University of Nevada, Reno

Protecting File Transfers Against Silent Data Corruption with Robust End-to-End Integrity Verification

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in
Computer Science and Engineering

by

Batyr Charyyev

Mehmet Hadi Gunes/Thesis Advisor

Engin Arslan/Thesis Co-Advisor

August, 2019

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

BATYR CHARYYEV

entitled

Protecting File Transfers Against Silent Data Corruption with Robust End-to-End Integrity Verification

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Mehmet Hadi Gunes, Advisor

Engin Arslan, Co-Advisor and Committee Member

Mohammed Ben-Idris, Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

August 2019

ABSTRACT

Scientific applications generate large volumes of data that often needs to be moved between geographically distributed sites which has led to a significant increase in data transfer rates. As an increasing number of scientific applications are becoming sensitive to silent data corruption, end-to-end integrity verification has been proposed. End-to-end integrity verification minimizes the likelihood of silent data corruption by comparing checksum of files at the source and the destination using secure hash algorithms such as MD5 and SHA1. However, existing implementations of end-to-end data integrity verification for file transfers compute checksum of files based on memory copy (i.e. cache) of the file, thus fall short to detect silent disk errors that take place while writing cached data to disk. In this thesis, we inspect the robustness of existing end-to-end integrity verification approaches against silent data corruption and propose a Robust Integrity Verification Algorithm (i.e. RIVA) to enhance data integrity. Extensive experiments show that unlike existing solutions, RIVA is able to detect silent disk corruptions by invalidating file contents in page cache and reading them directly from disk. Since RIVA clears page cache and reads file contents directly from the disk, it incurs delay to execution time. However, by running transfer, cache invalidation, and checksum operations concurrently, RIVA is able to keep its overhead below 15% in most cases compared to the state-of-the-art solutions in exchange of more secure file transfers. We also introduce a novel fault injection mechanism to assesses the robustness of RIVA against undetected disk errors by altering file content on the disk. Finally, we present dynamic parallelism to adjust the number of transfer and checksum threads to overcome performance bottlenecks. The results show that dynamic parallelism lead to more than 5x increase in RIVA's speed.

TABLE OF CONTENTS

List of Tables . . . . . **iv**

List of Figures . . . . . **v**

Chapter 1: Introduction **1**

Chapter 2: Background **5**

2.1 Undetected Disk Errors (UDEs) 5

2.2 Storage errors causing UDEs and manifestation 7

2.3 End-to-end Integrity Verification 9

Chapter 3: Literature Review **12**

3.1 High-performance data transfers 12

3.2 Integrity verification 13

3.3 Data corruption in storage systems 13

Chapter 4: System Design of Robust Integrity Verification Algorithm **16**

Chapter 5: Experimental Results **20**

5.1 Evaluation Results 20

5.1.1 Fault Injection 26

5.1.2 Impact of Block Size on Performance 28

5.1.3 Dynamic Checksum and Transfer Parallelism 29

5.1.4 Statistical Analysis on Likelihood of UDEs with RIVA 33

Chapter 6: Conclusion and Future Work 38

References 44

LIST OF TABLES

5.1 System specification of networks. 21

5.2 The results of extreme-injection experiment in HPCLab-WS network with 16GB system memory. RIVA is able to detect all injected faults whereas FileLevelPpl can only detect it for a 50GB file. 27

5.3 Examples of UDEs that affect I/O during the transfer of a two bit data originally stored in the disk of the sender as *"00"*. Out of four I/O operations (two for transfer and two for integrity verification), RIVA is able to capture all single and multi bit errors which that affects only one I/O operation. However, RIVA may miss UDEs if they occur in multiple I/O operations in a way that sender and receiver checksum return same value. 34

LIST OF FIGURES

2.1 When undetected read errors (i.e., UREs) happen at sender or receiver, it will cause checksum mismatch and re-transfer of the file. 5

2.2 When undetected write errors (i.e., UWEs) happen for small files, it will be missed since checksum will be computed based on memory copy of pages. . 6

2.3 Checksum computation processes at sender and receiver servers read files from memory when file sizes are smaller than free memory space. 10

2.4 Transferring a mixed dataset with the state-of-the-art integrity verification algorithm reveals that only files that are larger than memory size contribute to page misses during the checksum computation. 11

4.1 RIVA extends end-to-end integrity verification coverage to ensure data integrity between receiver memory and disk. 16

4.2 System architecture of RIVA. When *Checksum* thread attempts to read UWE-exposed page-3, it will trigger a page miss since *Cache Evictor* has evicted it. Consequently, page-3 will be read from the disk and the UWE will be captured. 17

5.1 Performance comparison of algorithms in LAN experiments. While RIVA is able to keep its overhead below 17% in HPCLab and Pronghorn networks, slow disk speed and large memory size in Chameleon Cloud deteriorates its performance significantly. 22

5.2 Performance comparison of algorithms in WAN experiments. RIVA is able to keep its overhead below 12% in all networks. 23

5.3 RIVA yields high page miss rates as it evicts file pages from memory and reads directly from disk to capture UWEs. 25

5.4 Extreme-injection test involves injecting UWEs to all pages of files to determine if a transfer application reads any data from page cache. 26

5.5 While block size has negligible impact in most networks, it affects execution time by around 70% in Pronghorn and up-to 30% in Chameleon Cloud. 28

5.6 Disk I/O throughput for different block size values for the transfer of a 30GB file in Chameleon Cloud network. 30

5.7 RIVA can dynamically adjust the number of transfer and checksum threads to overcome bottlenecks. 32

CHAPTER 1
INTRODUCTION

Large scientific experiments such as internet measurements [1, 2, 3], complex network modeling [4, 5, 6, 7], epidemic analysis [8], genetic disease analysis [9, 10] environmental and coastal hazard prediction [11], climate modeling [12, 13], genome mapping [14], and high-energy physics simulations [15, 16] generate data volumes reaching petabytes per year. This massive amount of the data often needs to be moved for various purposes including processing, collaboration, and archival. Most of the earlier works focused on the optimization of the data transfers [17, 18, 19]. However, the integrity of transfers are also critical for many applications such as Dark Energy Survey [20] and Sky Survey Simulation [21] as they rely on correctness of the data to operate.

As data transfer rates are rapidly increasing, legacy integrity verification mechanisms fall short to detect corruption. Although some of the data transfer components have built-in integrity verification mechanisms, they are either weak or applicable to a subset of available systems. For instance, TCP uses 16-bit checksum to capture data corruption but it fails to detect errors once in 16 million to 10 billion packets [22], which is not rare for big scientific data transfers.

In addition to system and network faults [23, 24], data corruption can also happen at the storage systems during file read and write operations as disk drives suffer from a significant number of silent data corruptions, referred as undetected disk error (UDE) [25, 26, 27]. UDEs occur mainly due to firmware or hardware malfunctions in disk drives, and silently corrupt data without being detected by the disk. Modern drives have advanced error detection and correction mechanisms. For instance, they can recover from certain internal errors, they can be instructed to remap logical to physical sectors to avoid problem

areas on the platters, and they can usually report when the data they are storing is no longer readable (e.g., in the case of an uncorrectable read error, or hard error). Disk drive manufacturers specify reliability parameters, such as MTTF (mean time to failure) or hard-error rate. Storage systems implement several approaches to detect and recover from UDEs through file system scrubbing, RAID reconstruction, however these are costly operations and recovery may not always be possible [26]. UDEs are categorized into two groups, undetected write error (UWE) and undetected read error (URE). UREs manifest as transient errors, and are unlikely to affect system state beyond causing transfer repetitions. UWEs, on the other hand, are persistent errors which are only detectable during a read operation subsequent to the faulty write, and thus posing a significant threat to data reliability [25].

Application-layer end-to-end integrity check is proved to be a robust solution to detect and recover from UDEs as it covers complete path of operations [28, 29, 30, 31, 32, 33]. A typical implementation of end-to-end integrity verification for data transfers works as follows: Sender first reads the file from the disk and sends it to the destination. Once data transfer is completed, the sender reads the file again to compute checksum using a hash algorithm such as MD5 and SHA1. At the receiver side, the file is first streamed from network interface and written to the storage. The file is read back to compute its checksum, which is sent to the sender. Finally, the sender compares the destination's checksum against its own copy to verify integrity. If the checksum values of source and destination servers are the same, then the transfer is marked as successful. Otherwise, the file at the destination is assumed to be corrupt and the transfer is restarted. If the dataset consists of multiple files, then the transfer of next file will begin only after the current file's integrity verification is completed successfully.

Several approaches are proposed to optimize the execution time of transfers when end-to-end integrity verification is enabled including file-level pipelining [34], block-level pipelining [35], and FIVER [36]. In file-level pipelining checksum of a file is computed

while another file is being transferred. However, it fails to overcome performance issues due to two main reasons. First, it is hard to achieve perfect pipelining of data transfers and checksum computation if the file sizes in datasets are different or the speed of transfer and checksum computation operations is different. Secondly, running checksum computation and transfer operations of two different files simultaneously may cause I/O contention due to which both processes may experience slow down. Block-level pipelining is proposed to address the limitations of the file-level pipelining by dividing large files into smaller blocks to better overlap transfers and checksum operations for datasets with mixed file sizes. However, it requires an upfront work to determine the block size that achieves optimal pipelining for different network and host configurations. Moreover, dividing large files into smaller blocks could deteriorate transfer throughput when network transfer runs faster than checksum computation and have to stay idle while checksum calculation is going on which may trigger TCP window size reset for every block transfer [37]. Arslan et. al. proposed FIVER [36] that overlaps transfer and checksum operations for the same file to improve execution time by reading file once and use for both operations. Although FIVER reduces execution time significantly, it causes transfers to be vulnerable against UDE. In fact, experiments shows that UDE are also missed by blocks-level pipelining, file-level pipelining, and sequential approaches in most cases.

Existing implementations of end-to-end integrity verification for data transfers are vulnerable to receiver-side UWEs due to calculating checksum using cached copy of files. When a file is recently written or read, the OS kernel keeps the file blocks in the main memory to optimize subsequent accesses. This causes file reads to operate on cached copies of file pages, which might be different than the disk copies in case of UWE. Even though file system scrubbing and RAID re-construction techniques could potentially catch such errors, they can only be executed in the order of days or weeks due to incurred overhead, causing file transfers to miss UWEs and accept files with corruption.

In this thesis and in our previous works [38][39], we propose a Robust Integrity Verification Algorithm (RIVA) to ensure that checksum calculations of transferred files operate on disk copy to detect silent data corruption that may occur when flushing data from memory to disk. It works as follows: When transfer of a file is completed, RIVA first finds the virtual address space range of the file. Then, it deletes the mappings for the specified address range such that further references to the address range will generate invalid memory references (i.e., page faults). Finally, the file is read to calculate its checksum which requires I/O operations to go to the disk, allowing the detection of UWEs.

