A Lightweight MapReduce Framework Using Block-Level File Sharing over iSCSI

WooJun Lim

wjlim.eric@gmail.com

Technical Research Report

https://github.com/wjnlim/mrwsi.git

Abstract

This report presents an experimental MapReduce framework that leverages shared iSCSI (Internet Small Computer Systems Interface) block devices for intermediate file sharing. Unlike conventional MapReduce implementations that rely on shuffle daemons or network-based file transfer services, this approach enables direct shared access to intermediate data stored on iSCSI volumes. The objective is to simplify the system architecture, reduce resource overhead, and eliminate redundant kernel-level data transfers. Through iterative experimentation, the study identifies and resolves cache consistency challenges arising from multi-node access to shared iSCSI disks. The findings demonstrate that, with proper cache management, iSCSI can serve as an efficient and reliable foundation for lightweight distributed computation.

1. Introduction

The MapReduce programming model provides a simple yet powerful abstraction for large-scale data processing. However, implementations such as Hadoop and Spark rely on auxiliary services—including shuffle handlers, fetchers—to transfer intermediate data between map and reduce tasks. These mechanisms introduce substantial overhead from redundant user–kernel memory copies, persistent background services, and complex coordination protocols.

To address these inefficiencies, this research explores a block-level data sharing mechanism based on iSCSI volumes. In this model, worker nodes access intermediate data directly from shared block devices instead of transferring files through network-based protocols. This approach aims to create a minimal, efficient MapReduce system that removes unnecessary software layers and reduces data movement overhead.

However, sharing iSCSI targets introduces new challenges. The iSCSI protocol does not provide native cache coherence between clients. Each node's Linux kernel maintains its own caching layers for pages and block buffers. Consequently, concurrent access to shared volumes

can result in stale data or inconsistent metadata states. Analyzing and resolving these cache inconsistency issues is crucial for correctness of the intermediate file sharing.

The following section presents an exploration of these problems through repeated experiments and refinements, which guided the final design and implementation of the proposed framework.

2. Cache Inconsistency Analysis and Experimental Findings

iSCSI relies on each client node's own kernel caching mechanisms, and most file systems do not have built-in conflict resolution for shared access. When multiple nodes share an iSCSI volume, cached directory or data blocks may become outdated, causing inconsistent file views. However, in the context of MapReduce, where one node writes data (mapper) and others read it (reducers), iSCSI can be safely used for file sharing provided that mappers flush their data to the shared disk and reducers read the data directly from the disk bypassing their caches. This section describes the sequence of experimental approaches conducted to understand these behaviors and develop practical solutions.

2.1. Flushing Data to Shared Disk

After setting up shared volumes (which is described in the **Section 3**), a mapper created a file (`file0`) and wrote data using the `write()` system call. Since the `write()` call updates only the page cache and marks pages as dirty, additional steps were required to ensure that the data reached the mapper's shared disk. The following system calls were used:

```
fsync(fd);
fsync(blkdev_fd);
```

The first call flushes the dirty pages to the disk and the second invokes `blkdev_issue_flush()`, flushing the block device's write cache and pending block I/O operations.

2.2. First Successful Open

After the mapper flushed the data, reducers were able to open and read the file successfully. Since their caches were initially empty, all data were fetched directly from the shared disk.

2.3. Open Failure After New File Creation

When the mapper later created another file (`file1`), the reducers failed to open it because their cached directory blocks still reflected the earlier state, which contained only `file0`. To address this, we attempted to explicitly clear the caches on the reducer nodes using the following commands:

```
echo 3 > /proc/sys/vm/drop_caches
ioctl(blkdev_fd, BLKFLSBUF, 0);
```

The first command clears the inode, dentry, and page caches, while the second invokes `invalidate_bh_lru()` to flush the buffer-head LRU and buffer cache.

Despite these cache-drop steps, the subsequent `open()` call for the `file1` still failed. A closer examination revealed that during the initial failed attempt, the kernel's internal `lookup_dcache()` function—invoked by `open()`—was unable to locate the file's dentry in the dentry cache and failed to search for the file name within the stale directory block. As a result, it created and cached a negative dentry of the `file1`.

