Study and Improvement of page cache utility for cloned files in Btrfs File System

Chitwan Goel (210295)

Geetika (210392)

Parinay Chauhan (200667)

Shrey Bansal (210997)

Course: CS614 (Linux Kernel Programming)

Supervisor: ***Debadatta Mishra***

Introduction

The Btrfs file system stands as a testament to innovation in the realm of storage solutions, boasting a robust Copy-On-Write (CoW) mechanism at the block layer. This CoW implementation has enabled Btrfs to excel in various aspects of data management, particularly in the realm of file cloning. However, a notable disparity exists within Btrfs: while its CoW prowess extends to the block layer, it does not encompass the page cache layer. This limitation serves as the impetus for our project, where we endeavor to augment Btrfs with page-cache sharing functionality for cloned files. Our journey commences with a comprehensive examination of both Btrfs codebase and the Virtual File System (VFS) layer and their Documentation, where we delve deep into the intricacies of their architecture. With a keen focus on understanding not only the individual implementations but also the interactions between these vital layers of the Linux kernel, we tried to gain a holistic understanding of the inner workings of Btrfs and the VFS layer, laying the groundwork for our motive to augment Btrfs with page cache sharing functionality for cloned files. Through meticulous code study and analysis, along with some simple experiments, we unearth the nuances of the existing implementations and identify the challenges hindering the extension of CoW functionality to the page cache layer. Equipped with this valuable understanding, we developed a customized plan to address these hurdles and facilitate the smooth integration of page cache sharing into Btrfs. This report aims to provide a thorough overview of the challenges we faced, our strategies, the details of our implementation plan, and the results we obtained through thorough testing and assessment. In documenting our journey, we aim to not only shed light on our project's technical intricacies but also underscore its significance in optimizing Btrfs for enhanced performance, efficiency, and versatility in modern computing environments.

Analysis and Findings

Evidences for the Existing Implementation

After conducting a thorough code study, we found no evidence of page cache sharing through known mechanisms such as page refcounting, copying, or just-in-time reading through parent-child caches. Furthermore, our investigation revealed documentation from Btrfs indicating that sharing at the page cache layer is not implemented.

To confirm this, we performed a simple experiment. The steps of the experiment included creating a large file ($\sim$1.1 GB in size), cloning it into a child file, and subsequently reading both the parent and child files. Upon analysis of the results, we observed that the time taken to read the parent file was minimal. However, when reading the child file for the first time, there was a significant delay, indicating that even when the parent file was in the cache, the child file was read from the disk. Subsequent reads of the child file showed a reduction in the time taken. These results strongly suggest that page cache sharing is not implemented in the default Btrfs implementation.

Challenges in implementing page cache sharing

Implementing page cache sharing for cloned files in Btrfs presented several significant challenges. Firstly, Btrfs inherently lacks an explicit parent-child relationship for cloned files. Instead, it merely maintains a reference count for each block, complicating the task of tracking shared data.

Additionally, ensuring the persistence of this relationship across system reboots and crashes posed a considerable challenge. Storing large data structures, arrays, or bitmaps within on-disk inodes, even with extended attributes (xattrs), was not feasible due to alignment and size constraints. With only constant-sized attributes available, we had to devise alternative methods to maintain essential information within these limitations.

Furthermore, operating at the page level to copy cache content introduces complexities and the need to manage concurrency issues. Dealing with these challenges required careful design and implementation to achieve reliable and efficient

```
test@test1:~/btrfs$ cp --reflink big2file.txt big3.txt
test@test1:~/btrfs$ free -m
               total        used        free      shared  buff/cache   available
Mem:            3928         559        1077           0        2292        3307
Swap:           1162          22        1140
test@test1:~/btrfs$ time cat big2file.txt > /dev/null

real    0m0.250s
user    0m0.002s
sys     0m0.248s
test@test1:~/btrfs$ free -m
               total        used        free      shared  buff/cache   available
Mem:            3928         559        1077           0        2292        3303
Swap:           1162          22        1140
test@test1:~/btrfs$ time cat big3.txt > /dev/null

real    0m1.080s
user    0m0.004s
sys     0m0.418s
test@test1:~/btrfs$ free -m
               total        used        free      shared  buff/cache   available
Mem:            3928         556          44           0        3327        3306
Swap:           1162          22        1140
test@test1:~/btrfs$ time cat big3.txt > /dev/null

real    0m0.214s
user    0m0.001s
sys     0m0.213s
test@test1:~/btrfs$ free -m
               total        used        free      shared  buff/cache   available
Mem:            3928         557          43           0        3327        3305
Swap:           1162          22        1140
test@test1:~/btrfs$
```

Figure 1: Results of the timing experiment

page-cache sharing functionality within Btrfs.

