Scalability Analysis of F2FS for Concurrent Writes on Shared Files on Manycore servers

Sunghyun Noh and Youngjae Kim
Department of Computer Science and Engineering
Sogang University, Seoul, Republic of Korea
{nsh0249, youkim}@sogang.ac.kr

서강대학교 SOGANG UNIVERSITY

Systor2019 June 3 - 5 Haifa, Israel

Introduction

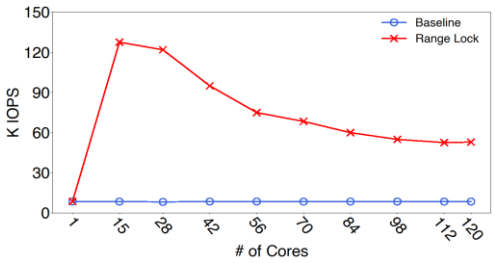
- In F2FS, concurrent write requests to a file are serialized by inode mutex lock.
- If the writing ranges of each thread do not overlap, the file system may allow parallel writes.
- In our previous work[1], we allowed parallel writes to files using Range-Lock instead of inode mutex lock, but the performance increased only up to 15 cores out of 120 cores.
- We analyze the causes of these performance scalability limitations and suggest ideas to reduce the lock contention overhead in F2FS that occurs when searching for a direct node in the page cache.

Motivation

Experimental Setup

- We evaluate F2FS with Range-Lock on a testbed equipped with 120 cores and 740GB DRAM.
- We used a 400GB Intel 750 NVMe SSD. (Sequential I/O bandwidth: 900 MB/s, Random IOPs: 230 KIOPS)
- We evaluated the DWOM workload using a FxMark benchmark tool with direct IO.
  - ✓ In a DWOM workload, multiple threads write to non-overlapping areas of the same file.
- We use *perf* to profile the result.

F2FS Scalability



- Throughput of F2FS with Range-Lock does not scale after 15 cores.

Overhead Profiling

- CPU cycles consumed by the ***get_dnode_of_data*** function has drastically increased after 15 cores.
- We define $Overhead\ ratio$ as follows.

$$Overhead\ ratio = \frac{Total\ cycles\ consumed\ by\ \textbf{get\_dnode\_of\_data}}{Total\ cycles\ consumed\ by\ write\ function}$$

# of Cores	15	42	70	98	120
Overhead ratio	16%	49.3%	66.4%	68.3%	67.6%

- ***get_dnode_of_data*** occupies more than half of the total CPU cycles consumed by an entire write function after 15 cores.
- To mitigate performance bottleneck, we analyzed the behavior of ***get_dnode_of_data*** function.

Direct Node Search in F2FS

- ***get_dnode_of_data*** function has the purpose of getting the direct block information of the corresponding data.
- This function iteratively searches for a node log entry in the page cache and finds the direct node of the data.

Procedure of *get_dnode_of_data*

- Assume that ***get_dnode_of_data*** retrieves direct node of data block A.

Step 1 Find an inode page in the page cache and acquire an exclusive lock on that page. (See *A* in Figure)

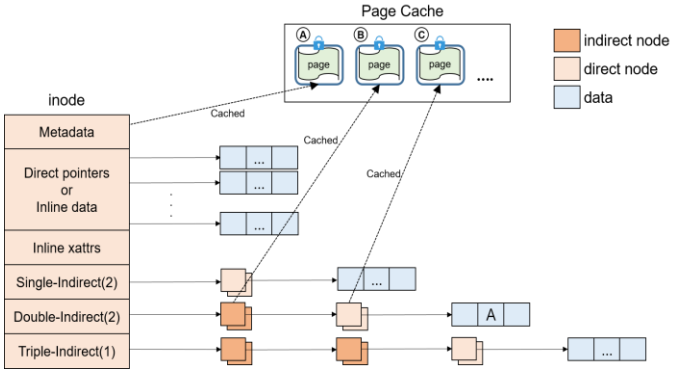
Step 2 Because inode does not directly reference the data block A, the lock to the inode page is released

Step 3 Find an indirect node referenced by the inode from the page cache

Step 4 Acquire an exclusive lock on an indirect node. (See *B* in Figure)

Step 5 Until the direct node which directly references the data block A is found, repeat **Step 2~4.** (See *C* in Figure)

Step 6 After finding a direct node, the lock to the direct node is released. Then only required information of direct node is returned to the caller.



Summary and Future Work

- When performing concurrent writes, a page-level lock at the page cache cause lock contention which seriously degrades file system scalability.
- ***get_dnode_of_data*** requires an exclusive lock on each page from the page cache to prevent pointer changes during direct node searching.
- However, ***get_dnode_of_data*** returns only required information without modification in direct node.
- Therefore it is possible to consider to employ a shared read lock on each node page while searching.
- To mitigate the lock contention occurred in ***get_dnode_of_data***, we will implement a page cache that uses a reader/writer-aware lock for each page entry.

[1] C. Lee et al. "Write Optimization of Log-structured Flash File System for Parallel I/O on Manycore servers." The 12th ACM International Systems and Storage Conference (Systor 19).

Acknowledgement
This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2014-0-00035, Research on High Performance and Scalable Manycore Operating System).