Although the command '`echo 3 > /proc/sys/vm/drop_caches`' is expected to clear the dentry cache, dentries marked with the `DCACHE_REFERENCED` flag-set automatically when allocated-are not immediately evicted. Instead, the command only clears the flag and leaves the object in the cache. Consequently, the negative dentry persisted after the cache-drop, causing the repeated `open()` failures for the `file1`.

2.4. Successful Open Before Caching

After identifying this behavior, we adopted a new strategy of dropping caches before opening newly created files and this method resolved the issue. By dropping the buffer cache—including cached directory blocks—using the command `ioctl(blkdev_fd, BLKFLSBUF, 0)` on the reducer nodes, the reducers were able to fetch the updated directory blocks containing new files created by the mapper from the shared disk and successfully opened them.

2.5. Pre-Allocated Files approach

While pre-dropping caches ensured correctness, performing cache invalidation before every file access introduced significant performance overhead. To address this, the design shifted toward pre-allocated files strategy: fixed-size files were created before the shared disks were mounted by remote nodes. By reusing these pre-created files instead of dynamically creating new ones, the directory blocks always contained valid dentries, eliminating the need for repeated cache drops. Mappers overwrote existing pre-allocated files and flushed data using both `fsync(fd)` and `fsync(blkdev_fd)`, while reducers opened the files with the `O_DIRECT` flag to bypass the page cache and read the data from the disk.

2.5.1 Limitations of `fallocate()`

An attempt to create pre-allocated files using `fallocate()` was failed. Although `fallocate()` reserves space on disk, it marks the allocated blocks as uninitialized, causing subsequent reads to return zeros. To avoid this, files were pre-initialized through explicit writes(e.g., `dd` command), ensuring proper block allocation and initialization.

2.6. Summary of Experimental Insights

The iterative experiments yielded several key insights:

1. Cache inconsistencies primarily originate from unsynchronized directory metadata, not only data pages.
2. Post-failure cache drops are ineffective because negative dentries persist in the cache.
3. Dropping caches before file open ensures correct visibility of newly created files on shared disks.
4. Pre-allocated files provide a stable solution by eliminating the need for repeated cache invalidations.

These insights served as the foundation for the final design of the framework, described in the following section.

3. Design and Implementation

The final design incorporates these findings to enable safe and efficient data sharing among worker nodes via shared iSCSI volumes. Each node exports its local disk as an iSCSI target and mounts the targets provided by other nodes, allowing their remote disks to be accessed as if they were local.

3.1. Architecture Overview

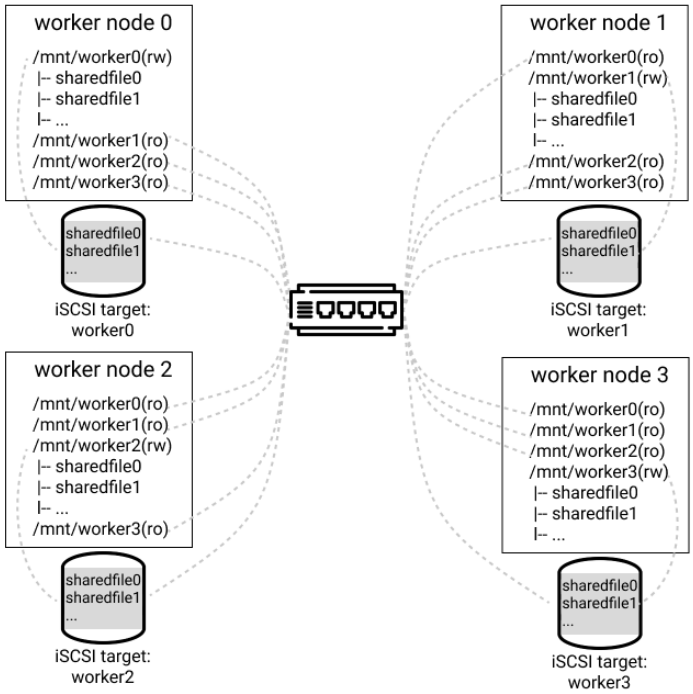


Figure 1: Overall architecture of the shared iSCSI-based MapReduce worker nodes.

