Avoiding Data Loss and Corruption for File Transfers with Fast Integrity Verification

Ahmed Alhussen*, Engin Arslan

University of Nevada, Reno, 1664 N Virginia St, Reno, NV 89557

Abstract

End-to-end integrity verification is used to avoid silent data corruption in file transfers by comparing the checksum of files at source and destination end points. However, it increases transfer times significantly as checksum computation requires reading files back from the storage and running compute-intensive hash computation. In this paper, we propose Fast Integrity VERification (FIVER) algorithm which alleviates the overhead of end-to-end integrity verification by overlapping checksum computation with transfer operation and enabling I/O sharing between the two. The results obtained from various network and dataset settings show that FIVER is able to bring down the cost of end-to-end integrity verification from up to 120% by the state-of-the-art solutions to below 15%. Moreover, existing implementations of end-to-end integrity verification are vulnerable to permanent data loss in the case of an unexpected power outage due to completing the integrity verification process while data is still on memory. FIVER addresses this issue by enforcing dirty data on memory to be flushed to disk before finishing integrity verification such that power loss during any phase of a transfer would cause integrity verification to fail and transfer application to retransfer lost data.

Keywords: File transfers, integrity verification, data corruption, data loss.

1. Introduction

With the advancement of computing and sensing technology, the amount of data generated by scientific applications is growing at an unprecedented rate. For example, the high energy physics and particle experiment ATLAS produces 1 PB of data each second, which is reduced to 1-2 GB after filtering [1]. Similarly, cosmology project Large Sky Survey Telescope will be operational in 2022 and take high-quality pictures of the universe using a 3200 megapixel camera for 10 years and is expected to produce 15 TB of raw data every night [2]. This massive amount of data often needs to be moved to geographically dispersed locations for various purposes such as processing, collaboration, and archival. Ensuring the integrity of data while moving it is critical for many applications (e.g., Dark Energy Survey [3] and Sky Survey Simulation [4]) whose computation is extremely sensitive to data manipulation. However, built-in integrity verification methods for file transfers (e.g., TCP checksum) lack robustness to capture data corruption that may occur while transferring data from source to destination.

To give an example, TCP is the de-facto standard for file transfers due to its efficiency and reliability. However, previous studies found that TCP's 16-bit checksum fails to detect errors once in 16 million to 10 billion packets [5].

*Corresponding author
Email address: `aalhussen@nevada.unr.edu` (Ahmed Alhussen)

While this results in a missed packet corruption once in every 30 minutes to 300 hours in 1G networks, the mean time between missed errors reduces to 18 seconds to 3 hours in a 100 Gbps networks when MTU size is set to 1.5 KB. As a result, 5% of all file transfers in a 100 Gbps network developed data corruption that TCP checksum failed to detect and recover [6]. Therefore, end-to-end integrity verification is introduced to improve the robustness of transfer applications against data corruption.

The naive implementation of end-to-end integrity verification for file transfers works as follows: Sender first reads a file from disk and sends it to the receiver. Once the transfer of the file is completed, the sender reads it again to compute its checksum using a secure hash algorithm such as MD5 and SHA256. The receiver does the same and computes the checksum of the file after it is received and written to the storage. Finally, the receiver sends the checksum it computed to the sender to compare. If the checksum values of the sender and the receiver match, then the transfer is marked as successful, otherwise the receiver copy of the file is deemed corrupted and transfer is restarted. If the dataset consists of multiple files, then the transfer of the next file begins only after the previous file's transfer and integrity verification is completed successfully. While end-to-end integrity verification is crucial for many applications, it imposes significant overhead due to involving I/O and CPU-intensive checksum computation. To alleviate its overhead, *File-Level Pipelining* is proposed to overlap the checksum computation of a file with the transfer of another file [7]. Despite improving the performance over the naive approach, *File-Level Pipelining* has two main drawbacks. First, pipelining transfer and checksum operations for different files make it difficult to take full advantage of pipelining since their execution time differs especially when the dataset is a mix of small and large files. Second, running checksum computation and transfer operations for different files at the same time is likely to cause I/O contention, due to which both processes would experience a slowdown. Liu et al. [8] proposed *Block-Level Pipelining* to address these limitations by dividing large files into smaller blocks to better overlap transfers and checksum operations for datasets with mixed file sizes. However, transferring large files in small blocks increases transfer times since small files yield lower transfer throughput than larger ones [9]. Further, if the transfer speed is faster than checksum computation speed, enforcing synchronization for these operations would cause transfer thread to stay idle while checksum thread is trying to process a block. Keeping the transfer channel idle for longer than connection's round trip time, in turn, would lead to TCP window size to reset to the initial value for every block transfer [10], deteriorating transfer performance.

To overcome performance limitations of integrity verification for file transfers, we propose Fast Integrity VERification (FIVER) that executes transfer and checksum operations simultaneously for the same file to minimize the cost of integrity verification. While previous studies focus on overlapping checksum computation and transfer operation of different files or blocks, FIVER overlaps them for the same file. For example, if checksum computation of a file takes 30 seconds and transfer time takes 10 seconds, then FIVER finishes transfer and integrity verification in around 30 seconds with the help of simultaneous execution of both processes. In addition, rather than reading files from the storage twice (one for transfer and the other for checksum computation), FIVER reads each file once and shares I/O between the checksum and transfer operations since file read requests of checksum computation is being served from cache memory anyway for files that are smaller than the size of memory.

In addition to incurring high overhead, existing implementations of end-to-end integrity verification are also vulnerable to data loss in the event of a power outage as they complete the integrity verification before files are fully written to disk. When the receiver of a file transfer receives data from the network and attempts to write it to disk, write requests may not be fulfilled immediately as operating systems try to optimize disk I/O performance by aggregating write calls and flushing them all at once when enough requests were issued. This, however, leads to data loss when the power supply of volatile memory (i.e., random access memory) is lost even if the transfer application had already confirmed the integrity of the transfer. While this could be a reasonable trade-off for applications that keep the copy of data on nonvolatile storage to recover from failures, it would lead to partial or complete data loss for file transfers as transfer application would be unaware of power loss if it takes places after the integrity verification. Therefore, FIVER enforces data on volatile storage to be flushed to disk before verifying the integrity of transfers such that power loss can be detected and recovered. In summary, the contributions of this paper are as follows:

- We introduce FIVER that overlaps transfer and checksum compute operations for a file and enables I/O sharing between them to minimize the overhead of integrity verification.
- We enhance FIVER with data loss protection by forcing cache eviction before completing the integrity verification.
- We introduce dynamic parallelism to identify and mitigate performance bottlenecks in integrity verification-enabled file transfers.
- We run extensive analysis to evaluate the performance of FIVER in various network and datasets settings.

The rest of the paper is organized as follows: Section 2 describes related work and Section 3 demonstrates the issues of existing implementations of end-to-end integrity verification algorithms. Section 4 describes FIVER and presents pseudo code for transfer sender and receiver. Section 5 presents experimental results and Section 6 concludes the paper with a summary and potential future directions.

2. Related Work

As an increasing number of applications rely on the accuracy of data to produce accurate results, integrity verification has been widely studied in many areas including storage outsourcing [11, 12, 13], long term achieves [14, 15], file systems [16, 17, 18], databases [19], provenance [20] and data transfer [8].

Zhang et al. [18] evaluated Zettabyte Files System (ZFS) in terms of robustness to disk and memory fault injections. It has been found that while ZFS is able to detect and mostly recover from disk corruptions, it is susceptible to memory corruptions since it does not check the integrity of data blocks when they reside in the memory. FIVER also makes a similar assumption and rely on existing control mechanisms to recover from possible data corruption in memory. On the other hand, Error Correcting Codes (ECC) [21] has been proposed to detect and recover from the most single and multi-bit failures as well as memory leaks [22]. Meza et al. [23] have shown that a very large portion of memory errors (over 98%) is correctable by the common ECC implementation. Yet, more advanced error correction algorithms such as Chip-kill [24] are proposed to recover from more sophisticated (up to 4-adjacent bit corruption) errors.

Xiong et al. [25] proposed *fsum* that uses a bloom filter to compute the checksum of very large datasets stored in long term archival storage. Instead of calculating the checksum for each file, they partition files into blocks such that multiple threads can process different portions of the file simultaneously. To achieve ordering among threads, a bloom filter is used to combine the results from threads. Once all the blocks are processed and corresponding hash values are inserted into the bloom filter, the final checksum is calculated by computing the hash of the bloom filter. *fsum* runs up to 4x faster than the traditional file-level checksum computation approach but comes with the cost of false positives due to relying on bloom filter.