The rest of the thesis is organized as follows: Chapter 2 describes integrity verification of the data transfer and silent corruption scenarios. Chapter 3 presents the related work and Chapter 4 details design principles of the proposed algorithm and experimental results. Finally, we conclude with a summary and future work in Chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, we provide background on the end-to-end integrity verification and undetected disk errors, causes and manifestations.

2.1 Undetected Disk Errors (UDEs)

There are two classes of undetected disk errors; undetected read error (URE) and undetected write error (UWE). UREs causes applications to see a different version of data than the one stored on the disk. In Figure 2.1, while disk hosts correct file page, URE leads to corrupted file page n to be served to the data transfer application. When UREs happen during the checksum operation at the sender or the receiver, it will trigger an integrity verification failure and re-transfer of the file as the checksum calculated by the sender and the receiver servers will not match. In a very unlikely case, both the sender and the receiver servers may be exposed to the same URE and it will cause the URE to go undetected harmlessly. Otherwise, integrity verification will fail, and the file will be transferred again.

On the other hand,undetected write errors (i.e., UWEs) could easily pass integrity ver-

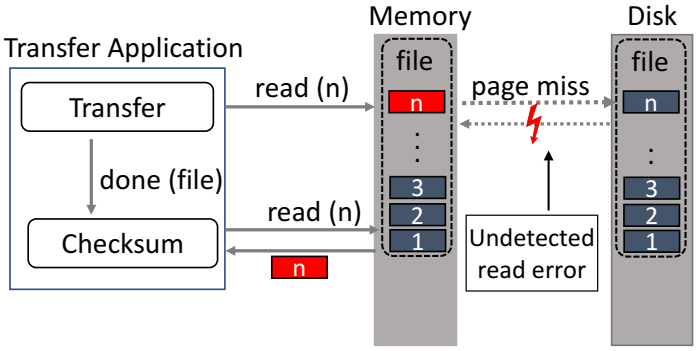


Figure 2.1: When undetected read errors (i.e., UREs) happen at sender or receiver, it will cause checksum mismatch and re-transfer of the file.

ification and corrupted data will be assumed as correct. Unlike UREs, UWEs could only happen at the receiver since file transfers do not require write operations at the sender. Figure 2.2 illustrates how UWE occurs for small files at the receiver server. Transfer application first sends the file from the sender to the receiver. While writing file to the disk, UWE corrupts page content which goes undetected. Unaware of the UWE, the OS keeps the correct copy in memory (i.e., page cache) to optimize future accesses. Upon completion of the file transfer and write operations, checksum thread starts to read the file to compute the checksum. Since the OS holds the cached copies of the file pages in the memory, it will serve checksum read requests the page cache. This causes checksum to be calculated on the correct data while the disk holds corrupted data.

While UREs are transient and can easily be tackled by the retransmission of files, UWEs cause permanent impact by accepting corrupted data as genuine. Thus, it is necessary for checksum computation to read files from the disk to capture UWEs. Note that UWEs may be harmless if files are read only once immediately after their transfer. However in many cases, large datasets are transferred completely first and processed afterwards which would cause UWEs to go undetected. When a client accesses the file after some time, correct file copies will not be located in the memory and corrupt data will be served from the disk. Although ECC can detect/fix single/double bit errors and S.M.A.R.T can detect some of

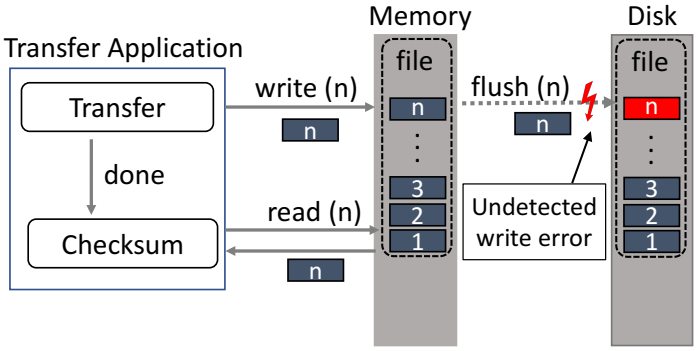


Figure 2.2: When undetected write errors (i.e., UWEs) happen for small files, it will be missed since checksum will be computed based on memory copy of pages.

the disk errors [25, 40, 26] explain that ECC and S.M.A.R.T are not guaranteed to capture all silent errors.

Even if undetected write errors are not observed as much as other kind of errors, Li et al. estimate that the probability of an UWE is between 10^{-12} and 10^{-14}, once in every 1-100 terabytes [40]. While parity-enabled RAID architectures are more resilient against disk failures and latent sector errors, previous studies found that existing implementations are also susceptible to silent data corruption [25, 26, 27]. Although several techniques have been proposed to prevent silent errors in RAID systems, the empirical results show that they incur up-to 43% performance overhead [41]. Moreover, even if disks are error-proof, data corruption can still happen during data transmission from memory to disk due to faulty cables or firmware bugs. In fact, researchers observed up-to 5% checksum mismatch when one petabyte data is transferred between two HPC clusters with parti-enabled RAID filesystem, where existing integrity measures failed to detect/recover [42].

2.2 Storage errors causing UDEs and manifestation

There exist different types of storage errors such as Latent Sector Errors (LSEs), Lost, Torn and Off-track writes [26, 40, 27]. LSEs are caused by physical problems within the disk drive, such as media scratches and mostly detected by drives internal error-correcting codes (ECC) [26]. Lost writes also referred as dropped writes occur when the write head fails to overwrite the data already present on a track causing the disk remain in its previous state as if the write never occurred [40]. In torn writes disk drives end up writing only a portion of the sectors in a given write request and the rest of the sector contains stale data. This often occurs when the drive is power-cycled in the middle of processing the write request [27]. Off-track writes also referred as misdirected writes occur when write head is not properly aligned with the track. When the data is written in the gap between tracks, adjacent to the intended track it is called near off-track write and when data is written even

further off-track, such that it corrupts data in another track entirely it is called far off-track write [40]. LSEs occur more frequent compared to other errors affecting about 19% of nearline and about 2% of enterprise class disks within 2 years of use [27]. Luckily they can be detected by storage systems. However, lost, torn and off-track writes can go undetected causing Undetected Disk Errors (i.e. UDEs). Lost or off-track writes occur in about 0.04% of nearline and 0.007% of enterprise class disks within the first 17 months of use whereas these values for torn writes are 0.6% and 0.06% respectively [27].

It is difficult to detect errors causing UDEs because when they occurs the drive does not report errors and the state of the drive is as expected. To overcome this problem, widely accepted approach is appending data integrity segment to each block of the data [26]. Data integrity segment contains a checksum of the entire 4 KB file system block and block identity information which is the disk blocks identity within the file system. The checksum and identity information are cross-checked when data is read to ensure that there is no corruption and the block being read belongs to the file being accessed. UDE manifestations are checksum mismatches, identity discrepancies and parity inconsistencies [26, 40]. Checksum mismatches can be caused by any data corruption and corrupted block can usually be restored through RAID re-construction. Identity discrepancies refers to inconsistency in identity of the data block. The reason might be lost write when a write destined for disk is not written but thought of as written or a misdirected write, when the original disk location is not updated. Parity inconsistency is mismatch in between the parity computed from data blocks and the parity stored on disk.

Existing protection mechanisms from UDEs are RAID level protection, Data scrubbing, Write-Verify, Version mirroring, Checksum mirroring and Write cache [27, 25]. In Raid level protection data is read from all disks and in data scrub, each physical disk block is read and checksum is computed over its data and compared to checksum located in its data integrity segment. Write-verify is aimed to protect from lost writes by reading the disk

block back after it is written and verify write operation by comparing the data already in memory. Version mirroring addresses parity pollution after a lost write by keeping version number for each block and incrementing it with every write to the block. In checksum mirroring the checksum of each data block is attached to the end of the neighboring block in the same stripe and also appended to the end of each of the parity blocks. Write cache is similar to the write-verify but the data is held in the cache even after being written to disk and when the data needs to be evicted from the cache, the data is first read from disk and compared to the cached copy.

2.3 End-to-end Integrity Verification

The simple sequential approach implements integrity verification in three steps. In the first step, a file is transferred from source to destination using preferred transfer application. Once the transfer is completed and it is written to the storage at the destination, the checksum of original file at the source and the transferred copy at destination are computed using a hash function such as MD5 or SHA1. In the third and final step, checksum values of the original file and the transferred copy are exchanged between the source and the destination servers to compare. If the checksum values are the same, then the file transfer is marked as completed. Otherwise, the transferred copy of the file is assumed to be corrupt and whole process is restarted from the beginning.

The main objective of running end-to-end integrity check is to detect possible data corruption by comparing the checksum of the file at the source and the 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 successive accesses to the file content. This causes checksum thread to use cached copy of the file pages for small files, preventing the detection of silent data corruption that may occur during disk write operations. Figure 2.3 illustrates hit ratio at the sender and receiver side when

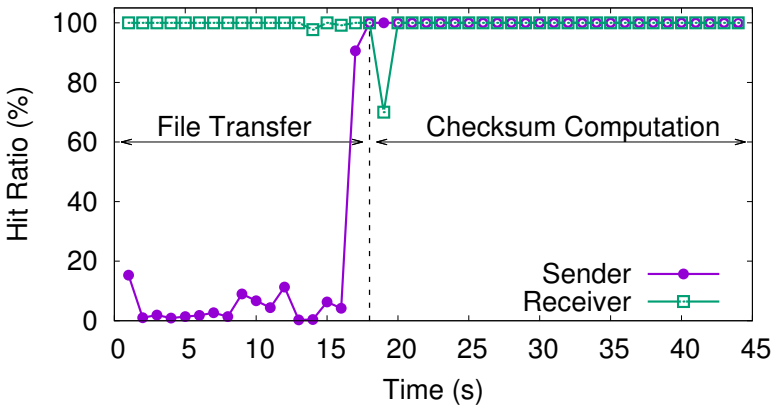


Figure 2.3: Checksum computation processes at sender and receiver servers read files from memory when file sizes are smaller than free memory space.

data is transferred with sequential integrity verification approach. Checksum computation processes at sender and receiver servers read files from memory when file sizes are smaller than free memory space. Note that on the sender side when file is transferred hit ratio is small because sender fetches data from disk and transfers it. However after transfer is completed hit ratio is almost 100% because data is cached on the memory. On receiver side hit ratio is around 100% in both file transfer and checksum computation phase.

