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

WooJun Lim

wjlim.eric@gmail.com

Technical Research Report

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

Abstract

This report presents an experimental MapReduce framework that leverages shared iSCSI (Internet Small Computer Systems Interface) block devices for direct access to intermediate data without a distributed file system. Unlike conventional MapReduce implementations, where reducers retrieve intermediate data through network-based transfer mechanisms, the proposed design allows reducers to read data directly from shared iSCSI volumes, eliminating the need for network data transfer. The objective is to simplify the system architecture, reduce resource overhead, and avoid redundant user–kernel space data copies. Through iterative experimentation, the study identifies and resolves cache consistency challenges arising from multi-node access to shared iSCSI storage. The results demonstrate that, with appropriate cache management, iSCSI can function as an effective shared storage mechanism for lightweight and efficient distributed computing architectures.

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 data through network-based protocols. This approach aims to create a minimal and efficient MapReduce system that removes unnecessary software layers and reduces data movement overhead.

However, sharing iSCSI targets without a distributed file system introduces new challenges. Because iSCSI provides no native cache coherence, direct shared access to iSCSI volumes without cache control can lead to stale data or inconsistent metadata states. Understanding and resolving these cache-consistency issues is essential for ensuring correctness in the proposed design.

The following section examines these issues through repeated experiments and refinements, which guided the final design and implementation of the framework.

2. Cache Inconsistency Analysis and Experimental Findings

iSCSI relies on each client node's own kernel-level caching mechanisms, and most file systems lack built-in conflict resolution for shared access. When multiple nodes access the same iSCSI volume without a distributed file system, cached metadata may become outdated, causing inconsistent file views across nodes. However, in the context of MapReduce—where one node writes data (mapper) and others read it (reducers)—iSCSI volumes can be safely shared for direct access if mappers explicitly flush data to the shared disk and reducers read it directly while bypassing their caches. This section presents a series of experimental investigations conducted to analyze these behaviors and develop practical solutions for correct shared access.

2.1. Flushing Data to Shared Disk

After setting up shared iSCSI 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`), wrote data to it, and flushed the data using `fsync(fd)` and `fsync(blkdev_fd)`, the reducers failed to open the file. This occurred 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 objects in the cache. This explains why 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 dropped caches before opening newly created files to prevent negative dentry creation, and this approach 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 from the shared disk and access the newly created files from the mapper successfully.

2.5. Pre-Allocated Files approach

While the cache-dropping strategy described above ensured correctness, performing cache invalidation before every file access introduced significant performance overhead. To address this, the design shifted toward a pre-allocated file strategy. In this approach, fixed-size files are created before the shared disks are mounted by remote nodes. Mappers overwrite existing pre-allocated files and flush data using both `fsync(fd)` and `fsync(blkdev_fd)`. Because the files already exist, the cached directory blocks on reducer nodes contain the necessary dentries. Reducers then open the files with the `O_DIRECT` flag and read the data directly from the shared disks bypassing their page caches. After adopting this approach, the reducers were able to successfully access the data written by the mapper without the cache-drop steps.

2.5.1 Limitations of `fallocate()`

An initial attempt to create the pre-allocated files used `fallocate()`, but it did not behave as expected. Although `fallocate()` reserves space on disk, it marks the allocated blocks as uninitialized, causing subsequent `read()` calls to return zeros. To avoid this, the files were pre-initialized through explicit writes(e.g., using `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 arise from unsynchronized directory metadata across worker nodes, rather than from data pages alone.
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 an efficient 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 direct access to shared iSCSI volumes from worker nodes. 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