Solution Strategies

From maintaining metadata to incorporating concurrency and locking, we had to probe into various sites in the codebase for various use cases and user commands. The in-detail implementation changes for each of these use cases is presented below-

Structures augmented

We added some structures in the VFS layer inode `struct inode *`, for page cache sharing purposes.

```
struct inode {
    ...  // all default members
    struct inode *parent;
    char *p_path;
    struct shared_bitmap par_child_bitmap;
    struct inode *children[MAX_CHILDREN];
};
```

```
struct shared_bitmap {
    u64 bitmap_size;
    char *bitmap_array;
};
```

`shared_bitmap` is the structure that tracks which file offsets are shared with its parent. `bitmap_size` stores the size of a parent at the time this particular clone was made, and it is effectively the size of the bitmap array. The bitmap array has entry `1` if the page corresponding to that index is shared with the parent and `0` otherwise. We embedded 3 more members in `struct inode` along with this structure. `parent` and `p_path` are pointers to the parent inode and the path of the parent file, respectively. `children` is an array of inodes, statically allocated (`MAX_CHILDREN` is defined as 16, indicating the maximum number of cloned children a file can have), and it stores inode pointers for all existing cloned children.

Cloning files

When a file is cloned, we initialize the structures in the child and the parent inode. In the parent inode, we iterate through the `children` array until we reach the maximum allowed number of children `(MAX_CHILDREN)`. During this iteration, if we encounter a null index (`children[i] == NULL`), we assign the child inode to that index, effectively implementing a round-robin allocation strategy. On successfully setting one of the indices equal to the child inode, in the child inode, we set its `parent`, `p_path`, and the `par_child_bitmap`. The `bitmap_size` in this struct is set equal to the size of the parent file at the time of clone, and the `bitmap_array` is allocated equal to that size. All pages are shared with the parent at the clone time, so the `bitmap_array` is all 1's.

Reading the child file

When a read syscall is issued to access a cloned file within the range `start` to `end`, we first examine the `bitmap_array` of the child and partition the range `[start, end]` into segments comprising continuous sequences of 1's (indicating shared pages) and 0's (indicating unshared pages).
For segments representing shared offsets, we read through the parent through a newly defined function `filemap_get_pages_parent` which looks up the page cache of the parent for corresponding indices, returns the pages if found in cache and even if the `folios` for those indices are not in cache, it reads the content from the disk (disk contents are anyways shared for clean offsets between child and parent by default in Btrfs) and puts in the parent page cache before returning them.
On the other hand, segments representing unshared pages prompt a call to the conventional `filemap_get_pages` function. This function operates within the context of the child cache, retrieving pages from there or, if absent, reading them directly from the disk.

Writing to child file

During a write operation targeting the range `start` to `end` of a cloned file, we perform an operation to unset the corresponding bits in the `bitmap_array` of the child. This action signifies that these pages are no longer shared with the parent file and, henceforth, are to be read directly from the child.

Writing to parent file

During a write operation targeting the range `start` to `end` of a parent file, we iterate over the `children` array. For each existing child, we invoke the `btrfs_copy_page_cache` function (newly defined). This function is responsible for managing the synchronization of page caches between parent and child files within the specified range `[start, end]`.
In detail, the `btrfs_copy_page_cache` function traverses the range and examines the indices previously set in the `bitmap_array` of the child inode. For each such index, after unsetting the bit, it attempts to retrieve the corresponding folio from the parent's page cache. If the folio is found and is `up-to-date`, the function allocates a new folio and inserts it into the child's page cache. Subsequently, it copies the content from the parent's page into the newly allocated folio within the child's cache. This meticulous process ensures the efficiency of subsequent reads from the child file, as the content is readily available in its cache.

Removing/Truncating child file

When removing/truncating a cloned file, firstly, we iterate over the `children` array of its parent inode. During this iteration, we locate the index `i` such that `children[i]` points to the inode of the child file to be removed. Once identified, we set `children[i]` to `NULL`, effectively removing the reference to the child file from the parent's children array.
Additionally, we perform cleanup operations on the child inode. This includes freeing the allocated memory for the `bitmap_array` and any associated structures such as `p_path` and other relevant data.

Removing/Truncating parent file

When removing or truncating a parent file, a meticulous process is undertaken to manage its cloned children. We call the `btrfs_copy_page_cache` function for each existing cloned child for the range spanning from 0 to the end of the parent file. This function mirrors the cache handling process akin to that of writing to the parent file, ensuring synchronization of cache contents across parent and child files.
Subsequently, we execute cleanup operations for each cloned child. This frees the allocated memory for the `bitmap_array`, `p_path`, and any associated structures. Furthermore, we set the parent inode pointer of each child to `NULL`, effectively disassociating any parent-child relationship.