As UWEs pose a significant threat for file transfers, we evaluated sequential end-to-end integrity verification approach in terms of its receiver-side cache behaviour as given in Figure 2.4. Both the sender and the receiver servers are equipped with 16GB of RAM, thus we transferred a mixed dataset that contains 273 files with total of 274GB size. Only three files in the dataset are larger than the memory size, as a result page misses increase only when checksum thread attempts to read large files. Checksum computation for all small files triggers page hits as they are found in the page cache of the memory. The total number of page misses is around 145M that corresponds to 56GB, the total size of three large files. We also confirmed that other implementations of integrity verification exhibit similar behavior of reading small files from page cache. Hence, current integrity verification approaches for file transfers are susceptible to UWEs for files that are smaller than the memory size as most production systems use data transfer nodes with 64GB or larger memory size and

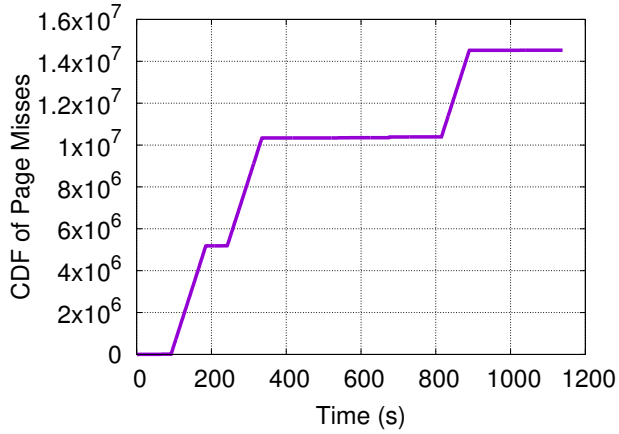


Figure 2.4: Transferring a mixed dataset with the state-of-the-art integrity verification algorithm reveals that only files that are larger than memory size contribute to page misses during the checksum computation.

average file sizes of scientific data transfers are in the order of megabytes [43]. Moreover, even if a file systems is robust against UWEs, silent data corruption can still affect file transfers when data is transmitted from memory of data transfer node to file system disks due to various reasons such as faulty cables and driver firmware bugs. Hence, despite the availability of integrity verification solutions, file transfers in scientific networks are still vulnerable to silent data corruption since existing implementations of integrity verification do not guarantee end-to-end coverage.

CHAPTER 3
LITERATURE REVIEW

In this chapter, we discuss related works on high-performance data transfers, end-to-end integrity verification, and data corruption in storage systems.

3.1 High-performance data transfers

Most of previous work on high-speed data transfers focus on scheduling [44], throughput optimization [45, 18, 46, 47], and power consumption optimization [48, 49, 50, 19, 51]. Globus [34] offers data transfer and sharing services and is well adopted by research community. Yun et al. proposed ProbData [47] to tune the number of parallel streams and buffer size for memory-to-memory TCP transfers using stochastic approximation. ProbData is able to explore the near-optimal configurations through sample transfers, but it takes several hours to converge. Rao et al. [52] presented stochastic gradient descent based solution to tune the number of parallel flows. HARP [46] models data transfers using historical data and real-time sampling, and uses this model to estimate the application layer transfer parameters that would maximize the throughput of given transfer task. Alan et al. [48] proposed energy-efficient data transfer algorithms to tune application layer transfer parameters, and find a balance between transfer throughout and energy consumption at the end hosts. They monitor CPU usage of end hosts and estimate energy consumption with the help of models that relate CPU usage to energy consumption. A cost function is used to determine the energy efficiency of each configuration based on the transfer throughput and energy consumption values. Finally, a configuration with minimum cost is identified and used for the rest of the transfer.

3.2 Integrity verification

Researchers studied integrity verification in the context of storage outsourcing [30, 53, 54], long term archiving [55, 32], file systems [56, 57, 58], databases [33], provenance [59], and data transfer [35]. Zhang et al. [58] evaluated Zetabyte 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.

Globus [34] supports end-to-end integrity verification for data transfers. It pipelines data transfers and checksum computation to minimize the overhead of integrity verification. However, its pipelining approach fails to work well when a dataset consist of mixed file sizes. Liu et al. propose block-level pipelining to improve pipelining of mixed size datasets by dividing large files into blocks [35]. It reduces execution time considerably especially when dataset is composed of files with mixed sizes, however it requires careful tuning of block size to perform well. In a previous work, we proposed Fast Integrity Verification Algorithm (FIVER) that reads files once and run the transfer and checksum computation processes simultaneously, reducing I/O overhead and checksum computation time [36]. FIVER outperformed state-of-the-art solutions by reducing the overhead of integrity verification from up-to 60% to less than 10%.

3.3 Data corruption in storage systems

Studies on disk fault analysis investigates drive failures [60, 61, 62], latent sector errors [63], and data corruption [26, 27, 58]. Shah et al. investigated the underlying reasons for disk failures and identified several factors including media errors include head hits bump, scratch in disk, high-fly writes, rotational vibration, hard particles, and head slap [61]. Schroeder et al. [62] analyzed data from 100,000 disks over a five- year pe-

riod and found that disk failures have positive correlation with disk ages. Hence, modern storage systems store checksum of file blocks next to the block in the disk to detect data corruption.

Checksum mismatch defines the discrepancy between the stored checksum and calculated checksum of a block. It can happen because of several reasons such as (i) a misdirected write in which the data is written to an incorrect disk location, thus overwriting and corrupting data, (ii) write error in which only a portion of the data block is written successfully, and (iii) data corruption caused by components within the data path [28, 64]. Bairavasundaram et al. monitored 1.53 million disk drives over 41 months and observed more than 400,000 checksum mismatches [26]. They also found that nearline disks have an order of magnitude higher probability of developing checksum mismatches than enterprise-class disks. Yet, corrupt enterprise-class disks tend to develop more checksum mismatches. Although data scrubbing and RAID reconstruction can detect and possibly recover checksum mismatches, they take a long time to finish during which data becomes inaccessible. In another work, Bairavasundaram et al. monitored 1.5 million HDDs over 32 months and found that 8.5% of all disks developed at least one latent sector error during observation period [63]. Moreover, Krioukov et al. showed that even tough silent data corruption is detected, the system may not recover the block, causing data to be lost permanently [27].

In literature there exist few studies analyzing rates for occurrence of UDEs. [40] evaluates the impact of UDEs in RAID systems using combination of the data presented in [63] and [26]. In [63], a study of latent sector errors was presented for the same set of field data that was used to derive the statistics on UDEs in [26]. From dataset [63] they first approximate the workload run on the field systems by back calculating from the rate of hard errors observed. Then using dataset [26] they compute rates for occurrence of UDEs. Study [26] classifies corrupted events by the way they manifest in the detection logic the system such as checksum, identity and parity mismatches. Thus, they assume that checksum mis-

matches are caused by far off-track writes and by hardware bit corruption, and identity, and parity mismatches are caused by dropped writes and near off-track writes. Using dataset [63, 26] they found estimate rates of UDEs in $\frac{UDEs}{I/O}$, for nearline systems estimate values for dropped I/O, near off-track I/O and far off-track I/O are 9×10^{-13}, 10^{-13}, and 10^{-12} respectively. The corresponding rates for enterprise systems are 9×10^{-14}, 10^{-14}, and 10^{-13}.

CHAPTER 4

SYSTEM DESIGN OF ROBUST INTEGRITY VERIFICATION ALGORITHM

Existing end to end integrity verification approaches are vulnerable to silent data corruptions, thus we present Robust Integrity Verification Algorithm (RIVA). RIVA extends the coverage of end-to-end integrity verification to detect errors that happen between memory and storage systems by reading the data from disk when computing the checksum as shown in Figure 4.1.

RIVA consists of three threads as shown in Figure 4.2. *Transfer* thread receives the files and writes them to the disk, *Cache Evictor* thread evicts file pages from the memory, and *Checksum* thread reads evicted pages back from the disk to calculate the checksum. *Cache Evictor* uses `mmap`, `munmap`, and `mincore` system calls to locate and evict file pages. When a file is recently received by the receiver and written to the disk, its pages are kept in the page cache. Thus, *Cache Evictor* uses `mmap` to find location of the file pages in the virtual address space. Then, it uses `munmap` syscall to evict pages that are in the memory. Although `munmap` removes file pages in one call in most systems, it cannot guarantee the removal of pages as OS can bring back some of the pages immediately. Hence,

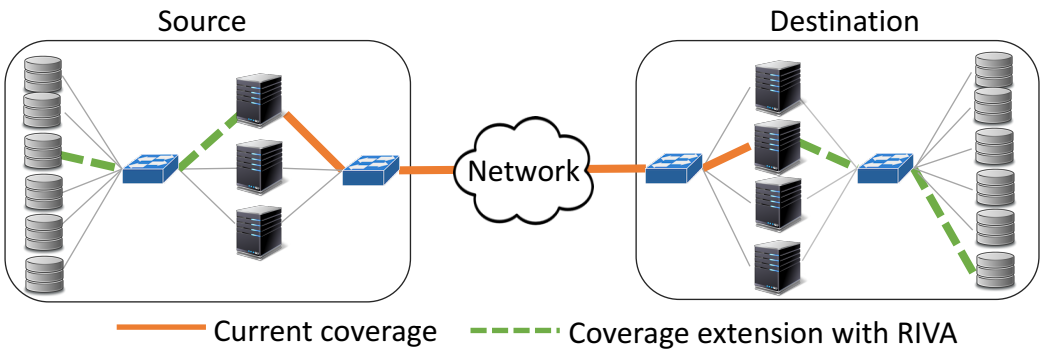


Figure 4.1: RIVA extends end-to-end integrity verification coverage to ensure data integrity between receiver memory and disk.

Cache Evictor checks whether or not the file is fully removed using `mincore`, which determines whether or not requested pages are resident in the memory. If file pages are not fully removed from the page cache, then *Cache Evictor* will keep calling `munmap` until `mincore` verifies that none of the chunk pages are resident in the memory. Then, *Cache Evictor* passes the evicted chunk to the *Checksum* to read them to calculate checksum and exchange with sender to verify.