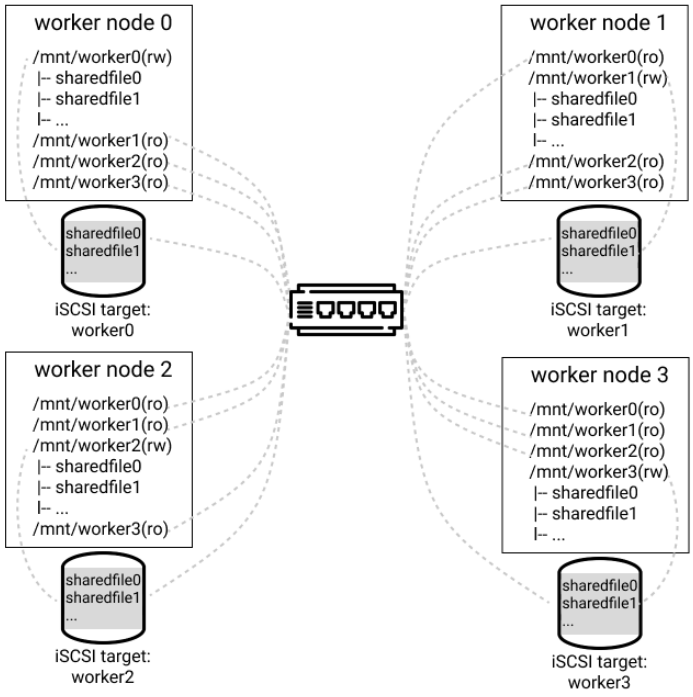


Figure 1: Overall architecture of the shared iSCSI-based MapReduce worker nodes.

Each node consists of:

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

3.2. Overview of MapReduce Execution with Shared iSCSI Storage

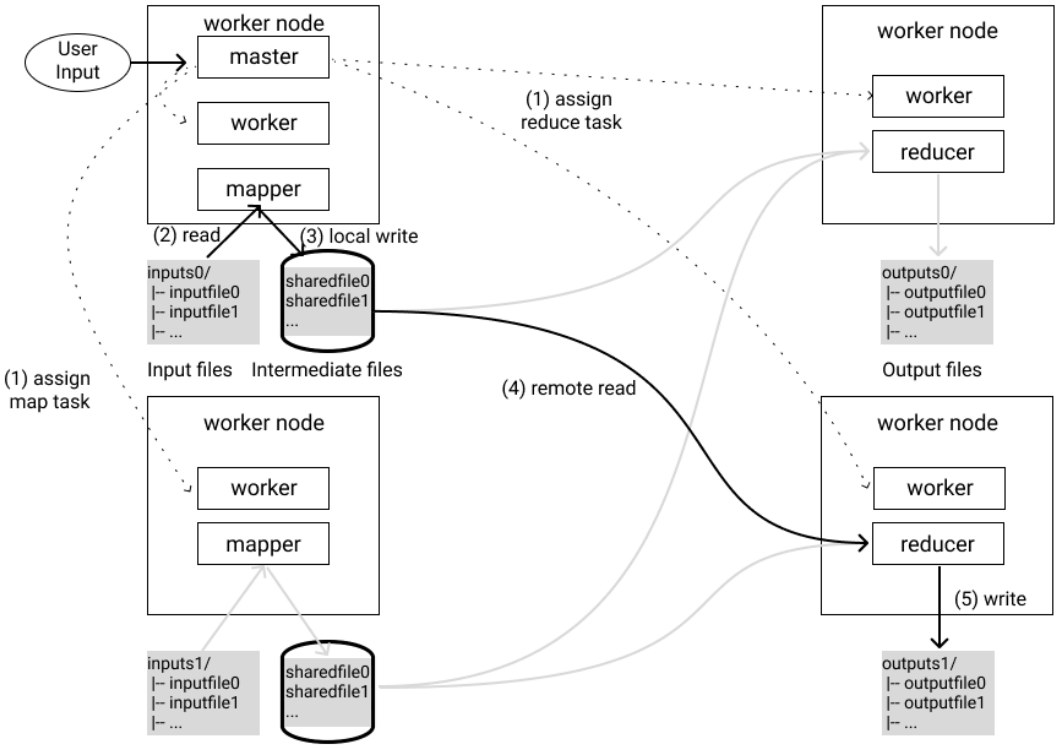
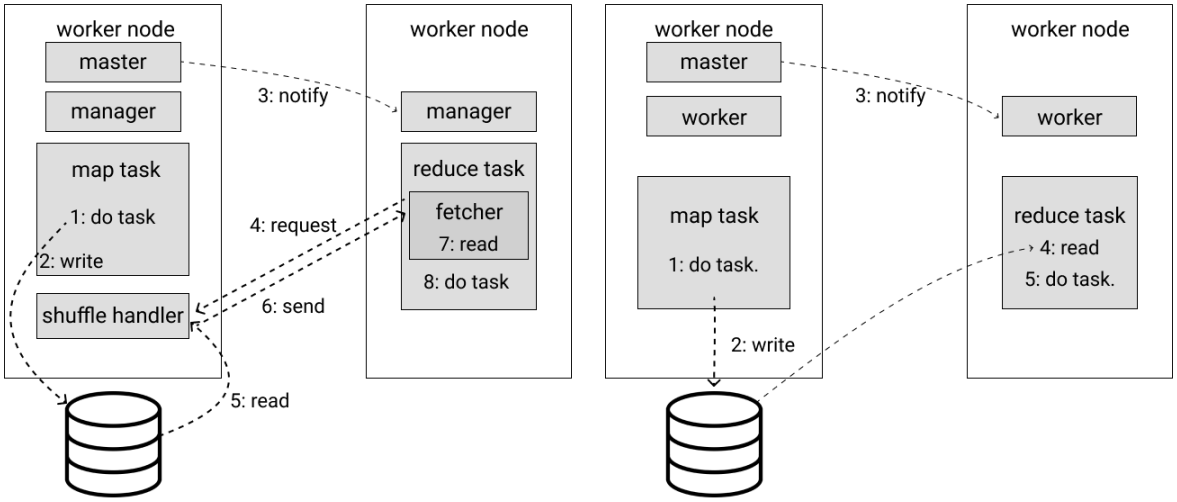