Locking Mechanisms

Implementing concurrency introduced numerous challenges. To ensure coherence and correctness, we employed locks (inode locks) at critical points whenever reading or writing to the inode. We used the `inode_lock` or `inode_lock_shared` function to acquire locks on the inode. `inode_lock` basically takes writer lock on the read-write-semaphore `i_rwsem` present in the inode. On the other hand, `inode_lock_shared` takes a read lock on `i_rwsem`. For instance, in the `filemap_read` function, we call `inode_lock_shared` on the child inode since we had to read the bitmaps in order to direct the read to the appropriate function, and in `btrfs_copy_page_cache`, we call `inode_lock` on the relevant child inode entry in the `children` array. Similar locking mechanisms were implemented during file removal or truncation operations.
Despite these meticulous precautions, we encountered a major and difficult-to-resolve deadlock scenario. This deadlock occurred when attempting to delete a child file and concurrently write to its parent. The issue arose from a circular dependency: during parent file writing, we needed to lock the parent inode and take the child lock when copying the page cache to it, and subsequently, when deleting the child file, we required a lock on both the child and parent inodes. However, a deadlock ensued if the child file had already acquired its lock and attempted to obtain the parent lock.

To address this deadlock, we implemented a solution wherein when the child file acquires its lock for deletion, it also attempts to acquire the parent lock. If both locks are successfully acquired, the child file proceeds with its deletion. However, if the parent lock is not available, the child releases its own lock and re-attempts to acquire both locks in a continuous loop (`while(true)`). This iterative approach allowed the parent file to acquire the child lock during the cache copying process, and subsequently, the child file could acquire both locks and proceed with its deletion, resolving the deadlock issue.

```
while(true) {
    inode_lock(target);    // target is child
    if (target->parent) {
        if (inode_trylock(target->parent))
            break;
        else {
          inode_unlock(target);
          continue;
        }
    }
    else
        break;
}
.... // further code for child deletion
if(target->parent) {
    inode_unlock(target->parent);
}
inode_unlock(target);
```

In addition to the aforementioned efforts, we tried minimizing overhead from locking mechanisms. To achieve this, we strategically employed locks only where necessary and released them promptly when no longer required. For instance, during the reading process of a child file, within the `filemap_read` function, we

acquired the inode lock for the child inode being read. Subsequently, in scenarios where shared chunks of pages are read within `filemap_get_pages_parent`, if these pages are not found in the parent page cache and necessitate retrieval from disk, we release the lock held on the child inode during the disk read operation. Before reacquiring the lock, the state of the bitmap is checked again. If there are any changes in the bitmap state, the function returns with all the folio that have already been copied otherwise the copy function proceeds as before. This approach avoids unnecessary serialization, thereby minimizing potential overhead attributed to locking mechanisms.

Persistent book-keeping of the parent-child relationship

While the strategies outlined in the preceding sections address the management of virtual layer inodes during runtime, ensuring the persistence of the parent-child relationships is essential for system stability and integrity. In scenarios such as system reboots, crashes, or eviction of inodes from memory under high load conditions, restoring these relationships upon inode restoration is imperative. This section gives an overview of the methods we adopted to restore the relationships.

We utilize extended attributes (xattrs) to store additional persistent information with disk inodes. For each cloned file, we maintain two xattrs: the parent inode number (referred to as `parent_ino`) and the size of the bitmap array (referred to as `bmap_size`). Additionally, for the parent file, we maintain a single xattr, representing the maximum size of the bitmap among all its child files (referred to as `bmap_size`). During the cloning process, the xattrs for the child file (parent inode number and bitmap size) are set up accordingly. Simultaneously, the parent file's xattr `bmap_size` is updated to reflect the current maximum bitmap size among its children or set up if not already present.

Two scenarios can occur when an inode is loaded from the disk into the cache. If parent file P has k children $C_1, C_2, .., C_k$, either P is brought first, or one of $C_i's$ is brought first. If the latter is the case, then we call the `btrfs_iget` function to get P before any modifications are made to C_i. This is because, if P is brought later on, there is no way to establish parent-child relationships for previously loaded children (since we don't store child inode numbers in the parent xattrs).

So, in bringing P to the memory, we allocate its `bitmap_array` size equal to $P's$ `bmap_size` and initialize it to all 1's. The corresponding bits are unset when a write occurs to P. The purpose of this array is explained below.
Suppose C (one of the $C_i's$) is loaded after P is in memory. In this scenario, we retrieve P using the `iget` function based on the `parent_ino`. Additionally, we obtain the `p_path` using the `path_from_inode` function and assign it to C. Subsequently, using the xattr `bmap_size`, we set the `bitmap_size`, allocate the `bitmap_array` in C, and initialize it to all zeros. Then, we utilize several Btrfs ioctl(s) to obtain the inodes with reference for each block and process the outputs to populate the bitmap in the child.
The previously described setup of the `bitmap_array` in P becomes relevant here. If any writes had occurred to P when it was in memory before C was loaded, it is possible that these writes were only in the cache and had not yet been written back. As a result, the bits in the bitmap of C that we constructed above may have been set for corresponding positions. However, our approach ensured that the corresponding bits in the parent bitmap were unset. Thus, to maintain correctness and consistency, we perform a bitwise AND operation for the entire child `bitmap_array` with that of the parent. This exercise zeroes down the chance of any dirty offsets being shared between the parent and the child.
After completing these operations, we set $C's$ parent as P, allocate an index for C in $P's$ children array, and perform any other necessary metadata setup.