Unlike sequential approach, all threads of RIVA work concurrently to minimize the total execution time. In Figure 4.2, *Transfer* thread is transferring page-n, *Cache Evictor* thread is evicting page-k from memory, and *Checksum* thread is reading page-3 to calculate checksum. *Transfer* thread sends periodic signals to *Cache Evictor* to inform about written pages so that those pages could be removed from the cache. Similarly, *Cache Evictor* thread sends messages to *Checksum* to notify it about evicted pages. Instead of performing this per-page basis, RIVA sends messages after a certain amount of data, called chunk. Chunk size is configurable but is set to 256MB by default. Thus, each chunk contains 65,536 pages as most OS kernels define page size as 4,096 bytes. Assume that page-3 is corrupted due to a UWE as shown in Figure 4.2. As opposed to sequential approach, RIVA is able to capture this error by removing page-3 from memory after it is saved to the disk. Thus,

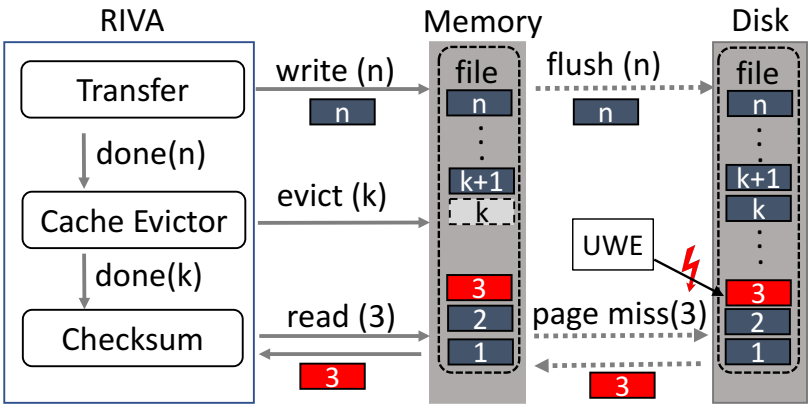


Figure 4.2: System architecture of RIVA. When *Checksum* thread attempts to read UWE-exposed page-3, it will trigger a page miss since *Cache Evictor* has evicted it. Consequently, page-3 will be read from the disk and the UWE will be captured.

when *Checksum* thread attempts to read page-3, it will trigger a page miss. Then, page-3 will be brought from the disk to the memory to be relayed to *Checksum* thread. Finally, the UWE will be detected and mitigated as integrity verification will result in a checksum mismatch and file retransmission. When checksum mismatch happens RIVA assumes that the sender copy of the file is correct one, so will discard the sender side copy and transfer file from sender to receiver again. While this may cause retransfers in cases where successful transfers are followed by incorrect checksum calculations due to undetected read errors while reading file from file system. This is default assumption by existing end to end integrity verification systems which we did not change. However, since we are calculating and comparing checksum of files in blocks (default is 256MB), only failed blocks will be resent, so the impact of mismatch is minimized.

Arslan et. al. [36] showed that verifying integrity of large files in small chunks reduces time to recover from failures. For example, when a 100GB file is transferred and checksum of source and destination copies do not match due to a single-bit failure, then we need to transfer whole file again as there is no way to determine which bit failed. On the other hand, we can verify the integrity of 100GB file at smaller scales such as 256MB chunks which obviates the need to transfer whole file due to single-bit failure because we can locate the chunk that the bit failure occurred and only transfer that block to recover. Thus, RIVA adopts block-level integrity verification and uses 256MB as default block size.

Performance overhead of RIVA relies on the speed of cache eviction and disk read operations. Cache eviction is a quite lightweight operation as it only updates virtual memory mappings of a file. In the worst case, cache eviction for a file chunk takes 0.5 seconds as multiple `munmap` calls are made to guarantee page eviction. Yet, RIVA runs *Cache Evictor*, and *Checksum threads*, cache eviction is unlikely to be bottleneck since checksum computation of a file block tend to take more time than running repeated system calls. On the other hand, disk read could impose high overhead if disk read speed is slower than disk

write speed. This is because file read speed becomes a limiting factor when the network transfer and disk write speeds are faster than the disk read speed. We observed this scenario in one of our test cases, Chameleon Cloud, where memory size is 128GB and disk read/write speed is around 800 Mb/s. Although disk write speed is also slow, the OSes can buffer write requests in memory for files that are smaller than available memory size. This leads the write speed of disks to appear higher for small files, whereas disk read speed cannot go beyond hardware limitations.

CHAPTER 5
EXPERIMENTAL RESULTS

5.1 Evaluation Results

We run the experiments using two types of dataset; uniform and mixed datasets. Uniform datasets consist of one or more files in same size and mixed dataset consists of mixture of small and large files. Experiments were run on four different networks: HPCLab, ESnet, Pronghorn, and Chameleon Cloud whose specifications are given in Table-5.1. In HPCLab, we have two sets of servers; workstations (HPClab-WS) and data transfer nodes (HPCLab-DTN). Workstations are connected with the 1G link whereas data transfer nodes are connected with the 40G link. Although data transfer nodes are located in the same local area network, we injected artificial delay between them to emulate wide-area network condition using traffic controller `tc` of Linux. Pronghorn is a campus cluster and its nodes are connected with 10G links. Chameleon Cloud is an academic cloud service provider and its nodes are connected with 10G links. ESnet consists of two nodes that are connected with a dedicated 100G link. Finally, we used one Pronghorn server as the sender and one Chameleon Cloud server as the receiver to run Pronghorn-Chameleon experiments. We repeated the experiments five times and present average results unless otherwise noted.

$$Overhead = 100 * \frac{t_{algorithm} - t_{FIVER}}{t_{FIVER}} \tag{5.1}$$

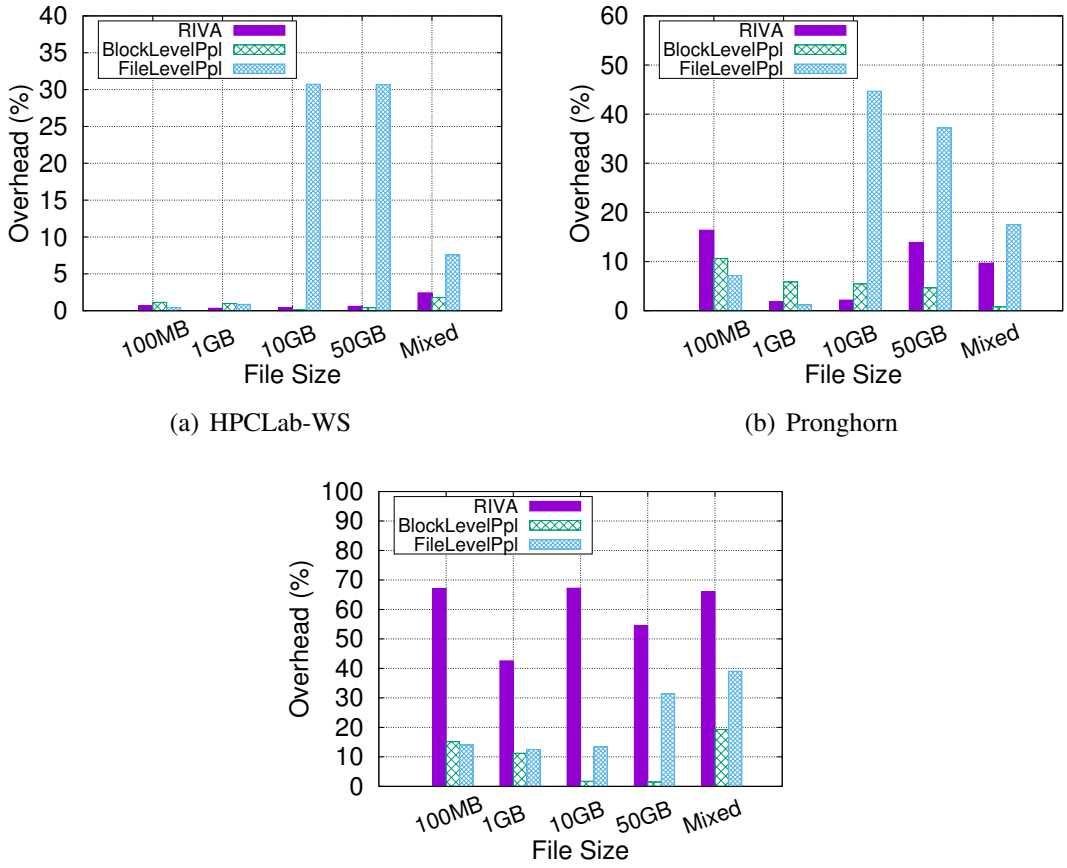
We compare RIVA against file-level pipelining (i.e., FileLevelPpl), block-level pipelining (i.e., BlockLevelPpl), and FIVER. FileLevelPpl overlaps the transfer of a file with the checksum of another file. BlockLevelPpl splits large files into smaller blocks and overlaps the transfer of a block with the checksum of another block. Finally, FIVER overlaps trans-

Specs	Storage	CPU	Memory (GB)	Bandwidth (Gbps)	RTT (ms)
HPCLab-WS	SATA HDD	8 x Intel Core i5-7600 @3.50GHz	16	1	0.2
Chameleon Cloud	SATA HDD	12 x Intel Xeon E5-2670 @2.30GHz	128	10	0.2
Pronghorn	GPFS	16 x Intel Xeon E5-2683 @2.10GHz	192	10	0.1
HPCLab-DTN	NVMe SSD	16 x Intel Xeon E5-2623 @2.60GHz	64	40	30
ESnet	RAID-0	12 x Intel Xeon E5-2643 @3.40GHZ	128	100	89

Table 5.1: System specification of networks.

fer of a file with the checksum of the same file to share I/O between the two. We omitted the performance results of sequential integrity verification approach as it performs similar to FileLevelPpl in page miss behavior and worse in execution time. Our results indicate that FIVER always yields the shortest execution time. Hence, we calculate the performance of the algorithms relative to FIVER and define overhead as shown in Equation 5.1. t_{FIVER} and $t_{algorithm}$ refer to the times it takes to transfer a dataset using FIVER and another algorithm, respectively. For example, if a transfer takes 120 seconds with FIVER, and 130 seconds with RIVA, then the overhead becomes $8.3\%(100 * \frac{130-120}{120})$. We also collected page miss values to compare disk access behavior of the algorithms. Page miss values define the number of blocks that the OS fetched from disk since it could not find them in page cache of main memory.