Globus [7] supports end-to-end integrity verification for data transfers. It can pipeline data transfers and checksum computation to minimize the overhead of integrity verification. However, it calculates the checksum of files after their transfer completes. That is, files are read twice in the source server, one to send to destination and one to compute the checksum. Yet, its pipelining approach fails to work well when a dataset consists of mixed file sizes. Liu et al. [8] proposed block-level pipelining to achieve better overlapping of checksum computation and file transfer operations. It reduces execution time considerably especially when the dataset is composed of files with mixed sizes. Similar to Globus, block-level pipelining also reads files twice at the source server. However, as opposed to Globus, its second read is faster for all file sizes since blocks are small enough to be kept in memory for some time, so once a block is read for transfer operation, it will be cached. When the checksum computation process attempts to read the file block, it will find it in the memory. On the contrary, FIVER reads files once and runs the transfer and checksum computation processes simultaneously, reducing I/O overhead and time required to compute the checksum [26].

In the previous work, we presented RIVA which aims to detect undetected write errors that might happen while flushing file data from memory to disk [27, 28]. RIVA does this by enforcing cache eviction immediately after the transfer such that checksum computation has to read files directly from disk. While RIVA offers stronger integrity verification coverage, it can lead to more than two times longer execution time in networks where I/O throughput is slower than transfer speed. Therefore, FIVER can be used to execute integrity verification while keeping its overhead at a minimum while preventing permanent data loss when end servers experience an unexpected power outage.

On the other hand, previous work in high-speed networks mostly focuses on transfer scheduling [7, 29], throughput optimization [30, 9, 31, 32], and power consumption optimization [33]. Globus [7] offers data transfer and sharing services and it is well adopted by the research community. HARP [34] models data transfers using historical data and real-time sampling and uses this model to estimate the set of values for application-layer transfer parameters that would maximize the throughput of given transfer tasks. PCP [35] finds the optimal values for transfer parameters by running a series of sample transfers in the runtime. Alan et al. [33] proposed scheduling algorithms that can tune application-layer transfer parameters to find a balance between transfer throughout and energy consumption at the end hosts. The algorithms monitor the CPU usage of end hosts and estimates energy consumption with the help of models that relate CPU usage to energy consumption. Then, a cost function is used to determine the energy efficiency of each configuration based on transfer throughput and energy consumption values. Finally, a configuration with minimum cost function is identified and used in the rest of the transfer.

Testbed	Storage	CPU	Memory Size	Bandwidth	RTT	Disk Write Speed
HPCLab-WS	SATA SSD	8 x Intel Core i5-7600 @3.50GHz	16 GB	1G	0.2 ms	< 500MB/s
HPCLab-DTN	NVMe SSD	16 x Intel Xeon E5-2623 @2.60GHz	64 GB	40G	0.2 ms	< 3GB/s
Chameleon-WAN	SATA SSD	12 x Intel Xeon E5-2650 @2.30GHz	64 GB	1G	32 ms	< 500MB/s
Chameleon-LAN	SATA HDD	12 x Intel Xeon E5-2670 @2.30GHz	128 GB	10G	0.2 ms	< 100MB/s
Pronghorn	GPFS	16 x Intel Xeon E5-2683 @2.10GHz	192 GB	10G	0.1 ms	< 3GB/s

Table 1: Specifications of test environments.

3. End-to-End Integrity Verification for File Transfers

The simple implementation of integrity verification works in three steps. In the first step, a file is transferred from source to destination using a preferred transfer protocol, such as TCP. Once the transfer of the file is completed and it is written to the storage at the destination, the checksum of the original file at source and transferred copy at destination are computed using a desired hash function such as MD5 or SHA256 as part of the second step. In the third and final step, the destination server sends the checksum value it computed to the source server to compare. If the checksum values match, then the transfer is marked as successful. Otherwise, the copy of the file at the destination is deemed corrupt and file transfer is restarted.

The objective of end-to-end integrity verification is to detect data corruption by comparing the checksum of the file at the source and destination servers. On the other hand, operating systems are designed to minimize cache misses, so if a file is recently read or written, it will be kept in the memory to optimize future accesses. For example, when a 1 MB file is transferred from a remote server and written to disk, the operating system will keep the file data in the main memory (even after flushing it to disk) to be able to respond following read requests faster. This, however, causes integrity verification process to be restricted to the cached copy when it is executed immediately after the transfer of a file.

Figure 1 shows the cache hit ratio for source and destination servers when *File-Level Pipelining* is used for a transfer of an 8GB file with integrity verification in the HPCLab-DTN network whose specification is given in Table 1. Hit ratio defines the percentage of I/O requests that are found in cache memory. High cache hit ratio for a file read indicates that the file pages were mostly served from page cache, so no disk I/O is performed. As shown in the figure, the cache hit ratio is always 100% during checksum computation both for sender and receiver servers. Sender-side hit ratio is small during the file transfer since this is the first time the file is being read. Once the file transfer is completed, check computation thread starts to read the file to compute its checksum as part of integrity verification. Since file transfer does not involve disk read at the receiver server, the cache hit ratio is reported as 100% throughout the transfer. Upon the completion of the transfer, the checksum

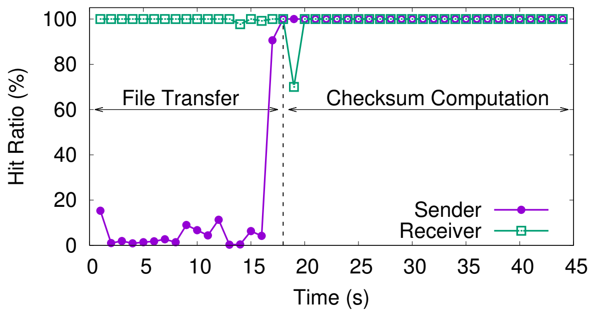


Figure 1: File read requests of checksum computation processes are served from page cache (i.e., page hit) when it is executed after the transfer.

computation thread starts reading the file to compute its checksum, which ends up being served from memory (cache hit ratio is 100%). This is because the operating system keeps file pages in the memory after they are accessed during the transfer. As a result, if free memory space of the sender and receiver servers is larger than the size of a file, I/O requests of checksum computation will be served from cache memory. Since the average file size of scientific data transfers is in the order of megabytes and most production data transfer nodes are equipped with 64 GB or larger RAMs [36], this will result in checksum computation of most file transfers to be operated on cached data.

Observation 1: *Existing integrity verification algorithms read files from cache memory during checksum computation.*

Furthermore, we investigated the impact of caching (aka write-back caching) on file transfers in the event of sudden power loss. Figure 2 shows the size of dirty pages for the receiver of a 1GB file transfer with integrity verification (using File-level Pipelining) in the HPCLab-WS network. Dirty page refers to file content whose copy on memory is modified recently but has not been flushed to disk yet. For file transfers, it refers to data that the receiver streamed from the network and issued write system calls, but the operating system has not written to disk yet.

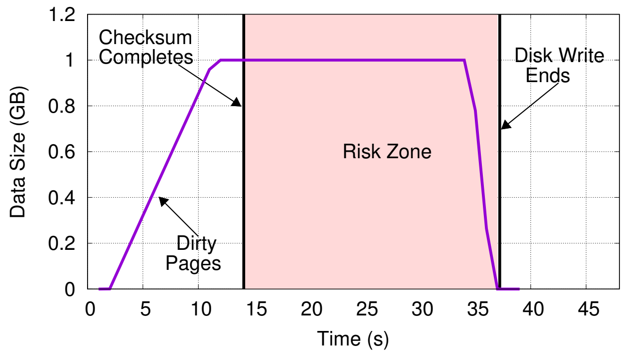


Figure 2: Write-back caching can cause permanent data loss for file transfers despite using integrity verification.

The file transfer completes at around 11s which is followed by checksum computation to verify the integrity of the transfer. The integrity of the transfer is verified at around 14.5s, however, write-back caching postpones disk write for the buffered data until 32s which causes actual disk write operation to complete at 37s. Consequently, the interval between the completion of integrity verification and disk write poses permanent data loss risk for data stored in volatile storage in the event of power loss. We believe that integrity verification for file transfers should be hardened to prevent such data losses even if underlying systems are prone to data loss. This is especially important for third-party transfer services such as Globus [7] where users receive a confirmation upon the successful completion of transfers since any power loss after the confirmation would mislead users to accept incomplete files as complete.

