exim server used by FxMark [3]. In order to observe the impact of on-disk

logging in XFS, we configured it to turn on the option to call `fsync()` for each message delivered.

The result shows that ScaleXFS$_{P}$ improves performance by up to about 10% on Optane and up to 12% on Ramdisk over ScaleXFS, respectively. ScaleXFS$_{PB}$ shows performance improvement of up to about 28% on Optane and up to about 47% on Ramdisk compared to ScaleXFS, respectively. In ScaleXFS$_{P}$ and ScaleXFS$_{PB}$, the optimization of on-disk logging of XFS shows performance improvements over XFS and ScaleXFS. It also alleviates the performance collapse at a large number of cores.

In both varmail-ptd workload and exim mail server, ScaleXFS$_{P}$ and ScaleXFS$_{PB}$ show significant performance improvements over ScaleXFS when a large number of cores are operating. This is because the lock contention on the workqueue and waitqueue is reduced by using our methods.

VI. Related Works

The most essential way to address the many-core scalability is changing a single global structure to multiple structures such as a per-core based one. SpanFS [6] separates the file system partition into several units called domains and allows parallel journaling in each domain. Z-journal [7] allocates the transactions and the journal threads in per-core basis and allows journaling to proceed independently at each core. iJournaling [23] effectively reduces the overhead of journal synchronization by introducing file-level transactions and separated journal regions in per-core basis. ScaleFS [24] demonstrates the separation of in-memory and on-disk filesystem and the use of oplog [25] to achieve scalability. Several studies have also proposed the use of optimized lock structures to address the contention generated by the global lock of filesystems. MAX [26] proposes a reader pass-through semaphore using a scheduler to mitigate the contention on the global counter. Son et al. [27] resolve the high lock contention that occurs in the journaling of EXT4 using the lock-free structure.

VII. Conclusion

In this study, we analyze the scalability bottleneck of XFS journaling. We find that lock contention by the async and await mechanism used in on-disk logging of XFS was severe. The associated overhead is caused by the lock contention on the workqueue or the waitqueue. We use per-core pool workqueue for triggering on-disk logging, eliminating the lock contention caused by the thread pool. By adopting the polling based method to the waiting mechanism, we eliminate the lock contention on the global waitqueue and alleviate the overhead of the corresponding lock. We improve the performance up to 1.3× compared with ScaleXFS with Intel Optane SSD by optimizing the use of workqueue and waitqueue in XFS journaling.

Acknowledgements. This work was in part supported by IITP, Korea (No. 2018-0-00549), NRF, Korea (No. NRF-2020R1A2C3008525), and SNU-SK Hynix Solution Research Center (S3RC) (No. MOUS002S).

References

[1] S. C. Tweedie *et al.*, "Journaling the linux ext2fs filesystem," in *Proc. of Annual Linux Expo*. Durham, North Carolina, 1998.
[2] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system." in *Proc. of USENIX ATC*, vol. 15, 1996.
[3] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *Proc. of USENIX ATC*, 2016, pp. 71–85.
[4] J. Kang, C. Hu, T. Wo, Y. Zhai, B. Zhang, and J. Huai, "Multilanes: Providing virtualized storage for os-level virtualization on manycores," *ACM TOS*, vol. 12, no. 3, pp. 1–31, 2016.
[5] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho, "Barrier-enabled io stack for flash storage," in *Proc. of USENIX FAST*, 2018, pp. 211–226.
[6] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai, "Spanfs: A scalable file system on fast storage devices," in *Proc. of USENIX ATC*, 2015, pp. 249–261.
[7] J. Kim, C. Campes, J.-Y. Hwang, J. Jeong, and E. Seo, "Z-journal: Scalable per-core journaling," in *Proc. of USENIX ATC*, 2021, pp. 893–906.
[8] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proc. of Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.
[9] D. Kim, K. Min, J. Oh, and Y. Won, "Scalexfs: Getting scalability of xfs back on the ring," in *Proc. of USENIX FAST*, 2022, pp. 329–344.
[10] C. Hellwig, "Xfs: the big storage file system for linux," *; login:: the magazine of USENIX & SAGE*, vol. 34, no. 5, pp. 10–18, 2009.
[11] "Red hat enterprise linux," https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux.
[12] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. of USENIX OSDI*, 2006, pp. 307–320.
[13] T. Heo and F. Mickler, "Concurrency managed workqueue (cmwq)," 2010, https://www.kernel.org/doc/Documentation/core-api/workqueue.rst.
[14] R. Russell, "Unreliable guide to hacking the linux kernel," https://www.kernel.org/doc/Documentation/kernel-hacking/hacking.rst.
[15] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *USENIX; login*, vol. 41, no. 1, pp. 6–12, 2016.
[16] "Exim," https://www.exim.org.
[17] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song, "Page-differential logging: an efficient and dbms-independent approach for storing data into flash memory," in *Proc. of ACM SIGMOD*, 2010, pp. 363–374.
[18] J. Lee, K. Kim, and S. K. Cha, "Differential logging: A commutative and associative logging scheme for highly parallel main memory database," in *Proc. of ICDE*. IEEE, 2001, pp. 173–182.
[19] "Intel optane 905p," https://www.intel.com/content/www/us/en/products/sku/129833/intel-optane-ssd-905p-series-1-5tb-12-height-pcie-x4-20nm-3d-xpoint/specifications.html.
[20] "Lockstat," https://www.kernel.org/doc/html/latest/locking/lockstat.html.
[21] "Workqueue flags and constants." https://elixir.bootlin.com/linux/latest/source/include/linux/workqueue.h.
[22] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt." in *Proc. of USENIX FAST*, vol. 12, 2012, pp. 3–3.
[23] D. Park and D. Shin, "ijournaling: Fine-grained journaling for improving the latency of fsync system call," in *Proc. of USENIX ATC*, 2017, pp. 787–798.
[24] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scaling a file system to many cores using an operation log," in *Proc. of SOSP*, 2017, pp. 69–86.
[25] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Oplog: a library for scaling update-heavy data structures," MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, NA, Tech. Rep., Sept. 2014.
[26] X. Liao, Y. Lu, E. Xu, and J. Shu, "Max: A multicore-accelerated file system for flash storage," in *Proc. of USENIX ATC*, 2021, pp. 877–891.
[27] Y. Son, S. Kim, H. Y. Yeom, and H. Han, "High-performance transaction processing in journaling file systems," in *Proc. of USENIX FAST*, 2018, pp. 227–240.