We conducted extensive experiments in six different networks which are grouped as local-area network (Figure 5.1) and wide-area network (Figure 5.2) results. The overhead of RIVA is always less than 5% in HPCLab-WS transfers (Figure 5.1(a)) which can be attributed to slow transfer speed. Since RIVA pipelines cache eviction and checksum computation operations with file transfers, RIVA incurs negligible overhead when transfer speed is the bottleneck. The overhead of FileLevelPpl reaches up-to 30% for 10GB



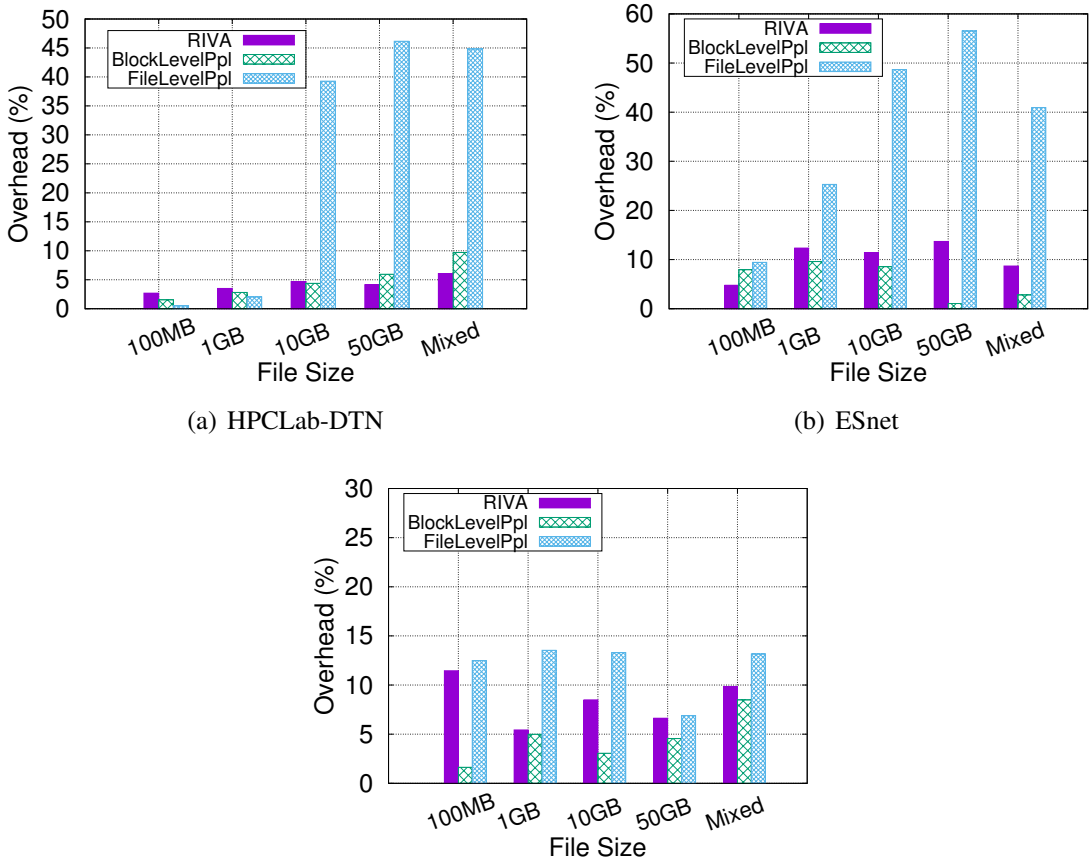
(a) HPCLab-WS

(b) Pronghorn

(c) Chameleon

Figure 5.1: Performance comparison of algorithms in LAN experiments. While RIVA is able to keep its overhead below 17% in HPCLab and Pronghorn networks, slow disk speed and large memory size in Chameleon Cloud deteriorates its performance significantly.

and 50GB files as there is only one file in these datasets, causing FileLevelPpl to perform similar to sequential approach and cannot take advantage of transfer and checksum pipelining. On the other hand, the overhead of RIVA increase up-to 17% at Pronghorn as its disk read speed is worse than disk write and transfer speeds. Unlike HPCLab-WS and Pronghorn networks, the overhead of RIVA exceeds that of FileLevelPpl in Chameleon Cloud as given in Figure 5.1(c). This mainly because Chameleon Cloud nodes are customized for high-performance computing with large memory size and multi-core CPUs, and exhibit poor disk performance. Disk read performance further degrades when it over-



(a) HPCLab-DTN

(b) ESnet

(c) Pronghorn-Chameleon

Figure 5.2: Performance comparison of algorithms in WAN experiments. RIVA is able to keep its overhead below 12% in all networks.

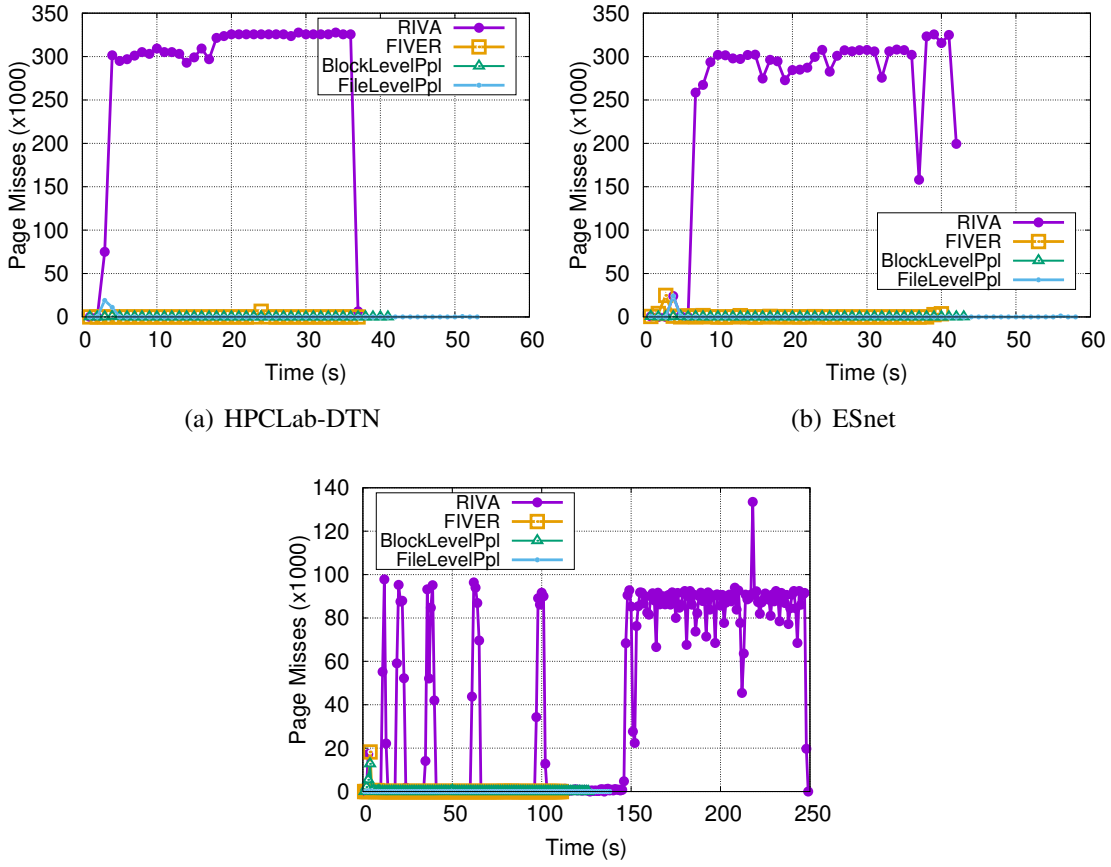
laps with disk write operations. While disk write speed is also low, the OS is able to cache file writes in the memory and flush them to the disk at a slower rate. Thus, BlockLevelPpl and FileLevelPpl can write fast and read back files from memory, allowing them to yield a small overhead. On the other hand, when we transferred a 150GB file, we noticed drastic performance degradation for FileLevelPpl, reaching over 70%, as the OS is unable cache file writes that are larger than the memory size.

RIVA keeps its overhead less than 12% in WAN experiments as shown in Figure 5.2. Its overhead is below 5% in HPCLab-DTN as overall speed is limited by checksum computation, thus reading data from the page cache or the disk has negligible impact on the overall

performance. Since the checksum computation is the bottleneck in ESnet, FileLevelPpl suffers significantly for the single file transfers (i.e., 10GB and 50GB) and takes up-to 50% time more than FIVER. Its performance is also 40% worse than FIVER for mixed dataset since it fails to benefit from overlapping of transfer and checksum processes when dataset contains both small and large files. BlockLevelPpl, however, achieves the lowest overhead in almost all cases since its pipelining approach overcomes suboptimal overlapping problem that FileLevelPpl experiences.

Overall in all networks except Chameleon, results show similar behavior; FileLevelPpl has the largest overhead and BlockLevelPpl has the lowest overhead. In Chameleon, RIVA has high overhead compared to other algorithms because of slow disk read-write speeds, which degradates even more when read and write overlaps. Thus, speed of RIVA strongly depends on disk read-write speed in exchange of increasing the robustness to silent data corruption.

We also collected the page misses (i.e. disk reads rates) during the transfer of a 10GB file in WAN as given in Figure 5.3. The transfer speed in HPCLab-DTN and ESnet testbeds is higher than checksum computation speed, letting *Checksum* thread to always have available data to process. Thus, RIVA sustains consistent disk I/O rate (around 2.4 Gbps) as shown in Figure 5.3(a) and 5.3(b). On the other hand, RIVA returns different behaviour in Chameleon Cloud where disk read speed is significantly worse than transfer and disk write speeds for files that are smaller than 100GB. Moreover, when the disk read and write operations (i.e., transfer of the file and checksum computation) overlap, the read performance degrades significantly. Consequently, RIVA has short periods of disk reads until transfer completely finishes. The transfer of the file completes at around 150s after which disk read performance recovers and RIVA's *Checksum* thread obtains high and consistent read I/O performance (around 800 Mbps). On the other hand, all the other approaches reads the file completely from page cache in all networks, leading zero page misses throughout the



(a) HPCLab-DTN

(b) ESnet

(c) Chameleon Cloud

Figure 5.3: RIVA yields high page miss rates as it evicts file pages from memory and reads directly from disk to capture UWEs.

checksum operation. We further calculated total page misses for RIVA and verified that it is close to 10GB. As page misses are calculated at application granularity, the measured pages miss values include misses caused by other operations such as network transfer and disk write. Thus, page miss values itself cannot guarantee reading all pages from the disk. Thus, we introduce *extreme-injection* test in the next section to confirm that none of the pages of a file is read from the page cache of main memory.

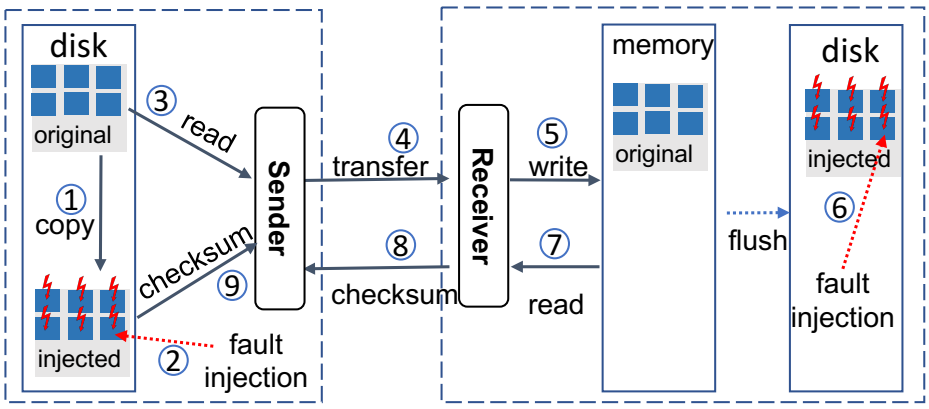


Figure 5.4: Extreme-injection test involves injecting UWEs to all pages of files to determine if a transfer application reads any data from page cache.