Observation 2: *Completing integrity verification without ensuring disk write poses data loss risk in the case of unexpected power loss.*

4. Fast Integrity VERification (FIVER) Algorithm

Taking on Observations 1 and 2, we present FIVER that aims to minimize the overhead of end-to-end integrity verification algorithm while offering robustness against data corruption and loss. To do so, FIVER exploits cache access behavior of existing integrity verification methods and implements simultaneous execution of transfer and computation operations for the same file. This approach has two main advantages over the state-of-the-art solutions: First, it allows better pipelining of transfer and checksum operations by running them for the same file, thereby shortening total execution time. As an example, the transfer takes 17 seconds and checksum computation takes

27 seconds in Figure 1. Sequential execution of them then leads to total execution time to be 44 seconds. On the other hand, it would only take 27 seconds if they could be executed in parallel, shortening execution time by 38%. Second, it speeds up checksum computation through I/O sharing with transfer operation. Transfer threads on the sender and receiver keep file data on memory after processing it such that checksum threads can use it to calculate the checksum. This data sharing minimizes I/O overhead when disk speed is the bottleneck of the whole process.

Algorithm 1: File transfer methods for FIVER sender and receiver

```
1  Function send(fileName, offset, length):
2      global queue, socket // Fixed size, synchronized queue
3      file = openForRead(fileName, offset)
4      size = 0
5      computeChecksum(fileName, offset, length)
6      while file.read(buffer) > 0 and size < length do
7          socket.write(buffer)
8          queue.add(buffer) // Synchronized operation
9          size = size + buffer.length
10     end
11     return

12 Function receive(fileName, offset, length):
13     global queue, socket // Fixed size, synchronized queue
14     file = openForWrite(fileName, offset)
15     size = 0
16     computeChecksum(fileName, offset, length)
17     while socket.read(buffer) > 0 and size < length do
18         file.write(buffer)
19         queue.add(buffer) // Synchronized operation
20         size = size + buffer.length
21     end
22     return
```

Algorithm 2: FIVER checksum computation method for integrity verification

```
1  Function computeChecksum(fileName, offset, length):
2      remaining = length
3      while remaining > 0 do
4          chunkSize = min(remaining, CHUNK_SIZE)
5          remaining = remaining - chunkSize
6          tmp = chunkSize
7          while tmp > 0 do
8              buffer = queue.remove()
9              checksum.update(buffer)
10             tmp = tmp − buffer.length()
11         end
12         localChecksum = checksum.digest()
13         if agent is Sender then
14             remoteChecksum = socket.read()
15             if localChecksum == remoteChecksum then
16                 Transfer is successful
17             else
18                 curOffset = length - remaining -chunkSize
19                 send(fileName,offset + curOffset, chunkSize)
20             end
21         else
22             cache.clear(fileName, offset, chunkSize)
23             socket.send(localChecksum)
24         end
25     end
26     return
```

Algorithm 1 and 2 illustrate how FIVER operates. Both sender and receiver execute transfer and checksum computation methods concurrently on different threads. *send* and *receive* methods are used to transfer files by the sender and the receiver, respectively. As mentioned in Observation 1 that when checksum threads attempt to read small files after their transfer, file pages are served from the page cache. Therefore, FIVER reads files once and shares data between threads via a synchronized *queue* to move file caching to user space and remove the need for system calls to read files (lines 8 and 19 in Algorithm 1). The total execution time of FIVER depends on the slowest of transfer and checksum computation operations as it executes them simultaneously and waits for both to finish. Since the shared *queue* object has a fixed size, it will ensure that the transfer thread waits for the checksum thread if it is faster. As queue size is important to determine how much space FIVER can reserve and let the transfer speed go ahead of the checksum thread, we assessed its impact in Section 5.4. To guarantee that files are flushed to disk at the receiver side before completing the integrity verification, FIVER evicts the file pages of the current chunk before sending its checksum to the sender (line 22 in Algorithm 2). Upon the completion of the cache eviction, the receiver sends the checksum to the sender to compare. If the checksum values do not match, then the destination copy of the file is assumed to be corrupted and the file is transferred again.

When an integrity verification algorithm detects a mismatched checksum value, it needs to transfer the file again to ensure the correctness of data at the destination. For algorithms that run integrity verification at the file level (i.e., calculating a checksum for the whole file), this means repeating the transfer and integrity verification operations for the whole file again even if only a single bit is corrupted, which incurs a significant cost for large files. Thus, we implemented chunk-level integrity verification for FIVER as follows: FIVER first calculates the size of a chunk as the minimum of file size (or remaining data size) and maximum chunk size ($CHUNK\_SIZE$) as shown in line 4. Then, it processes items in the queue until the total size reaches the size of the chunk. Then, it sends the computed checksum value for the chunk to the sender to compare. Since checksum computation is executed when *update*() function is called, calling *digest*() method frequently has a negligible computational cost. Since the size of checksum value is in the order of bytes (16 bytes for MD5 and 64 bytes for SHA512), calculating and exchanging checksum values does not affect the performance of FIVER noticeably unless $CHUNK\_SIZE$ is set to very small values. By default, $CHUNK\_SIZE$ is set to 256MB, but we evaluated larger and smaller values as well in Section 5.4. If integrity verification fails for a chunk of a file, then the sender will only retransfer the failed portion of the file, considerably shortening the time to recover from failures.

5. Evaluations

We tested FIVER in Chameleon, HPCLab, Pronghorn networks whose specifications are listed in Table 1. Chameleon is an academic cloud service that offers instances in two sites, Chicago, IL and Austin, Texas. Thus, we call the experiments that run transfers between the instances of the same site as Chameleon-LAN and different sites as Chameleon-WAN. For HPCLab experiments, we used two pairs of nodes as HPCLab-WS and HPCLab-DTN. HPCLab-WS involves two workstations in the same local area network with 1 Gbps network bandwidth. On the other hand, HPCLab-DTN is comprised of two data transfer nodes in the same local area network with 40 Gbps connectivity. Pronghorn is a campus cluster and its transfer nodes are connected with 10G links. Finally, we used one Pronghorn server as the sender and one Chameleon Cloud instance (in Chicago) as the receiver to test the algorithms in more wide-area network settings. We report the results in terms of overhead which is calculated as a percentage of increase in transfer duration by integrity verification algorithms compared to the execution time of the slower operation between file transfer and checksum computation as shown in Equation 1. $t_{chksum}, t_{transfer}, t_{algorithm}$ refer to the time it takes to run checksum computation, file transfer, and an integrity verification-enabled file transfer algorithm, respectively. For example, if file transfer without integrity verification takes 90 seconds, checksum computation takes 120 seconds, and FIVER completed in 130 seconds, then the overhead becomes $\frac{130-max(120,90)}{max(120,90)} = 8.3\%$.

$$Overhead = 100 \times \frac{t_{algorithm} - max(t_{chksum}, t_{transfer})}{max(t_{chksum}, t_{transfer})} \tag{1}$$