Results

We conducted extensive testing across various test cases, including multi-threaded scenarios such as reading-writing, writing-writing, writing-deletion, copy and write, and others. To rigorously validate the correctness of our solution, we employed a combination of C/C++ programs and manual inspection of outputs, ensuring consistency across various file sizes and operations. After confirming correctness, coherence, and the absence of deadlock scenarios, we bench-marked our solution using fio (Flexible IO Tester). We concurrently executed various workloads on parent and child files, employing multiple threads for each job type.

Subsequently, we analyzed the outputs and plotted the completion latencies for each job type, resulting in the following outcomes.
Note: In the diagrams, the green and red lines represent the latencies in the modified and original systems, respectively. We ran 4 parallel jobs for each workload in all of the benchmarks using the `libaio` *IO engine. For example, in the first case, we ran 4 readers each for parent and child and took the mean of the latencies of these 4 parallel jobs.*

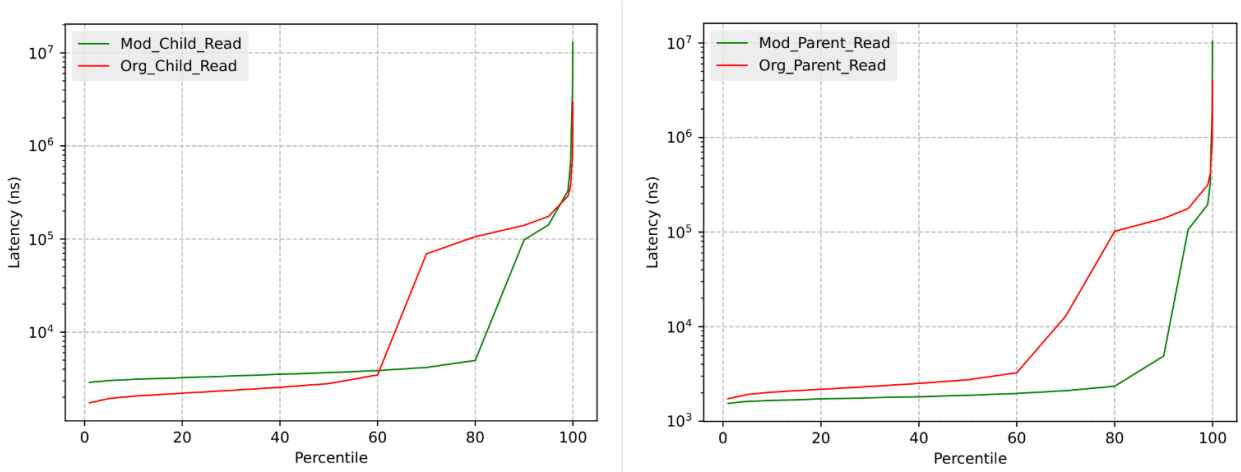

Figure 2: Comparison of latencies, with both parent and child reading

In the above case, when both the parent and child are reading, the modified kernel clearly performs much better than the original kernel. This is because when the child is reading, it reads through the parent's cache and only brings content from the disk into the parent cache. Moreover, the parent itself brings content into its cache. So, the data brought in the cache helps both parent and child in subsequent reads since both can read from the same cache.