5.1.1 Fault Injection

Given that the rate of UWE occurrence is low, testing RIVA in a real system would be costly, probably requiring a prohibitively large number of disks to observe within a reasonable period of time. Hence, we reproduced UWEs by injecting faults to files pages on disk UWEs. We first identify the disk sectors that file pages reside. Then, we changed the content of page by writing directly to disk partition which is not reflected to cached copy of pages. For example, when we inject fault to the page $k + 1$ in Figure 4.2, the cached copy will not be updated, creating a similar impact of UWEs as shown in Figure 2.2.

We introduce *extreme-injection* where all pages of a file are exposed to UWE to check if an integrity verification algorithm reads any file pages from page cache during the checksum operation. We first create a copy of the file on sender side and change one bit of all of its pages (step 1 in in Figure 5.4). This copy is called *injected* and used to validate the output of the receiver side checksum. The sender sends the *original* file to the receiver which is written to disk (step 5). As file is being written to disk, we flip a bit of all file pages that are flushed to disk just before checksum computation starts. We flip same bits of pages at the sender and the receiver to be able to compare injected copies. Upon completion of fault injection to all file pages, we let checksum thread to read the file and compute its checksum

(step 7). The computed checksum value is then sent to the sender to be compared against the its *injected* file's checksum (step 8 and 9).

If checksum values match, we can then confidently claim that the checksum thread of the receiver must have read all file pages from the disk. If the receiver happens to read even one page of the file from the memory, its checksum value will be different than that of the sender's *injected* version of the file. We evaluated the *extreme-injection* scenario in HPCLab-WS using several files that are smaller and larger than the memory size, 16GB. Table 5.2 presents the results of integrity verification algorithms for the *extreme-injection* test. FIVER and BlockLevelPpl failed to pass the test for all file sizes as they always read from page cache. FileLevelPpl is able to catch all faults only in 50GB file which is three times larger than memory size. Surprisingly, FileLevelPpl fails to catch all faults for the 20GB file despite yielding high page misses, inferring the occurrence of small number of page hits. Finally, RIVA is able to capture all fault injections regardless of file size as a result of invalidating cache copies of file blocks before starting checksum computation. Interestingly, in ESnet testbed we are able to inject UWEs even though it has RAID-0 storage and this injection would have gone unnoticed without RIVA.

	1MB	**100MB**	**1GB**	**10GB**	**20GB**	**50GB**
FIVER	✗	✗	✗	✗	✗	✗
BlockLevelPpl	✗	✗	✗	✗	✗	✗
FileLevelPpl	✗	✗	✗	✗	✗	✓
RIVA	✓	✓	✓	✓	✓	✓

Table 5.2: The results of extreme-injection experiment in HPCLab-WS network with 16GB system memory. RIVA is able to detect all injected faults whereas FileLevelPpl can only detect it for a 50GB file.

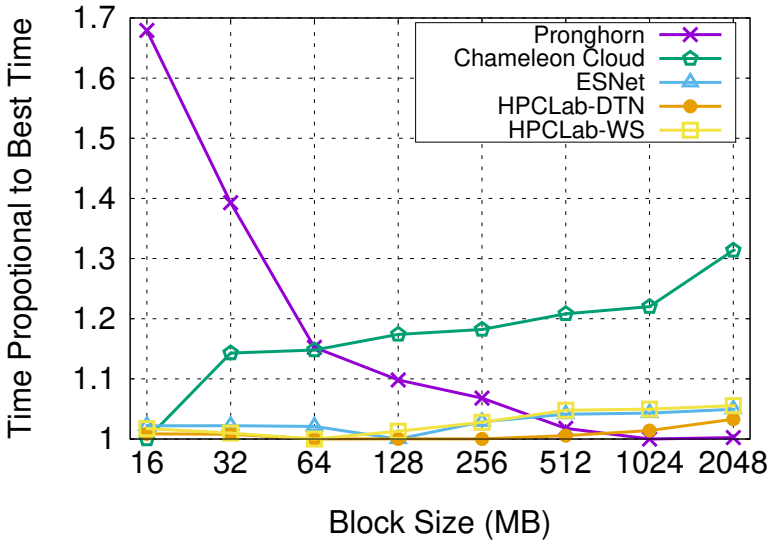


Figure 5.5: While block size has negligible impact in most networks, it affects execution time by around 70% in Pronghorn and up-to 30% in Chameleon Cloud.

5.1.2 Impact of Block Size on Performance

As discussed in Section 4, RIVA executes *Transfer*, *Cache Evictor*, and *Checksum* threads concurrently for different portion of the same file (aka block) to keep its overhead at minimum. By default, we define the block size to be 256 MB, however, this might not be an optimal value in all testbeds due to differences in storage subsystems. Thus, we evaluated the impact of block size on execution time by transferring a 30GB file with integrity verification as shown in Figure 5.5. It is clear that block size has negligible impact on the performance for HPCLab-DTN, HPCLab-WS, and ESNet networks. On the other hand, it leads to 30% and 70% change in execution time for Chameleon Cloud and Pronghorn testbeds, respectively. Interestingly, while small block size yields the best performance in Chameleon Cloud, it returns the worst performance in Pronghorn. As a result, the default value of 256MB yields up-to 20% and 8% lower performance compared to the best performing block size for Chameleon Cloud and Pronghorn networks.

We further analyzed receiver-side disk read and write I/O throughput for 16MB, 256MB,

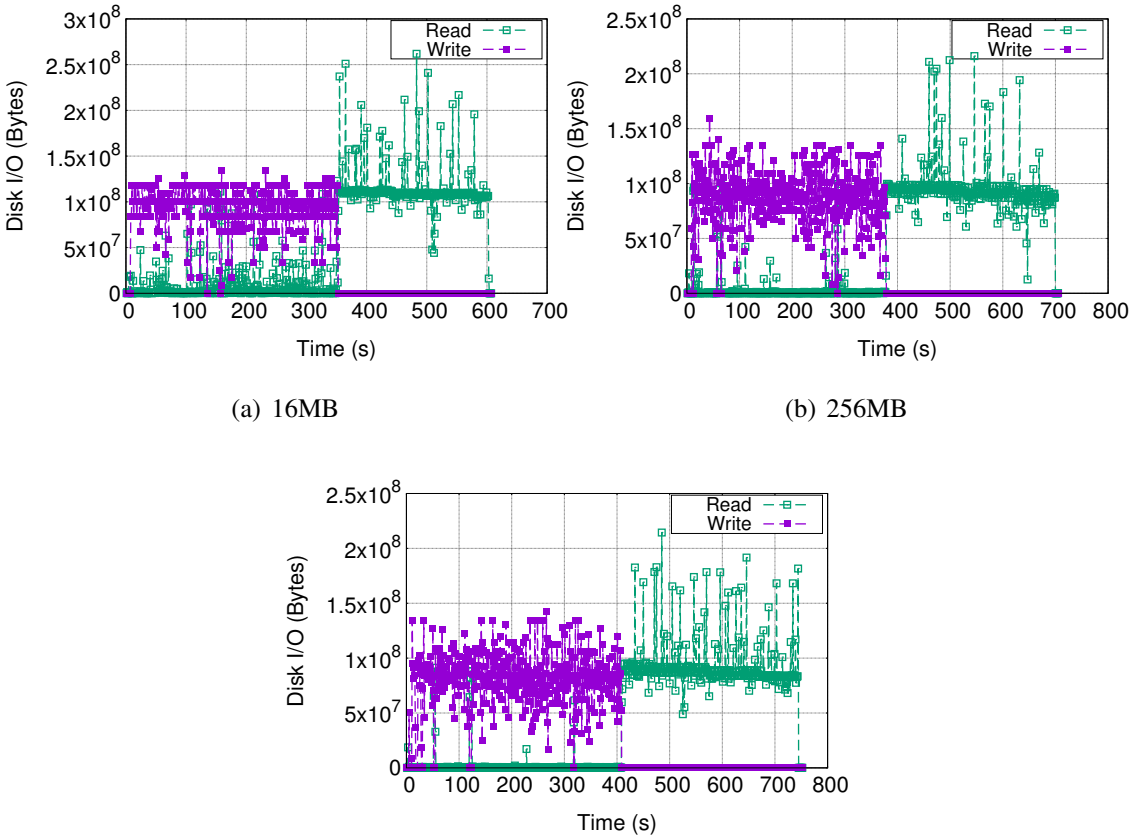
and 2048MB block sizes for Chameleon Cloud network in Figure 5.6. Receiver side write I/O is a result of writing the file to disk after receiving it from the network. Read I/O, on the other hand, is part of integrity verification process for which the file is read back from the disk to compute its checksum. Overlapping read and write I/O operations in Chameleon Cloud slows down both operations which is further exacerbated by increased block size. While file write operation finishes at around 350s for 16MB block size, it takes more than 400s for block size of 2048MB. Moreover, when block size is 16MB, considerable amount of disk read activity can still takes place, whereas it becomes rare when block size is 256MB and almost disappears when it is 2048MB. Consequently, remaining read operation (after write operation is complete,) takes around 250s (from 350s to 600s), 320s (from 380s to 700s), 340s (from 410s to 750s) for 16MB, 256MB, and 2048MB block sizes, respectively.

In summary, while block size can have significant impact on execution time, no single predefined value yields the close-to-optimal performance in all networks. Thus, a careful tuning is necessary to efficiently utilize underlying storage subsystem and shorten execution time. However, we leave such a runtime optimization as a future work.

5.1.3 Dynamic Checksum and Transfer Parallelism

By default, RIVA creates one transfer and checksum threads. We observed that Intel Xeon E5 processors with 3.4 GHz clock rate process around 300MB data per second for the checksum calculation when MD5 hash algorithm is used. The network and storage speeds, on the other hand, can be faster, causing checksum computation to be the bottleneck. Hence, we implemented dynamic parallelism algorithm to create a new checksum or transfer threads when it realizes that either one of them is slow.