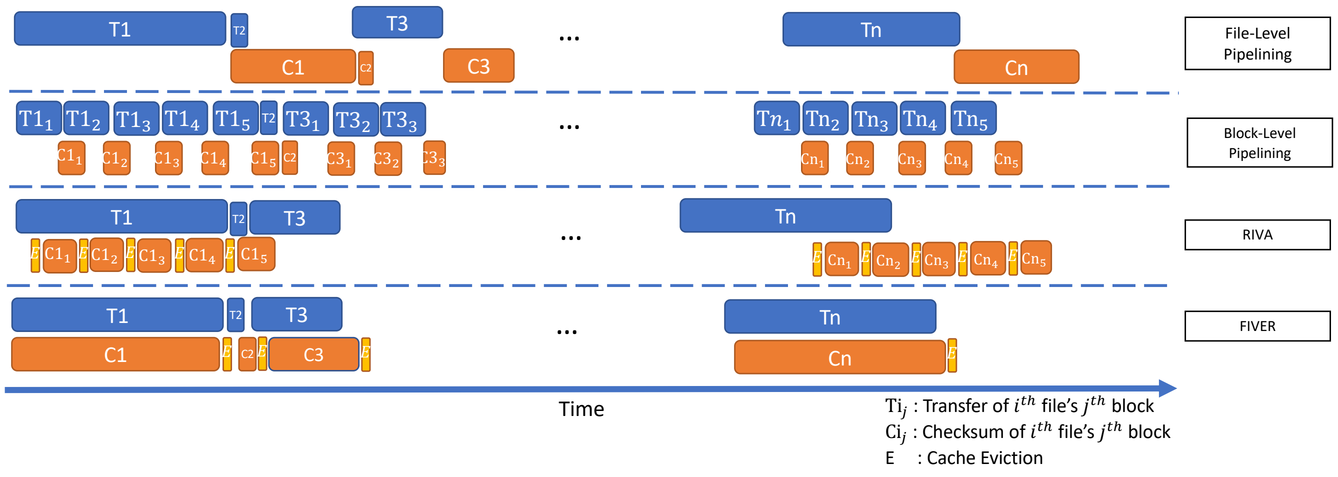
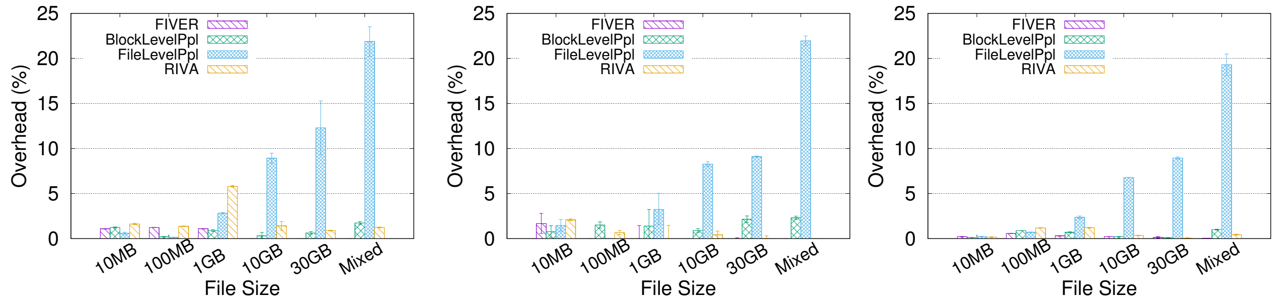

File-Level Pipelining
Block-Level Pipelining
RIVA
FIVER
Time
Ti_j : Transfer of i^th file's j^th block
Ci_j : Checksum of i^th file's j^th block
E : Cache Eviction

Figure 3: Illustration of the order of transfer (T), checksum (C), and cache eviction (E) operations in different integrity verification implementations. Note that cache eviction (E) is executed both by the sender and the receiver in RIVA, while it is only executed by the receiver in FIVER.

We run the experiments using two types of datasets; uniform and mixed datasets. Uniform datasets consist of one or more files in the same size. We created six uniform datasets where the size of files is chosen to represent small and large files in each network. Mixed datasets contain both small and large files, where files are shuffled before the transfer to randomize the order of files. Example of a mixed dataset used in Pronghorn experiments is as follows: 100x10MB, 100x50MB, 50x250MB, 10x2GB, 4x8GB, 4x10GB, 1x15GB, and 2x20GB; a total of 271 files and 165.5GB size. We repeat experiments at least six times and report the average and standard deviation values. FIVER is compared against following implementations of file transfer integrity verification:

- **File-Level Pipelining (FileLevelPpl)**: Transfer of a file is overlapped with checksum calculation of another file.
- **Block-Level Pipelining (BlockLevelPpl)**: Large files are split into small blocks (of size 256 MB, by default) and checksum calculation of a block is overlapped with the transfer of another block [8].
- **Robust Integrity Verification Algorithm (RIVA)**: Similar to BlockLevelPpl, RIVA processes large files in blocks but relaxes synchronization requirement between the checksum and transfer operations [27, 28]. This lets the transfer of a dataset to finish much earlier than its integrity verification when the transfer is faster than checksum computation. Moreover, it enforces cache eviction before calculating the checksum of file blocks to be able to detect undetected disk write errors, thus incurs significant performance overhead when disk I/O throughput is the bottleneck.

Figure 3 depicts the order of operations for different integrity verification methods. FileLevelPpl overlaps the transfer of a file with a checksum computation of another file. As a result, if files of different sizes are overlapped, the benefit of pipelining becomes marginal. BlockLevelPpl, on the other hand, tries to achieve better pipelining of transfer and checksum computation operations by dividing large files into blocks. As opposed to BlockLevelPpl, RIVA transfers files independent of the checksum computation, which could result in transfer operation to finish much earlier than integrity verification if network throughput is faster than checksum computation speed. Besides,



(a) HPCLab-WS (b) Pronghorn - Chameleon (c) Chameleon-WAN

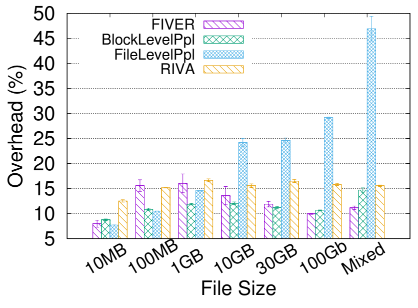
Figure 4: Comparison of algorithms in HPCLab-WS, Pronghorn-Chameleon, and Chameleon-WAN. The speed of the checksum is faster than the speed of transfer in these networks.

RIVA (both sender and receiver) also runs cache eviction for each file block before starting to compute its checksum. Finally, FIVER pipelines transfer and checksum computation operations for the same file and runs cache eviction (only receiver) after completing checksum computation but before terminating integrity verification to avoid data loss in the event of a power outage.

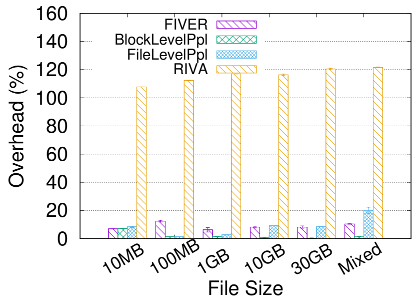
The performance of FileLevelPpl is affected by the file size distribution of the dataset as well as the speed difference between transfer and checksum operations. For example, if files are ordered in a way that 10 MB file is followed by a 10 GB file, then it will overlap transfer of 10 GB file with a checksum computation of 10 MB file which will decrease the benefit of pipelining since the transfer of 10 GB file will take much longer than checksum computation of 10 MB file. While BlockLevelPpl outperforms FileLevelPpl by dividing large files into small blocks, it has two main disadvantages. First, while it achieves better pipelining when the dataset contains very small and large files, misalignment can still occur for files that are smaller than the block size. For example, if the block size is set to 256 MB, pipelining the transfer of 5 MB file with the checksum computation of 256 MB file would result in poor pipelining. Finding the optimal block size could be challenging since small blocks will suffer from poor transfer throughput and large blocks will cause suboptimal pipelining of transfer and checksum operations. Second, dividing large files into blocks may have a negative impact on transfer throughput if the network speed is faster than checksum speed as a result of keeping data channel idle more than window size reset threshold [10].

5.1. *Overhead Analysis*

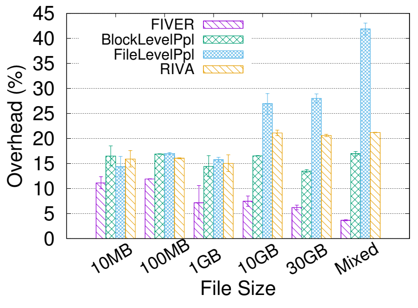
Figure 4 shows the results of experiments in networks where network speed is slower than the checksum calculation speed. Since FIVER creates two threads, one for network transfer and one for checksum computation, the thread that handles file transfer determines the baseline for the overhead calculations as shown in Equation 1. While all algorithms perform similar for small files, the overhead of FileLevelPpl becomes more than 20% for mixed files due to misalignment of transfer and checksum operations for files with different sizes as shown in Figure 4. On the other hand, FIVER is able to keep its overhead less than 3% for all uniform datasets and less than 1% for mixed datasets. FIVER outperforms BlockLevelPpl and FileLevelPpl algorithms by optimizing checksum calculation through I/O sharing. BlockLevelPpl and FileLevelPpl use system calls to open and read files to calculate their checksum, which incurs context switching overhead even though files are located in the page cache.