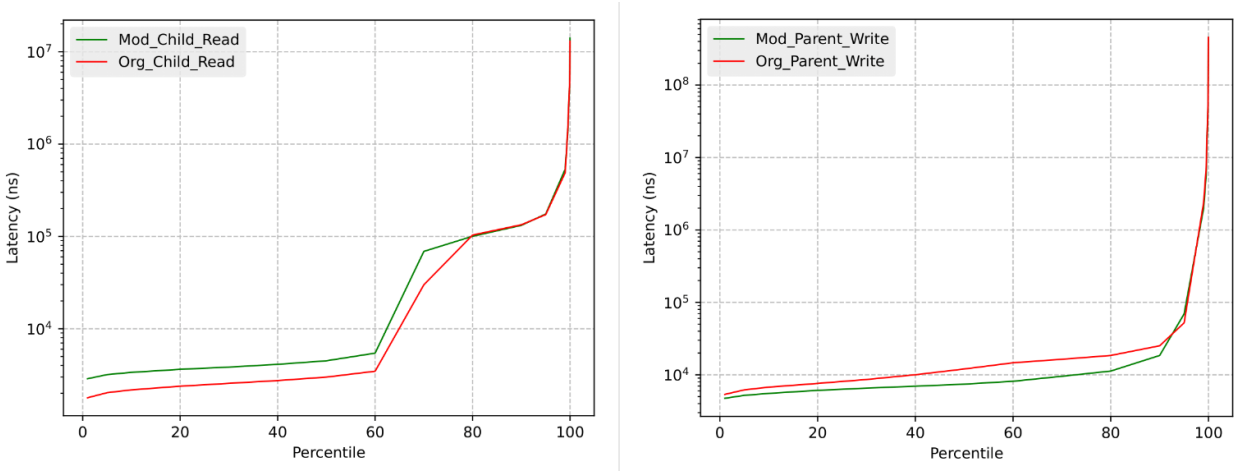

Figure 3: Comparison of latencies, with child reading and parent writing

In the depicted figure, the increased latency for the child file observed in the modified system can be primarily attributed to the fragmentation of bitmaps resulting from writes to the parent file. This fragmentation introduces additional overhead during child reading operations, as the presence of more bitmap chunks necessitates additional processing. Furthermore, acquiring locks while reading the child file exacerbates this overhead, as locking and unlocking operations incur additional computational costs. On the contrary, a plausible explanation for the decrease in parent latency could be attributed to the pre-fetching of pages into the parent cache through child reading operations. Consequently, if only a partial block write is required during parent writing processes, the parent file system can access the necessary pages directly from its cache, thereby eliminating the need to retrieve data from the disk.