We define $ratio$ to be the ratio of transfer throughput to checksum throughput as shown in line 9 in Algorithm 1. If the $ratio$ is higher than 1.1, (meaning transfer speed is at least 10% faster than checksum speed), it attempts to open a new checksum thread to



(a) 16MB

(b) 256MB

(c) 2048MB

Figure 5.6: Disk I/O throughput for different block size values for the transfer of a 30GB file in Chameleon Cloud network.

leverage spare cores in the system to keep up with transfer speed. We also define confidence interval to avoid transient variations in transfer and checksum throughput such that RIVA will open a new transfer or checksum threads only after it builds enough confidence ($THRESDHOLD$) on results. The threshold can be tuned for specific network conditions but we noticed that setting it to 5 is sufficient in our test environments. As $monitor$ function is called once a second, this would set the confidence threshold to five seconds. If the $ratio$ is smaller than 1.1 (line 17), RIVA deduces that transfer throughput could be limiting factor and creates a new transfer thread after building enough confidence on the assumption. Increasing the number of transfer threads could also improve checksum throughput

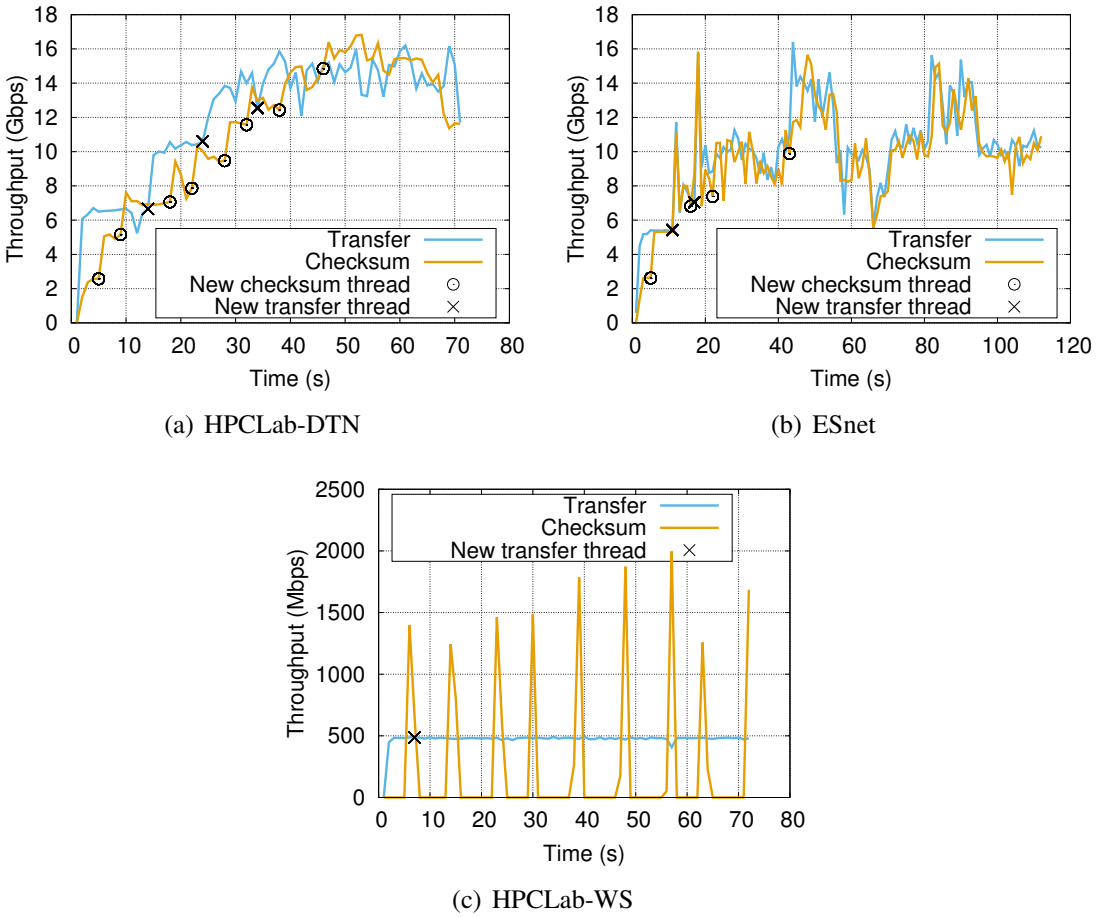
Algorithm 1: Dynamic parallelism of RIVA

```
1  global variables
2  tr-confidence= 0, ch-confidence= 0, prevTransferThr= 0,
     prevChecksumThr= 0, stopSearch= False
3  end global variables
4  function monitor(transferThr, checksumThr)
       /* If opening new transfer/checksum thread does not
          help, stop adding more                              */
5      if prevChecksumThr ≥ checksumThr or prevTransferThr ≥
        transferThr then
6          stopSearch== True
7      if stopSearch == True then
8          return
9      ratio = transferThr / checksumThr
       // Slow checksum case
10     if ratio ≥ 1.1 then
11         if ch-confidence ==THRESDHOLD then
12             prevChecksumThr = checksumThr
               OPENNEWCHECKSUMTHREAD( )
13             ch-confidence= 0
14         else
15             ch-confidence++
16         tr-confidence= 0
       // Slow transfer case
17     else
18         if tr-confidence == THRESDHOLD then
19             prevTransferThr = transferThr
20             OPENNEWTRANSFERTHREAD( )
21             tr-confidence== 0
22         else
23             tr-confidence++
24         ch-confidence= 0
```

since checksum throughput might be limited by the transfer performance. If adding a new checksum or transfer thread does not improve throughput, RIVA assumes that it reached to maximum performance and stops adding more threads since doing so will only overload system resources (line 5).

We evaluated the dynamic parallelism algorithm in HPCLab-DTN, ESnet, and HPCLab-

WS networks as illustrated in Figure 5.7. Single-threaded checksum computation can only reach to 2.4 Gbps speed and becomes bottleneck in HPCLab-DTN and ESnet networks. Dynamic parallelism is able to detect that checksum operation is the bottleneck and creates new checksum threads at around 5s as shown in Figure 5.7(a) and 5.7(b). Creating more threads eventually helps checksum operation to keep up with transfer speed which triggers new transfer thread creation at time around 15s in HPLab-DTN transfer. It keeps adding more checksum and transfer threads until no further improvement is observed, which happens at around 50s for HPCLab-DTN and 45s in ESnet. The dynamic parallelism algorithm increases RIVA's overall throughput from 2.4 Gbps to around 16 Gbps and 12 Gbps for



(a) HPCLab-DTN

(b) ESnet

(c) HPCLab-WS

Figure 5.7: RIVA can dynamically adjust the number of transfer and checksum threads to overcome bottlenecks.

HPCLab-DTN and ESnet, leading over 5 times 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 single thread is sufficient to achieve maximum possible performance. For example, RIVA's performance in HPCLab-WS is limited to 550 Mbps due to slow network speed. Unaware of this limitation, dynamic parallelism opens up new transfer thread at around 5s as shown in Figure 5.7(c). After realizing that this change does not improve transfer performance, dynamic parallelism terminates its search phase immediately.

5.1.4 Statistical Analysis on Likelihood of UDEs with RIVA

In overall, there are four I/O operations that take place during the transfer of a file with integrity verification. They are (i) file read on the source server to transfer file content (*Src(R)* in Table 5.3), (ii) file write on the destination server to save the transferred data to disk (*Dst(W)*), (iii) second file read on the source to compute the checksum of the transferred file (*Src(R)*), and (iv) file read on the destination to compute the checksum of the received file (*Dst(R)*). Existing integrity verification algorithms read all or some portion of files from memory cache during checksum operations, making them unprotected against UDEs that might happen during file transfer related I/O activities. On the other hand, RIVA ensures that file reads for checksum calculations avoid cache memory and read the file from disk to detect UDEs. Yet, there are still cases where RIVA can fall short to detect UDEs when multiple I/O operations are exposed to similar UDEs. For instance, if a UWE on destination server is followed by a URE on destination (while reading the file from disk to compute the checksum) exhibits in a way that it is exactly reverse of the UWE, then RIVA will miss UWE and corrupted data will be to be stored in the disk.

Table 5.3 lists few examples of single or multi-bit errors affecting one or more I/O operations during the transfer of a two-bit data whose original value on the source is *"00"*.

Row ID	# of I/Os Impacted	Transfer		Checksum		Detected by RIVA?
		Src (R)	Dst (W)	Src (R)	Dst (R)	
1	1	0**1**	01	00	01	Yes
2	1	00	**11**	00	11	Yes
3	1	00	00	0**1**	00	Yes
4	2	0**1**	01	00	0**1**	Yes
5	2	0**1**	01	0**1**	01	No
6	2	00	**11**	00	**00**	No
7	3	**1**0	**0**0	0**1**	00	Yes
8	3	0**1**	01	**1**0	**10**	No
9	3	00	**1**0	**11**	1**1**	No
10	4	0**1**	**10**	1**1**	**01**	Yes
11	4	**11**	**1**0	**11**	1**1**	No
12	4	0**1**	0**0**	**1**0	**1**0	No

Table 5.3: Examples of UDEs that affect I/O during the transfer of a two bit data originally stored in the disk of the sender as *"00"*. Out of four I/O operations (two for transfer and two for integrity verification), RIVA is able to capture all single and multi bit errors which that affects only one I/O operation. However, RIVA may miss UDEs if they occur in multiple I/O operations in a way that sender and receiver checksum return same value.

Note that when any one I/O operation (i.e. the number of I/Os impacted is 1) is exposed to single or multi-bit errors, RIVA is able to detect since it is guaranteed at least one of the checksum reads will return the correct value. On the other hand, when multiple I/O operations are affected by UDEs, then RIVA may miss them if they happen in a way that source and destination checksum values match. For example, consider a case where a UDE happens while file is read on source server to initiate the transfer, causing data to be read as *01* (row 5 on Table 5.3). The receiver then writes the corrupted data to disk as it receives, *01*. Once the transfer operation is complete, both source and destination servers read the data from disk again to compute checksum. Assume that another UDE takes place when reading the file on source server and causes data to be read as *01*, again. Then, since destination server saved the data as *01* as well and checksum I/O on destination is not exposed to a UDE, it will cause checksum values of source and destination servers to match, preventing RIVA to detect silent data corruptions. In the rest of this section, we will estimate the probability of UDEs that RIVA is unable to capture.