(a) HPCLab-DTN



(b) Chameleon-LAN



(c) Pronghorn

Figure 5: Comparison of algorithms in HPCLab-DTN, Chameleon-LAN, and Pronghorn networks. The speed of the transfer is faster than the speed of checksum.

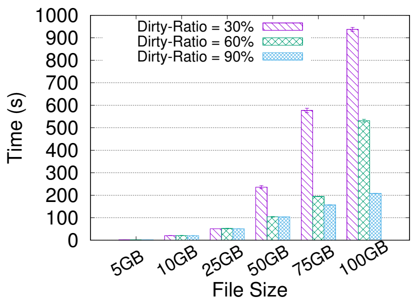
Policy	Definition	Default Value
`dirty_expire_centisecs`	Flush dirty data to disk if it has been in memory for more than this value.	**30 seconds**
`dirty_background_ratio`	If dirty data accounts for more than this policy value of total memory size, then start disk write.	**10%**
`dirty_ratio`	If dirty data accounts for more than this policy value of total memory size, then block file I/O and start disk write.	**20%**
`dirty_background_bytes`	Maximum dirty bytes that can stay in memory before disk write starts. This will only be enforced if `dirty_background_ratio` is set to zero.	**0 bytes**
`dirty_bytes`	The maximum dirty size that the memory can hold before file I/O is blocked for disk write. This policy will take effect only if `dirty_ratio` is set to zero.	**0 bytes**
`dirty_writeback_centisecs`	The frequency of `pdflush` process to check above policies and start disk write when necessary.	**5 seconds**

Table 2: Write-back caching policies. The default values are taken from Linux kernel 4.4.

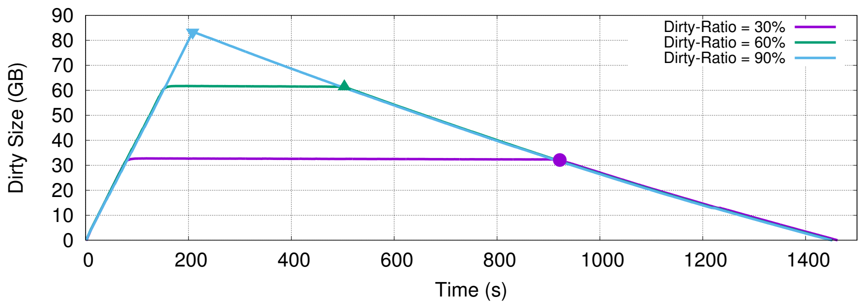
Figure 5 shows the results of experiments in networks where network throughput is faster than checksum computation throughput. Since the network bandwidth of HPCLab-DTN, Chameleon-LAN, and Pronghorn is faster (>10 Gbps as shown in Table 1) than the speed of checksum computation (around 2.4 Gbps using a single CPU core), checksum calculation becomes the bottleneck. While FIVER, BlockLevelPpl, and FileLevelPpl achieve less than 20% overhead, RIVA's overhead reaches more than 120% in Chameleon-LAN as in Figure 5(b) since its cache eviction policy when executed on slow disks takes a long time to complete. For mixed datasets, FIVER is able to keep the overhead less than 15% in all networks. The overhead of BlockLevelPpl is also around 15% in HPCLab-DTN and more than 15% in Pronghorn. The overhead of FileLevelPpl is more than 40% in HPCLab-DTN and Pronghorn networks and over 20% in Chameleon-LAN.

5.2. Data Loss Resilience

Operating systems try to improve the user experience for disk write I/O operations by caching write requests in main memory (i.e., page cache), also known as write-back caching. Cached data on the main memory is flushed to disk later at a disk write rate which is significantly slower compared to memory speed. However, write-back caching raises permanent data loss concerns since cached data can be lost when an unexpected power outage takes place. Table 2 lists policies and their default values (for Linux kernel 4.4) that regulate caching policy for disk write operations. `pdflush` is responsible to enforce the policies, which is executed once every five seconds, by default.



(a) Transfer Time

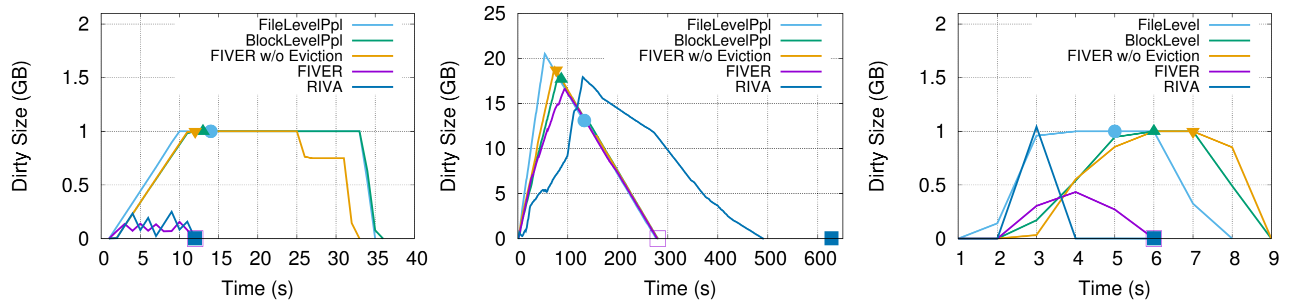


(b) Amount of dirty data size on memory

Figure 6: The impact of `dirty_ratio` on transfer time and dirty data size on memory when transferring a 100 GB file. High values lead to shorter transfer time in exchange of increase memory footprint to cache write I/O requests.

The first rule (`dirty_expire_centisecs`) defines the maximum duration a file page can stay on page cache before getting flushed to disk. If a dirty file page has been in the memory for more than this value, then `pdflush` starts to write the page to disk unless any other policy does it earlier. `dirty_background_ratio` and `dirty_ratio` define thresholds for the amount of dirty data that can be kept on memory before kernel initiates disk write. When dirty data size reaches to `dirty_background_ratio` of total memory size, the kernel will immediately start pushing dirty data to disk in the background. At this point, the kernel will still accept new write requests. However, write I/O requests arrive faster than disk write speed, the amount of dirty ratio on memory will keep growing. To avoid filling up the memory with dirty data, the kernel will stop caching write I/O requests when dirty data size reaches to `dirty_ratio` to let OS kernel to evict dirty data from memory to disk. Finally, `dirty_background_bytes` and `dirty_bytes` are used to specify dirty data size in bytes and will be effective only when the corresponding ratio value is set to 0.

To assess the impact of `dirty_ratio` on file transfer without integrity verification, we configured it to three different values (30%, 60%, and 90%) and measured transfer time and size of dirty data as shown in Figure 6. We use the Chameleon-LAN network where the receiver server is equipped with a 128 GB main memory as given in Table 1. Therefore, 30%, 60%, and 90% approximately account for 31, 62, and 93 GB after excluding memory space used by the kernel. Since page cache can hold 31 GB or more data in all three `dirty_ratio` values, their transfer times are almost the same. As the dataset size increases, however, lower `dirty_ratio` values start to experience higher transfer times since kernel blocks application I/O (in file transfer case it is blocking file write at the receiver) until it is able to bring total dirty page size on memory down to the defined threshold. As a result, transfer time of a 100GB file becomes around 200 seconds when `dirty_ratio` is set to 90% whereas it takes more than 900 seconds when it is 30%. This in turn comes at a cost of higher volume of dirty data to be kept in the main memory. Figure 6(b) demonstrates the volume of dirty data on main memory over time for a 100 GB file. When `dirty_ratio` is set to 30%, the dirty volume reaches 31 GB fairly quickly. Then, kernel throttles application I/O rate to the disk I/O rate to ensure that dirty data size does not go beyond the limit. Similar behavior is performed when the dirty volume reaches to 62 GB when `dirty_ratio` is 60%. On the other hand, when `dirty_ratio` is set to 90%, dirty data size does not reach to 93 GB which is because kernel starts to flush dirty data to disk when it reaches to `dirty_background_ratio` (10% by default) at around 25s, thus by the time dirty size reaches to 82 GB (at around 205s), kernel finished flushing 18 GB of data and no more write requests are issued. Please note that



(a) HPCLab-WS (b) Chameleon Cloud (LAN) (c) HPCLab-DTN

Figure 7: Analysis of dirty data size for file transfers with integrity verification. Points on lines mark the times that integrity verification processes is completed. For example, the integrity verification is completed at 13s for FileLevelPpl in HPCLab-WS network.