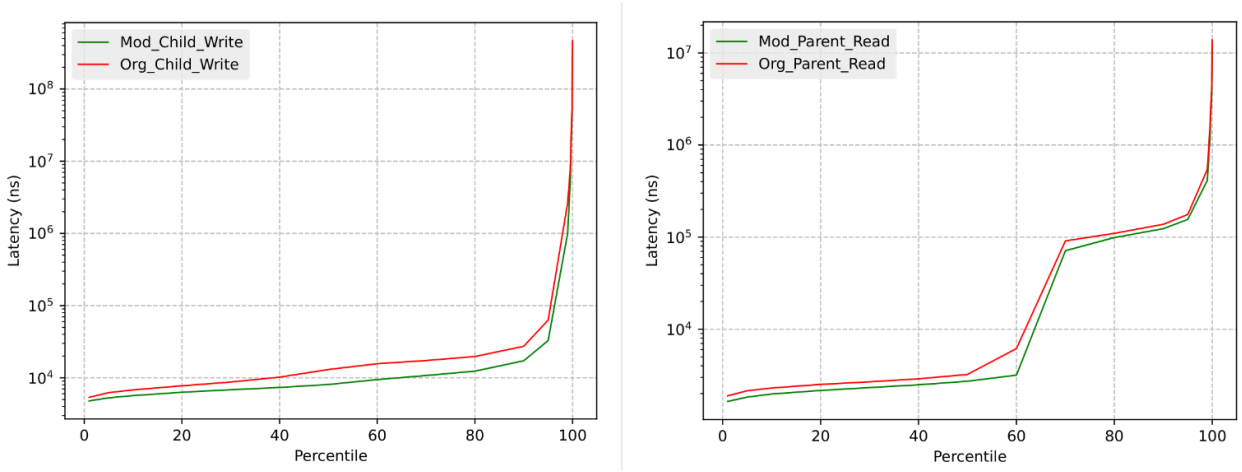

Figure 4: Comparison of latencies, with child writing and parent reading

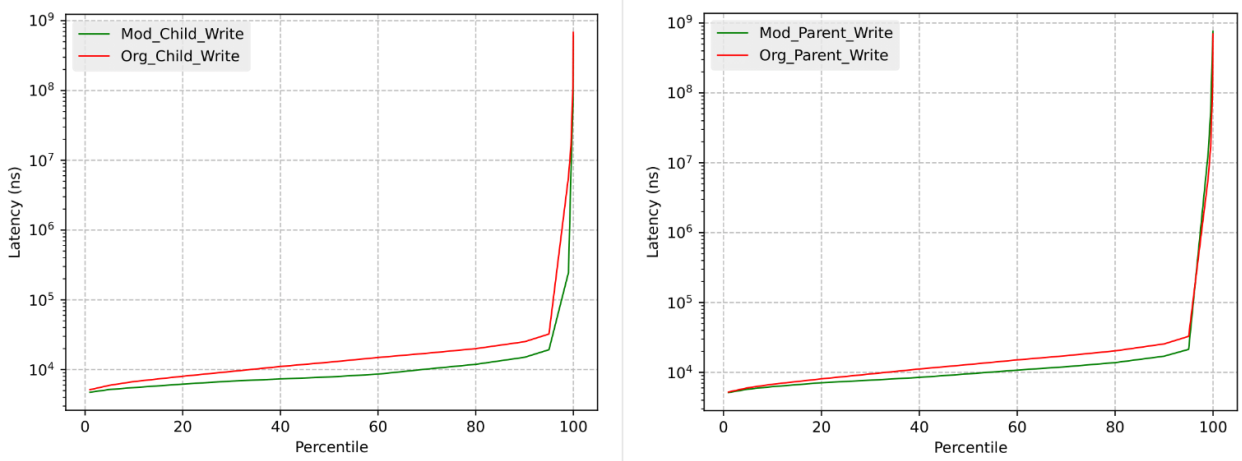

Figure 5: Comparison of latencies, with both parent and child writing

We don't clearly understand the reason for the improvement in latencies for the above 2 cases. Intuitively, due to writing in the parent, the pages are copied into the child's cache, which may act as a pre-fetcher for the pages when a child is written into. This may explain the slight decrease in latencies in the above case.

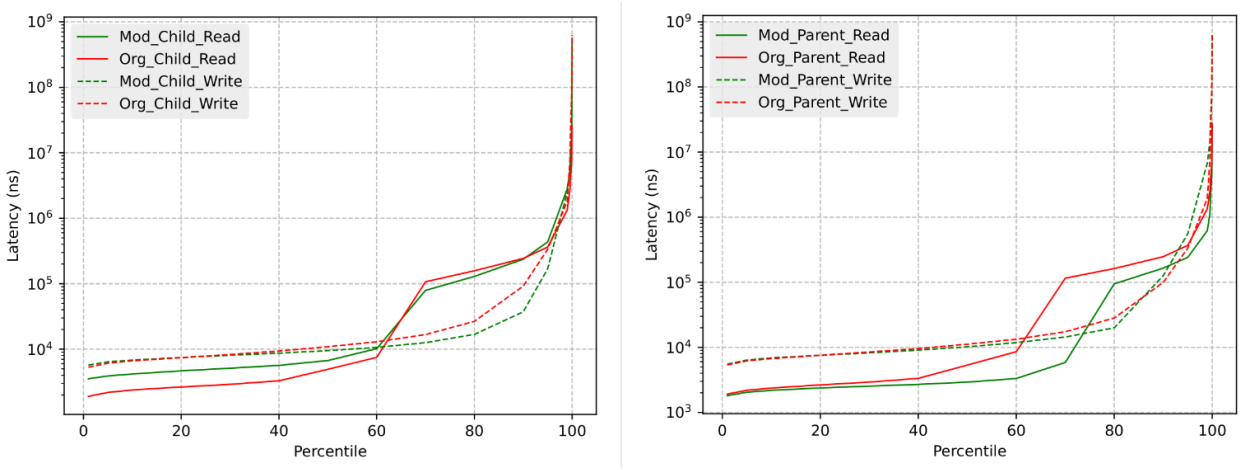

Figure 6: Comparison of latencies, with both child and parent reading and writing