Figure 2: Execution overview.

1. The master process assigns map and reduce tasks to workers.
2. Each mapper reads its input split and executes the user-defined map function.
3. Mappers write intermediate data to their assigned pre-allocated files on the shared iSCSI volumes and flush the data using both `fsync(fd)` and `fsync(blkdev_fd)`.
4. When each reducer is notified by the master about intermediate files, it opens the corresponding files with the `O_DIRECT` flag and reads the data directly from remote shared volumes.
5. After reading all intermediate data, reducers execute the user-defined reduce function and write the final output partitions to their local disks.

4. Results and Discussion



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

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

Compared to conventional MapReduce's shuffle mechanism—where intermediate data is transferred through auxiliary services such as shuffle handlers and fetchers—the block-level sharing approach replaces the network-based transfer phase with direct access to shared iSCSI storage, simplifying the overall architecture. By eliminating network data transfer components, the framework also reduces user–kernel I/O operations, context-switching overhead, and memory copy costs typically incurred during shuffle phase. The results further show that with appropriate cache handling—using pre-allocated files, explicit flushing, and page-cache bypassing—direct shared access over iSCSI is achievable without a distributed file system in the common MapReduce pattern where a single mapper writes data and multiple reducers read it. Although scalability remains constrained by iSCSI bandwidth, the findings demonstrate the feasibility of block-level data sharing as a lightweight alternative for distributed computation.

5. Experimental Setup

Experiments were conducted on 4-node and 9-node clusters connected over a 1 Gbps LAN. The iSCSI targets were configured using `targetcli`, and the initiators used `open-iscsi`. The experiments were performed in three 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
PC (×9)	Intel i7-6500U	8 GB	1 TB HDD	CentOS 7	3.10.0

6. Conclusion

This work demonstrates that shared iSCSI volumes can be leveraged for a simplified MapReduce execution model by replacing network data transfers with direct block-level access to intermediate data. Through experiments analyzing kernel caching behavior, the study identified the conditions required for correct access to shared iSCSI storage and developed an effective approach—using pre-allocated files, explicit flushing, and page-cache bypassing—to ensure correctness without a distributed file system. Future work will evaluate scalability under larger cluster sizes with higher network bandwidth and investigate kernel-level cache invalidation mechanisms to further improve 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/blfs_mr.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.