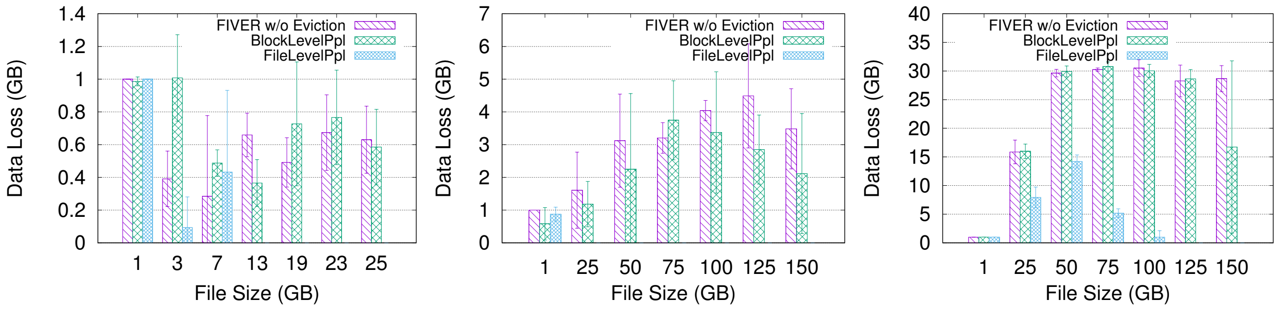
although the transfer time of 100 GB file varied for different values of `dirty_ratio`, it takes almost the same time to finish draining all dirty data to disk since disk write speed is not affected by the `dirty_ratio` values.

Figure 7 shows the volume of dirty size for the transfer of 1GB file in HPCLab-WS and HPCLab-DTN networks and 25 GB file for the Chameleon-LAN network with integrity verification. We marked the times that integrity verification is completed with different algorithms. In all networks, the dirty size is zero when integrity verification is completed by FIVER and RIVA as they enforce cache eviction before verifying its integrity. On the other hand, FileLevelPpl, BlockLevelPpl, and FIVER without cache eviction report positive numbers for dirty volume when integrity verification is completed. Therefore, a power outage that can happen after integrity verification but before finishing the synchronization of dirty pages to disk would result in data loss in such a way that transfer application would be unaware of. This poses a significant threat to the integrity of file transfers since the user will assume that the transfer was successful and will run analysis on the incomplete dataset. In particular, if the destination server experiences power loss and recovers from it quietly, then the user would be unaware of what has happened and will presume that the transfer is completed successfully despite lost content.

To validate this claim, we reproduced power loss in file transfers and measured the amount of data loss with end-to-end integrity verification. We utilized HPCLab-WS and Chameleon networks where we have physical or root access to shutdown the server/instance instantly. We applied a power shutdown immediately after the integrity of transfers is verified and measured file size after restarting the server. The amount of data loss for different file sizes is given in Figure 8. Since both FIVER and RIVA evict dirty file data from memory before finishing integrity verification, they are resilient to data loss in the event of a power outage. On the other hand, Fiver without cache eviction, BlockLevelPpl, and FileLevelPpl are vulnerable to data loss in all three networks. It is important to note that FileLevelPpl does not cause data loss for large files because checksum computation for large files takes long enough for `dirty_expire_centisecs` to kick in and flush all dirty pages to disk before completing the integrity verification process.

5.3. Dynamic Checksum and Transfer Parallelism

By default, FIVER creates one transfer and one checksum thread, however, this may lead to suboptimal utilization of network and compute resources when a single thread is insufficient. For instance, the throughput of checksum calculation is around 2.4 Gbps using a single core Intel E5-2623 core with a 2.60GHz clock rate, which



(a) HPCLab-WS (b) Chameleon-WAN (c) Chameleon-LAN

Figure 8: Amount of lost data when the power supply of the receiver is disconnected right after the integrity of the transfer is validated. FIVER and RIVA avoid data loss by employing cache eviction.

limits the overall performance in networks with over 10 Gbps throughput. Therefore, we implemented dynamic parallelism (*FIVER Dynamic*) to spawn new checksum or transfer threads to mitigate performance bottlenecks.

Algorithm 3 describes how dynamic parallelism operates. We keep track of maximum achieved transfer and checksum throughput to determine whether or not creating a new thread is improving the performance. We also define ρ_{thr} to be the ratio of average throughput of the last three intervals to the maximum observed throughput as shown in line 2. It is used to determine if creating new transfer threads help to increase transfer throughput. For example, if transfer throughput is 4.5 Gbps using three transfer threads, the algorithm will continue opening new transfer threads if using four transfer threads returns more than 5.4 Gbps (4.5 Gbps x 1.2) throughput to stop creating new threads when the gain becomes negligible. Although it's is configurable, we noticed that 20% expected gain is sufficient to take advantage of transfer parallelism while avoiding creating too many threads. In a similar way, new checksum threads are created as long as overall checksum speed increases noticeably (20% or more for each new thread) and there are enough items in the checksum queue. The latter condition ensures that new threads are only created if there is a sufficient amount of data to be processed both by existing and new threads.

We also define confidence interval to avoid transient variations in transfer and checksum throughput such that FIVER will open a new transfer or checksum threads only after it builds enough confidence ($CONF\_THRESHOLD$) on results. The threshold is configurable but we used 3 in our experiments. As monitor function is called once a second, this would set the confidence threshold to three seconds. If the throughput improvement ratio as a result of the recently created thread (ρ_{thr}) is greater than 20% (line 4), FIVER deduces that transfer throughput benefits from parallelism and creates a new transfer thread after building enough confidence to further enhance performance. Increasing the number of transfer threads could also improve checksum throughput since checksum throughput can only go as fast as transfer thread due to the producer-consumer relationship. If adding a new checksum or transfer thread does not improve the performance, FIVER assumes that it has reached to maximum performance thus stops adding more threads since doing so will only overload system resources (lines 4 and 12).

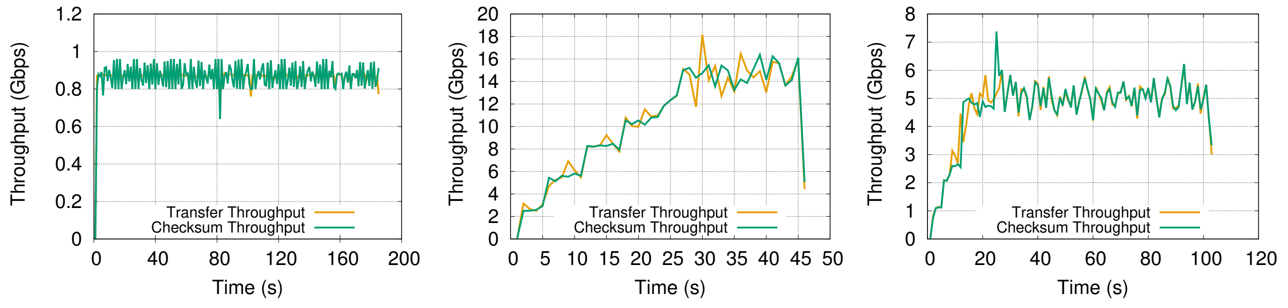
Algorithm 3: Dynamic parallelism algorithm for checksum and transfer operations in FIVER Dynamic

```
Function DynamicParallelism(curChecksumThr, curTransferThr, checksumQueue):
    Data: maxTransferThr, maxChecksumThr
    $\rho_{thr} = \frac{curTransferThr}{maxTransferThr}$
    $\rho_{chk} = \frac{curChecksumThr}{maxChecksumThr}$
    if $\rho_{thr} > 1.2$ then
        if thr-confidence ==CONF_THRESHOLD then
            createNewTransferThread()
            thr-confidence= 0
        else
            thr-confidence++
        end
        maxTransferThr = curTransferThr
    if $\rho_{chk} > 1.2$ & checksumQueueu is at least half full then
        if chk-confidence ==CONF_THRESHOLD then
            createNewChecksumThread()
            chk-confidence= 0
        else
            chk-confidence++
        end
        maxChecksumThr = curChecksumThr
```

We evaluated the dynamic parallelism algorithm in HPCLab-WS, HPCLab-DTN, and Pronghorn networks as shown in Figure 9. Single-threaded checksum computation can only reach around 3 Gbps speed and becomes the bottleneck in HPCLab-DTN and Pronghorn networks. Thus, dynamic parallelism creates new checksum threads and increases checksum computation throughput to keep up with transfer speed. At the same time, transfer parallelism is increased to improve transfer throughput. It keeps adding more checksum and transfer threads until no further improvement is observed, which happens at around 28s in HPCLab-DTN and 26s in Pronghorn. The dynamic parallelism algorithm increases FIVERs overall throughput from 3 Gbps to around 16 Gbps and 5.5 Gbps for HPCLab-DTN and Pronghorn, enabling 5x improvement over single-threaded implementation. On the other hand, it is also possible adding new transfer and checksum threads may not improve the performance when a single thread is sufficient to utilize available resources. For example, FIVER's performance in HPCLab-WS is limited to 800 Mbps due to bandwidth limitations. Unaware of this limitation, dynamic parallelism opens up a new transfer thread at around 5s as shown in Figure 9(a). After realizing that this change does not improve transfer performance, dynamic parallelism terminates the search immediately and keeps using a single thread.