Scalability Tests

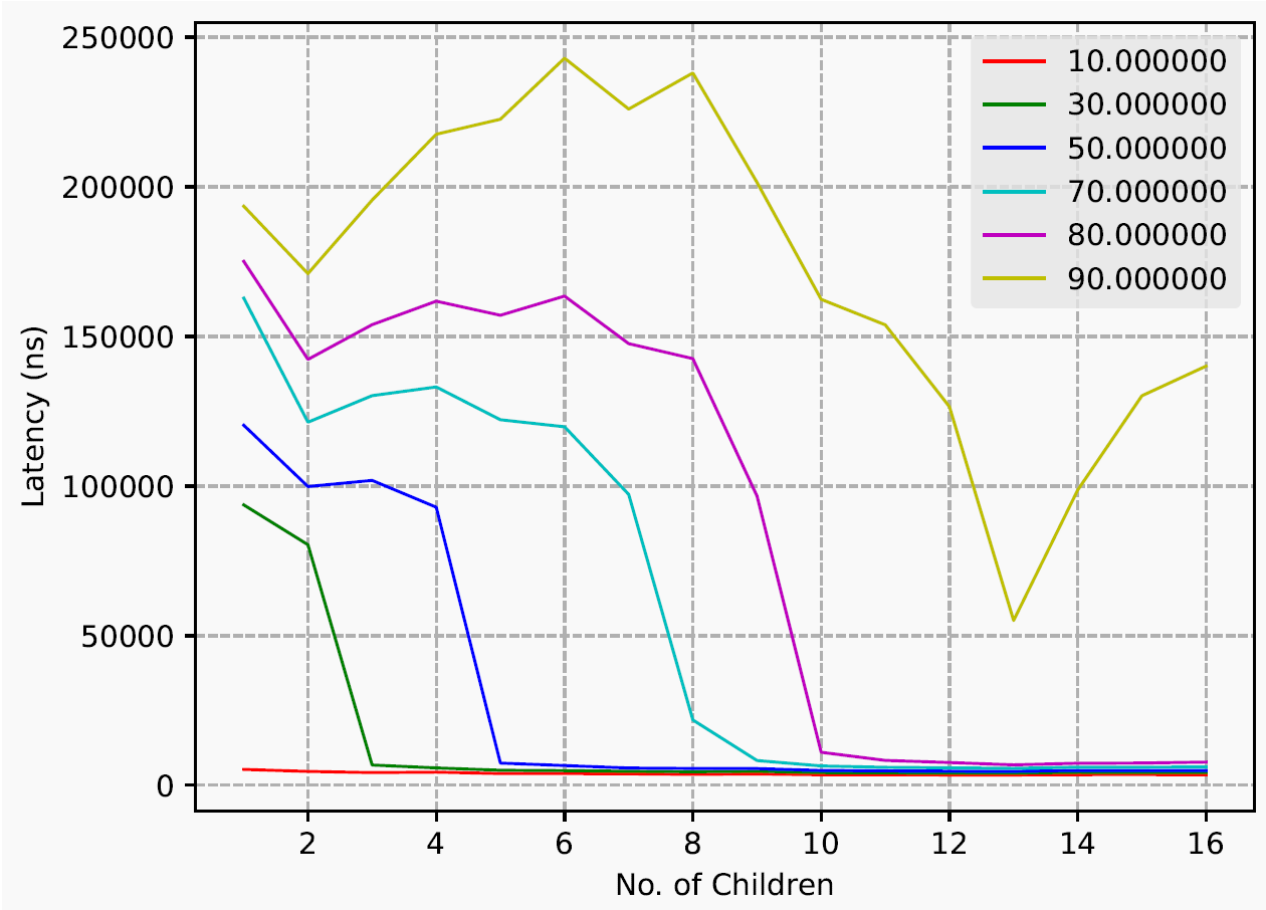

Figure 7: Average Read Latency with multiple children reading and writing

In the above diagram, the latency decreases for 2 children as compared to 1, because one of the children brings content in cache and may be utilised by the other child. Again due to contentions as the children increases, latency increases and later on the reduction may be attributed to high cache flood by large number of children.