Each node consists of:

- A **local iSCSI target** mounted read-write mode for intermediate outputs.
- **Pre-allocated files** within the local target disk.
- **Remote iSCSI targets** from peer nodes mounted in read-only mode.

3.2. MapReduce execution with shared iSCSI overview

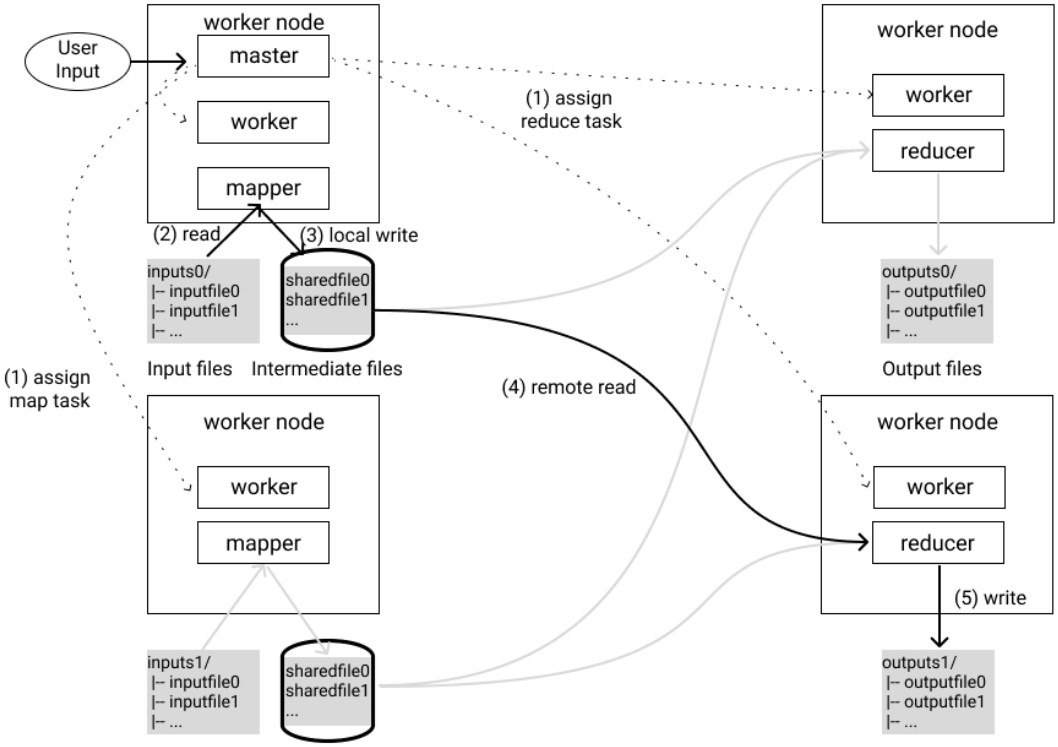
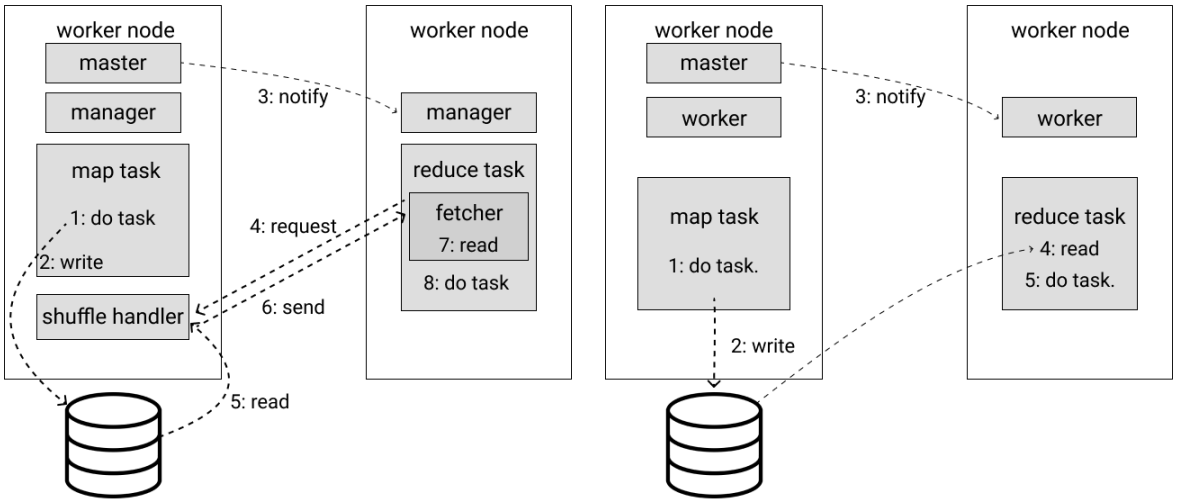