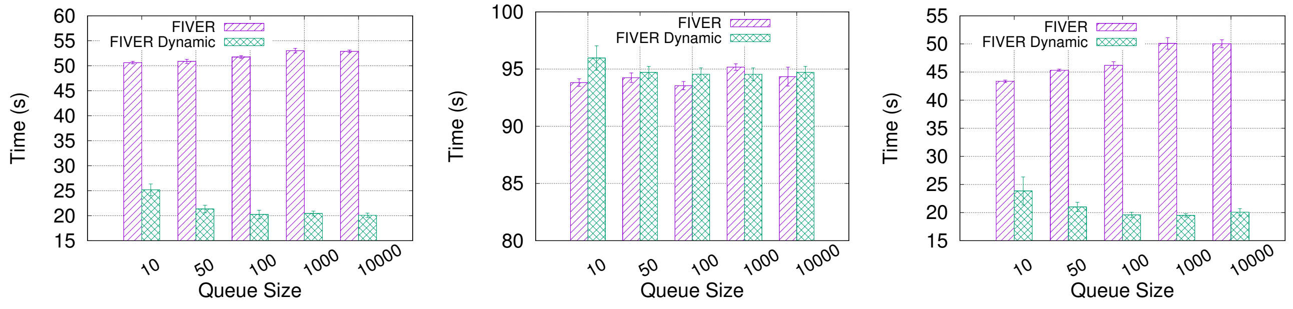
5.4. Impact of Queue Size and Chunk Size on Performance

As pointed in Algorithm 1 and 2, transfer and checksum threads have a producer-consumer relationship through a shared queue, where transfer thread adds file data into queue and checksum thread pulls from the queue to compute



(a) HPCLab-WS (b) HPCLab-DTN (c) Pronghorn

Figure 9: Dynamic parallelism helps to increase throughput from around 3 Gbps to 18 and 7 Gbps in HPCLab-DTN and Pronghorn networks. Since HPCLab-WS servers are connected with a 1G link where using single network and checksum threads is sufficient, dynamic parallelism does not improve the performance.

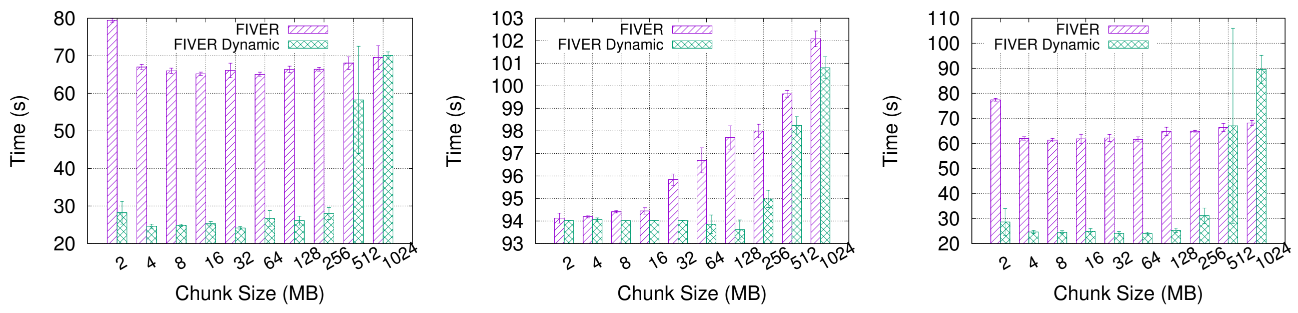


(a) Pronghorn (b) HPCLab-WS (c) HPCLab-DTN

Figure 10: Impact of the queue size on the performance of FIVER and FIVER Dynamic.

the checksum. Therefore, we evaluated the effect of queue size on the performance as shown in Figure 10. The results indicate that smaller queue size values yield the best performance for FIVER. However, when dynamic parallelism is enabled (FIVER Dynamic), multiple threads start to read and write from/to the same queue, necessitating a large queue to avoid filling it up quickly. Yet, setting queue size to large values without knowing how many threads that dynamic parallelism will create could lead to inefficient use of memory space. Thus, FIVER dynamically adjusts the queue size depending on the number of active threads. Specifically, it starts with a queue size of 10, then increases it by 10 for every new thread.

We also analyzed the impact of the integrity verification chunk size ($CHUNK\_SIZE$) on the performance in



(a) Pronghorn (b) HPCLab-WS (c) HPCLab-DTN

Figure 11: Impact of chunk size ($CHUNK\_SIZE$) on the performance of FIVER and FIVER Dynamic.

Figure 11. Chunk size defines the data size that checksum thread will digest before starting to verify its integrity as described in line 4 of Algorithm 2. The results reveal that a chunk size of 4 MB yields the optimal performance in all networks for both FIVER and FIVER Dynamic. Lower (i.e., 2 MB) or larger (e.g., 1024 MB) values result in a more than 30% increase in execution time for FIVER. Larger chunk size values (e.g., 1024 MB) causes the execution time to nearly triple in Pronghorn and HPCLab-DTN networks when FIVER Dynamic is utilized.

5.5. Impact of Hash Algorithm

Although MD5 is still widely used, it has been found weak to collision attacks[37]. Hence, we measured execution times for MD5, SHA1, and SHA256 hash algorithms as shown in Figure 12. We transferred a mixed dataset that is used in Figure 4.

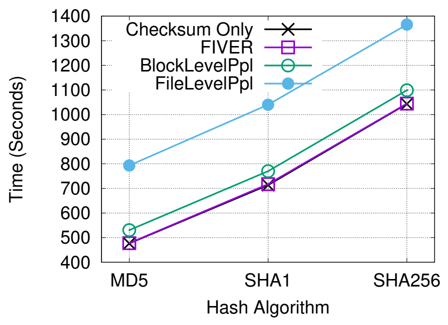


Figure 12: Impact of hash algorithms on the execution time of integrity verification algorithms.

As expected, the time spent on checksum computation is proportional to the complexity of hash algorithm. For example, checksum computation without data transfer (*Checksum Only*) took 476, 713, and 1043 seconds for MD5, SHA1 and SHA256 algorithms, respectively. It took more than twice time to compute the hash of files with SHA256 than it took when using MD5. As a result, total execution time of integrity verification algorithms also increased drastically. However FIVER still imposes the lowest overhead compared to BlockLevelPpl and FileLevelPpl. If we take *Checksum Only* as a baseline, BlockLevelPpl induces 50-60 seconds delay and FileLevelPpl induces 300 seconds delay.

5.6. Efficient Error Recovery

Table 3 shows the execution times of FIVER and BlockLevelPpl when a dataset with 15 large files (10x1GB files and 5x10GB files) is transferred in the HPCLab-DTN network when integrity verification detects checksum mismatches. We injected faults by flipping a random bit of randomly-chosen files during the transfer operation. $CHUNK\_SIZE$ is chosen to be the same size as the block in BlockLevelPpl (i.e., 256 MB). When the chunk size is set to the size of the file (i.e. FIVER with file-level integrity verification), the execution time increases drastically since it has to transfer the whole file again. Its execution time almost doubles in case of 24 integrity verification failures compared to no-failure case. On the other hand, when the chunk size is set to 256MB, FIVER is able to handle faults by only transferring a small portion of the file due to which its execution time and total transferred data size increase slowly. Moreover, running integrity verification in the chunk level does not degrade the execution time when there is no fault observed (i.e., fault count is 0) compared to running integrity verification for the whole file, showing the effectiveness of conducting integrity verification for small data sizes.