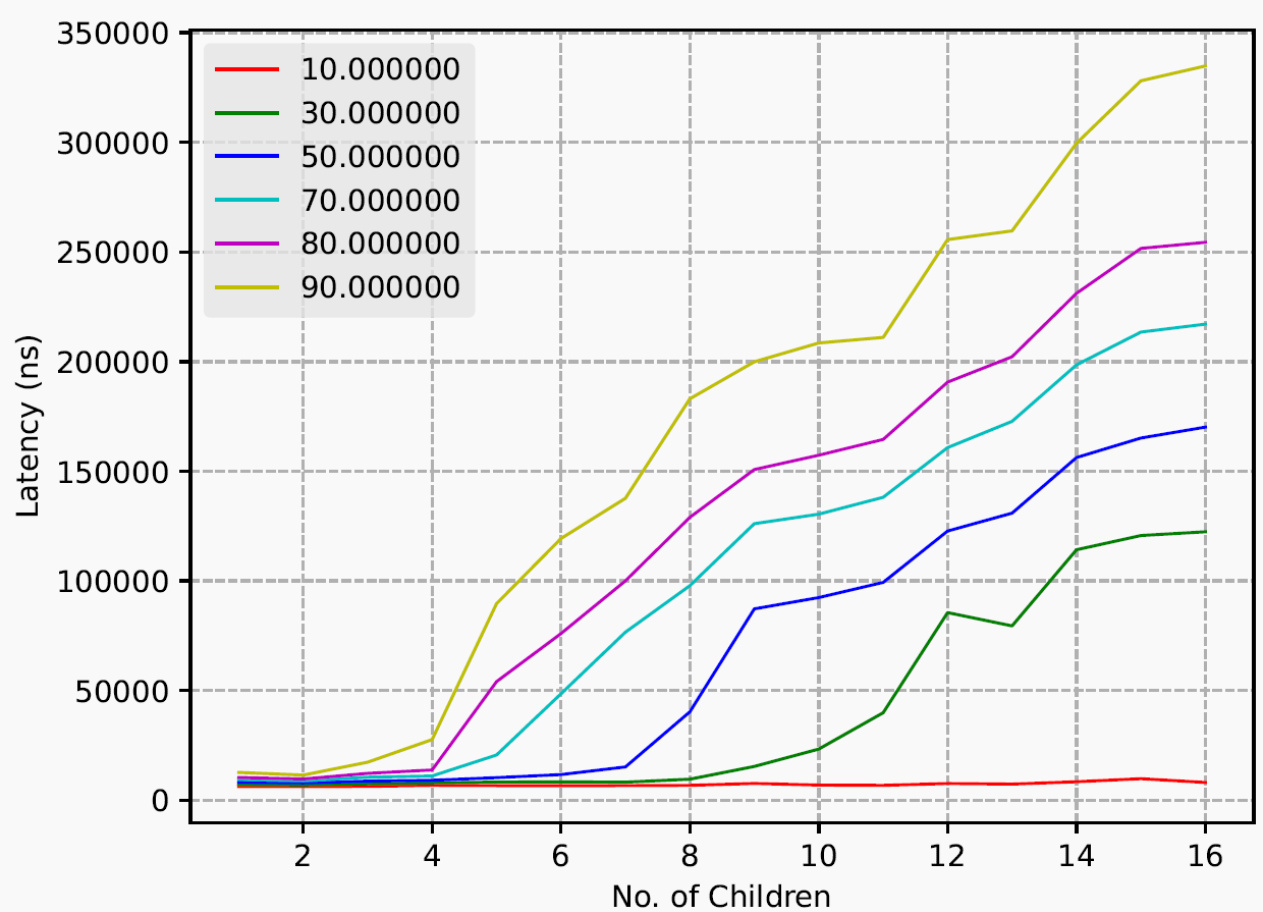

Figure 8: Average Write Latency with multiple children reading and writing

As clear from the above graph, the write latency increases with the increase in number of children. It is indeed expected, because there will be more contentions in acquiring lock. Also, for 1 and 2 children, the almost equal latency can be explained due to the presence of 2 cores on the system.

Coding Style

In this project, we have tried to insure that if any error occurs in our implementation the control switches back to the original kernel implementation, thus preventing the syscall / kernel from crashing in any unexpected scenario. Thus in a way we have tried to make our code fault tolerant.

Assumptions

We made the following assumptions during the course of the project -

- **Multilevel Parent-Child Relations** - We have assumed that multilevel parent-child relationships don't exist. Specifically, we have not accounted for scenarios where C is cloned from P and thereafter D is cloned from C.
- **Multiple parents for a file** - Our implementation doesn't consider a file with multiple parents. We have not considered the case where a parent for a particular child is overwritten. More specifically, we have assumed that if C is cloned from P_1, there are no actions equivalent to executing `cp --reflink` P_2 C.
- **No eviction of parent inode** - We have assumed that once after boot or load, the parent inode has come into memory, it will remain there until the system is rebooted again. As a trade-off for this assumption, we have not stored the inode number for each child in the extended attributes of the parent.
- **Limit on Clones** - We have assumed that the maximum number of clones of a particular file cannot go beyond `MAX_CHILDREN` (defined as 16).

Conclusions

The implementation of page-cache sharing functionality for cloned files in Btrfs presented several challenges, including the management of parent-child relationships, handling locking mechanisms, and ensuring persistence across system

reboots and crashes. Despite these challenges, our efforts have demonstrated significant improvements across various use cases, such as reading-writing scenarios and concurrent operations.

While our solution has shown promising results, there remains ample scope for further improvement. Specifically, addressing the assumption cases, such as multilevel parent-child relations and multiple parents for a file, would enhance the robustness and versatility of our implementation. Additionally, by making our solution more generic, it could potentially be integrated into the Btrfs filesystem as a standard feature, benefiting a broader range of users and use cases.

In summary, while implementing page-cache sharing functionality in Btrfs was a complex endeavor, the improvements observed, and the potential for further enhancements highlight its value as a valuable addition to the filesystem. With continued refinement and consideration of additional use cases, our solution has the potential to become a standard feature, offering enhanced performance and efficiency for Btrfs users.

Appendix: Modified Files and Locations

Below is a list of files and their corresponding functions, which were modified during the course of our project:

- ***include/linux/fs.h**: modified struct inode*
- ***mm/filemap.c**: filemap_get_pages_parent()*
- ***mm/filemap.c**: filemap_read()*
- ***fs/open.c**: do_truncate()*
- ***fs/namei.c**: vfs_unlink()*
- ***btrfs/reflink.c**: btrfs_clone()*
- ***btrfs/reflink.c**: btrfs_remap_file_range()*
- ***btrfs/inode.c**: btrfs_copy_page_cache()*
- ***btrfs/inode.c**: btrfs_unlink()*
- ***btrfs/inode.c**: btrfs_read_locked_inode()*
- ***btrfs/file.c**: btrfs_do_write_iter()*