Figure 2: Execution overview of the shared iSCSI-based MapReduce.

1. The master node assigns map and reduce tasks to workers.
2. Each mapper reads and processes its input split.
3. Mappers write their intermediate data to pre-allocated files assigned by the local worker process and flush the data using both `fsync(fd)` and `fsync(blkdev_fd)`.
4. When reducers are notified by the master about these files, they open the files using the `O_DIRECT` flag to bypass caches and read the data from remote shared disks.
5. Reducers then process the data and write final outputs to their local disks.



(a) conventional data transfer. (b) intermediate data sharing.

Figure 3: Comparison of Conventional Data Transfer and Block-Level File Sharing

4. Results and Discussion

Compared to conventional network-based shuffling—where intermediate data are transferred through auxiliary daemons such as shuffle handlers and fetchers—the block-level file sharing approach significantly reduces transfer overhead and eliminates the need for these additional services. The most notable improvements arise from the entire data movement path: by removing the shuffle network stage, which involves multiple user–kernel I/O operations, both context-switching overhead and memory copy costs are substantially reduced.

The results also show that with proper cache management—using pre-allocation, explicit flushing, and page-cache bypassing—consistent shared access can be achieved without relying on a distributed file system in the common MapReduce pattern where a single mapper writes data and multiple reducers read it. Although large-scale scalability remains limited by iSCSI bandwidth, this approach demonstrates the feasibility of block-level data sharing for distributed computation.

5. Experimental Setup

Experiments were conducted on a 4-node cluster connected over a 1 Gbps LAN. Each node ran CentOS 7 with Linux kernel 3.10.0. The iSCSI target was configured using `targetcli`, and initiators used `open-iscsi`. The experiments were performed in two different environments.

Table 1: Experiment Environments

Environment	**CPU**	**RAM**	**Storage**	**OS**	**Kernel**
Local VM (×4)	Intel i7-10700K	16 GB	64 GB SSD	CentOS 7	3.10.0
AWS t3.medium (×4)	Intel E5-2686 v5	4 GB	16 GB EBS HDD	CentOS 7	3.10.0

6. Conclusion

This project proposed a lightweight MapReduce framework based on shared iSCSI volumes. By leveraging direct block-level access, the system eliminates conventional shuffle mechanisms, simplifying architecture and reducing data movement overhead. The trial-and-error experiments detailed in **Section 2** identified the kernel caching behavior and guided the development of efficient MapReduce framework using pre-allocated files and explicit cache control. Future work will explore large-scale evaluations on clusters with eight or more nodes under higher network bandwidth and investigate kernel-level cache invalidation mechanisms to further enhance consistency and performance.

7. Souce code of the MapReduce with shared iSCSI

The full implementation source codes and scripts are available at:
`https://github.com/wjnlim/mrwsi.git`

References

[1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Sixth Symposium on Operating System Design and Implementation*, pages 137-150, San Francisco, CA, 2004.

[2] Linux kernel source code (Version 3.10.0) [Source code].

[3] D. Bovet and Marco Cesati. Understanding the Linux kernel (3rd Edition). O'Reilly, 2005.

[4] Apache Software Foundation. (2018). Apache Hadoop 2.9.2 documentation. `https://hadoop.apache.org/docs/r2.9.2/`.

[5] Tom White. Hadoop: The Definitive Guide (4h Edition). O'Reilly, 2015.

[6] Wickham, H. The Split-Apply-Combine Strategy for Data Analysis. In *Journal of Statistical Software 40 (1)*, pages 1-29, 2011.