Fault	**FIVER**		**Block-Level**
Count	File Int. Ver.	Chunk Int. Ver.	**Pipelining**
0	179.2s	180.2s	204.2s
8	253.1s	186.2s	208.8s
24	347.3s	198.5s	222.3s

Table 3: Execution times of algorithms in the presence of data corruption. FIVER's chunk-level integrity verification offers an efficient error recovery mechanism by resending only a small portion of files.

6. Conclusion and Future Work

End-to-end integrity verification is vital for many applications that are sensitive to data corruption. However, it could degrade transfer performance significantly due to compute and I/O intensive checksum computation. In this paper, we propose FIVER to minimize the cost of integrity verification by overlapping checksum computation and data transfer operations. The results show that FIVER reduces the overhead of running integrity verification from up-to 120% by the state-of-the-art solutions to less than 15% in all network settings. In addition, FIVER is resilient to data loss in the case of an unexpected power outage by enforcing cache eviction before validating the integrity of transfers. Furthermore, we implemented dynamic parallelism to detect and mitigate performance bottlenecks for integrity verification-enabled transfers, which improves overall performance by more than 5x with the help of multi-threaded checksum computation and multi-channel network transfers. As a future direction, we will investigate taking advantage of in-network compute powers to even further reduce the overhead of integrity verification. In particular, smartNICs offer compute resources that can be utilized to compute the checksum of files such that end-to-end integrity verification can be executed without causing performance slowdowns or consuming too much CPU power of end servers. We will further investigate the use of blockchain technology to store the checksum of the scientific dataset in a distributed database such that users can download files from third-party servers and check their integrity using blockchain transactions.

Acknowledgment

This project is in part sponsored by the National Science Foundation (NSF) under award number OAC-1850353. Some of the results presented in this paper were obtained using the Chameleon testbed supported by the NSF.

References

[1] ATLAS, https://home.cern/about/experiments/atlas (2017).

[2] Large Synoptic Survey Telescope, https://www.lsst.org/ (2017).

[3] Dark Energy Survey, https://www.darkenergysurvey.org/ (2020).

[4] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, et al., HACC: Simulating sky surveys on state-of-the-art supercomputing architectures, New Astronomy 42 (2016) 49–65.

[5] J. Stone, C. Partridge, When the CRC and TCP checksum disagree, in: ACM SIGCOMM computer communication review, Vol. 30, ACM, 2000, pp. 309–319.

[6] R. Kettimuthu, Z. Liu, D. Wheeler, I. Foster, K. Heitmann, F. Cappello, Transferring a Petabyte in a Day, Future Generation Computer Systems.

[7] Globus, https://www.globus.org/ (2020).

[8] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, M. Papka, Towards optimizing large-scale data transfers with end-to-end integrity verification, in: Big Data (Big Data), 2016 IEEE International Conference on, IEEE, 2016, pp. 3002–3007.

[9] E. Arslan, B. A. Pehlivan, T. Kosar, Big data transfer optimization through adaptive parameter tuning, Journal of Parallel and Distributed Computing 120 (2018) 89–100.

[10] TCP Congestion Control, https://tools.ietf.org/html/rfc2581.

[11] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song, Provable data possession at untrusted stores, in: Proceedings of the 14th ACM conference on Computer and communications security, Acm, 2007, pp. 598–609.

[12] Y. Zhu, H. Hu, G.-J. Ahn, M. Yu, Cooperative provable data possession for integrity verification in multicloud storage, IEEE transactions on parallel and distributed systems 23 (12) (2012) 2231–2244.

[13] C. Liu, C. Yang, X. Zhang, J. Chen, External integrity verification for outsourced big data in cloud and IoT: A big picture, Future generation computer systems 49 (2015) 58–67.

[14] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. Rosenthal, M. Baker, The LOCKSS peer-to-peer digital preservation system, ACM Transactions on Computer Systems (TOCS) 23 (1) (2005) 2–50.

[15] M. Vigil, J. Buchmann, D. Cabarcas, C. Weinert, A. Wiesmaier, Integrity, authenticity, non-repudiation, and proof of existence for long-term archiving: a survey, Computers & Security 50 (2015) 16–32.

[16] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, M. K. Mckusick, Ffsck: The fast file-system checker, ACM Transactions on Storage (TOS) 10 (1) (2014) 2.

[17] Y. Zhang, D. S. Myers, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Zettabyte reliability with flexible end-to-end data integrity, in: Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on, IEEE, 2013, pp. 1–14.

[18] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, End-to-end data integrity for file systems: A ZFS case study., in: FAST, 2010, pp. 29–42.

[19] M. U. Arshad, A. Kundu, E. Bertino, A. Ghafoor, C. Kundu, Efficient and scalable integrity verification of data and query results for graph databases, IEEE Transactions on Knowledge and Data Engineering 30 (5) (2018) 866–879.

[20] R. Hasan, R. Sion, M. Winslett, The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance., in: FAST, Vol. 9, 2009, pp. 1–14.

[21] C.-L. Chen, M. Hsiao, Error-correcting codes for semiconductor memory applications: A state-of-the-art review, IBM Journal of Research and Development 28 (2) (1984) 124–134.

[22] F. Qin, S. Lu, Y. Zhou, SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs, in: High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, IEEE, 2005, pp. 291–302.

[23] J. Meza, Q. Wu, S. Kumar, O. Mutlu, Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field, in: Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on, IEEE, 2015, pp. 415–426.

[24] T. J. Dell, A white paper on the benefits of chipkill-correct ECC for PC server main memory, IBM Microelectronics Division 11.

[25] S. Xiong, F. Wang, Q. Cao, A bloom filter based scalable data integrity check tool for large-scale dataset, in: Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS), 2016 1st Joint International Workshop on, IEEE, 2016, pp. 55–60.

[26] E. Arslan, A. Alhussen, A low-overhead integrity verification for big data transfers, in: 2018 IEEE International Conference on Big Data (Big Data), IEEE, 2018, pp. 4227–4236.

[27] B. Charyyev, E. Arslan, Riva: Robust integrity verification algorithm for high-speed file transfers, IEEE Transactions on Parallel and Distributed Systems 31 (6) (2020) 1387–1399.

[28] B. Charyyev, A. Alhussen, H. Sapkota, E. Pouyoul, M. H. Gunes, E. Arslan, Towards securing data transfers against silent data corruption, in: IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing, IEEE/ACM, 2019.

[29] T. Kosar, E. Arslan, B. Ross, B. Zhang, Storkcloud: Data transfer scheduling and optimization as a service, in: Proceedings of the 4th ACM workshop on Scientific cloud computing, ACM, 2013, pp. 29–36.

[30] E. Arslan, T. Kosar, High Speed Transfer Optimization Based on Historical Analysis and Real-Time Tuning, IEEE Transactions on Parallel and Distributed Systems.

[31] Y. Liu, Z. Liu, R. Kettimuthu, N. Rao, Z. Chen, I. Foster, Data transfer between scientific facilities–bottleneck analysis, insights and optimizations, in: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2019, pp. 122–131.

[32] E. Yildirim, E. Arslan, J. Kim, T. Kosar, Application-level optimization of big data transfers through pipelining, parallelism and concurrency, IEEE Transactions on Cloud Computing 4 (1) (2015) 63–75.

[33] I. Alan, E. Arslan, T. Kosar, Energy-aware data transfer algorithms, in: Proceedings of SC'15, SC '15, ACM, New York, NY, USA, 2015, pp. 44:1–44:12.

[34] E. Arslan, K. Guner, T. Kosar, HARP: Predictive Transfer Optimization Based on Historical Analysis and Real-Time Probing, in: Proceedings of SC'16, SC '16, IEEE Press, Piscataway, NJ, USA, 2016, pp. 25:1–25:12.

[35] E. Yildirim, E. Arslan, J. Kim, T. Kosar, Application-level optimization of big data transfers through pipelining, parallelism and concurrency, IEEE Transactions on Cloud Computing 4 (1) (2016) 63–75.

[36] Z. Liu, R. Kettimuthu, I. Foster, N. Rao, Cross-geography scientific data transfer trends and user behavior patterns, in: 27th ACM Symposium on High-Performance Parallel and Distributed Computing, HPDC, Vol. 18, 2018, p. 12.

[37] X. Wang, D. Feng, X. Lai, H. Yu, Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD., IACR Cryptology ePrint Archive 2004 (2004) 199.