Undetected bit error rate (UBER) is defined as the rate of errors that have escaped from *ECC* [57]. The probability of an undetected bit flip will be equal to UBER assuming each bit error in a data block is independent and the number of bit errors follows a binomial distribution. Equation 5.2 shows the probability of i bit flips in a b-bit block with assumption that there exist at most one flip for each bit, where E is uncorrectable bit error rate (UBER) [57, 65]. Equation 5.3 defines the probability of undetected bit error in a block, which is the sum of the probabilities of all possible combinations of bit flips (from total of 1 bit flip to b bit flips).

$$P_c(block, i) = \binom{b}{i} \times E^i(1-E)^{b-i} \tag{5.2}$$

$$\begin{aligned} P_c(block) &= \sum_{i=1}^{b} \binom{b}{i} \times E^i(1-E)^{b-i} \\ &= \sum_{i=0}^{b} \binom{b}{i} \times E^i(1-E)^{b-i} - (1-E)^b \\ &= (E + (1-E))^b - (1-E)^b \text{ (Binomial Theorem[1])} \\ &= 1 - (1-E)^b \\ &\approx bE \end{aligned} \tag{5.3}$$

RIVA will miss UDEs if they appear in more than one I/O operation as described above. Hence, two or more I/O operations must be exposed to UDEs out of total four disk read/write operations. Among those, we first explore the possibility UDEs that leads to data corruption in two I/O operations. Equation 5.2 shows the probability of i uncorrectable bit flips in an operation with block size of b. Since UDEs in two I/O operations have to happen in same or reverse order (see rows #5 and #6) for RIVA to miss, the probability becomes the multiplication of i bit UDE combinations and the probability of an

UDE to happen twice, as shown in Equation 5.4. Then, Equation 5.5 shows the cumulative probability of all bit errors to appear in two I/O operations.

$$P(block, i) = \binom{b}{i} \times (E^i \times (1-E)^{b-i})^2 \tag{5.4}$$

$$\begin{aligned}
P(block) &= \sum_{i=1}^{b} \binom{b}{i} \times (E^i \times (1-E)^{b-i})^2 \qquad (5.5)\\
&= \sum_{i=0}^{b} \binom{b}{i} \times E^i(1-E)^{b-i} - (1-E)^b \\
&= -(1-E)^{2b} + \sum_{i=0}^{b} \binom{b}{i} E^{2i}(1-E)^{2(b-i)} \\
&= -(1-E)^{2b} + (E^2 + (1-E)^2)^b \text{ (Binomial Theorem)} \\
&= (E^2 + (1-E)^2)^b - (1-E)^{2b} \\
&= (1 + 2E^2 - 2E)^b - (1-E)^{2b} \text{ (expanding both terms)} \\
&= (\binom{b}{0}1^b(2E^2-2E)^0 + \binom{b}{1}1^{b-1}(2E^2-2E)^1 \\
&+ \cdots + \binom{b}{b}1^0(2E^2-2E)^b) - (\binom{2b}{0}1^{2b}(-E)^0 \\
&+ \binom{2b}{1}1^{2b-1}(-E)^1 + \cdots + \binom{2b}{2b}1^0(-E)^{2b}) \\
&\quad E^b, E^{b-1}, \cdots, E^4, E^3 << E^2 \ \ (\forall E < 10^{-10}), \text{ thus} \\
&\text{ignoring terms with power of E is greater than 2} \\
&\approx (\binom{b}{0}1^b(2E^2-2E)^0 + \binom{b}{1}1^{b-1}(2E^2-2E)^1) \\
&- (\binom{2b}{0}1^{2b}(-E)^0 + \binom{2b}{1}1^{2b-1}(-E)^1) \\
&= (1 + b(2E^2-2E)) - (1-2bE)
\end{aligned}$$

$$= 1 + 2bE^2 - 2bE - 1 + 2bE$$

$$= 2bE^2$$

Since there are six possible combinations for two of four I/O operations to be affected by UDEs, the maximum probability of RIVA to miss UDEs becomes $6X2bE^2 = 12bE^2$. Please note that this is an upperbound as it counts all possible UDEs that happen twice whereas many of such error will be captured by RIVAas shown in row 4 in Table 5.3. In addition, the likelihood of three or four I/O operations to be exposed to UDEs in a way that it will mislead RIVA is much lower than that of two I/Os to be impacted as $E^2 >> E^3 >> E^4$ when E is lower than 10^{-10}. Thus, we can conclude that while the probability of UDEs to go undetected is bE with existing integrity verification algorithms, RIVAreduces it to bE^2. To given an example on real-world scenarios, consider a case where $E = 10^{-10}$ and $b = 256$MB, then $P(error) = bE = 0.2$ for traditional integrity verification algorithms whereas it is $\approx 10^{-11}$. Even for more conservative E value of 10^{-15}, RIVAreduces the probability of error from 10^{-6} to 10^{-21}. As a result, although RIVA cannot completely rule out undetected disk errors, it significantly reduces their likelihood for file transfers compared to existing end-to-end integrity verification algorithms.

CHAPTER 6
CONCLUSION AND FUTURE WORK

End-to-end integrity verification is vital for many applications which cannot tolerate silent data corruptions. However, its current implementations for file transfers fail to capture undetected disk write errors, creating possibility of permanent data loss. In this work, we propose RIVA to improve robustness of the end-to-end integrity verification by enforcing checksum calculation to read files directly from disk. RIVA invalidates the cached copies of file pages such that checksum calculation can read disk copies and detect silent data corruptions. Our extensive experiments show that RIVA offers a robust solution to capture and recover from all undetected disk write errors. In exchange, RIVA increases transfer execution time, however it can keep the overhead below 15% in most cases by concurrently executing transfer, cache eviction, and checksum operations. We proposed extreme fault injection model which injects fault to each page of the transferred data. Evaluations showed that RIVA is able to catch injected faults whereas file-level pipelining detects injected faults on the data larger than the size of the memory.

We further augmented RIVA with dynamic parallelism to identify and mitigate performance bottlenecks. Dynamic parallelism periodically measures performance of transfer and checksum threads to determine the slow component. Then, it creates new transfer or checksum threads to increase network throughput and the speed of checksum computation. We observed that dynamic parallelism can increase RIVA's performance more than 5 times by means of of increasing parallelism in the network and end servers.

REFERENCES

[1] "A measurement study on overhead distribution of value-added internet services," *Computer Networks*, vol. 51, no. 14, pp. 4153 –4173, 2007.

[2] M. H. Gunes, "Complex network discovery: Router-level internet topology mapping," AAI3323614, PhD thesis, Richardson, TX, USA, 2008, ISBN: 978-0-549-75832-7.

[3] M. A. Canbaz, J. Thom, and M. H. Gunes, "Comparative analysis of internet topology data sets," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017, pp. 635–640.

[4] A. Dittrich, M. H. Gunes, and S. Dascalu, "Network analysis of software repositories: Identifying subject matter experts," in *Complex Networks*, R. Menezes, A. Evsukoff, and M. C. González, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 187–198, ISBN: 978-3-642-30287-9.

[5] M. K. Popuri and M. H. Gunes, "Empirical analysis of crypto currencies," in *Complex Networks VII: Proceedings of the 7th Workshop on Complex Networks CompleNet 2016*, H. Cherifi, B. Gonçalves, R. Menezes, and R. Sinatra, Eds. Cham: Springer International Publishing, 2016, pp. 281–292, ISBN: 978-3-319-30569-1.

[6] B. Charyyev and M. H. Gunes, "Complex network of united states migration," *Computational Social Networks*, vol. 6, no. 1, p. 1, 2019.

[7] T. Goldade, B. Charyyev, and M. H. Gunes, "Network analysis of migration patterns in the united states," in *Complex Networks & Their Applications VI*, C. Cherifi, H. Cherifi, M. Karsai, and M. Musolesi, Eds., Cham: Springer International Publishing, 2018, pp. 770–783, ISBN: 978-3-319-72150-7.

[8] A. Breland, K. Schlauch, M. Gunes, and F. C. Harris Jr., "Fast graph approaches to measure influenza transmission across geographically distributed host types," in *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*, ser. BCB '10, Niagara Falls, New York: ACM, 2010, pp. 594–601, ISBN: 978-1-4503-0438-2.

[9] K. Komurov, M. H. Gunes, and M. A. White, "Fine-scale dissection of functional protein network organization by statistical network analysis," *PLOS ONE*, vol. 4, no. 6, pp. 1–9, Jun. 2009.

[10] M. Solmaz, A. Lane, B. Gonen, O. Akmamedova, M. H. Gunes, and K. Komurov, "Graphical data mining of cancer mechanisms with SEMA," *Bioinformatics*, May 2019.

[11] R. J. T. Klein, R. J. Nicholls, and F. Thomalla, "Resilience to natural hazards: How useful is this concept?" *Global Environmental Change Part B: Environmental Hazards*, vol. 5, no. 1-2, pp. 35 –45, 2003.

[12] J. Kiehl, J. J. Hack, G. B. Bonan, B. A. Boville, D. L. Williamson, and P. J. Rasch, "The national center for atmospheric research community climate model: CCM3," *Journal of Climate*, vol. 11:6, pp. 1131–1149, 1998.

[13] A. Dittrich, S. M. Dascalu, and M. H. Gunes, "Atmos - a data collection and presentation toolkit for the nevada climate change portal," in *ICSOFT*, 2012.

[14] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 3, no. 215, pp. 403–410, 1990.

[15] CMS, *The US Compact Muon Solenoid Project*, http://uscms.fnal.gov/.

[16] *A Toroidal LHC ApparatuS Project (ATLAS)*, http://atlas.web.cern.ch/.

[17] E. Arslan, B. Ross, and T. Kosar, "Dynamic Protocol Tuning Algorithms for High Performance Data Transfers," in *Proceedings of Euro-Par'13*, ser. Euro-Par'13, Aachen, Germany: Springer-Verlag, 2013, pp. 725–736, ISBN: 978-3-642-40046-9.

[18] E. Arslan and T. Kosar, "High Speed Transfer Optimization Based on Historical Analysis and Real-Time Tuning," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[19] E. Yildirim, E. Arslan, J. Kim, and T. Kosar, "Application-level optimization of big data transfers through pipelining, parallelism and concurrency," *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 63–75, 2016.

[20] *Dark Energy Survey*, https://www.darkenergysurvey.org/.

[21] 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*, vol. 42, pp. 49–65, 2016.

[22] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," in *ACM SIGCOMM computer communication review*, ACM, vol. 30, 2000, pp. 309–319.

[23] M. Gunes, M. A. Thornton, F. Kocan, and S. A. Szygenda, "A survey and comparison of digital logic simulators," in *48th Midwest Symposium on Circuits and Systems, 2005.*, 2005, 744–749 Vol. 1.

[24] F. Kocan and M. H. Gunes, "On the zbdd-based nonenumerative path delay fault coverage calculation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 1137–1143, 2005.

[25] J. L. Hafner, V. Deenadhayalan, W. Belluomini, and K. Rao, "Undetected disk errors in raid arrays," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 413–425, 2008.

[26] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 8, 2008.

[27] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Parity lost and parity regained.," in *FAST*, vol. 2008, 2008, p. 127.

[28] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Transactions on dependable and secure computing*, vol. 1, no. 1, pp. 87–96, 2004.

[29] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google file system*, 5. ACM, 2003, vol. 37.

[30] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and 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.

[31] J. Naruchitparames and M. H. Gne, "Enhancing data privacy and integrity in the cloud," in *2011 International Conference on High Performance Computing Simulation*, 2011, pp. 427–434.

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

[33] M. U. Arshad, A. Kundu, E. Bertino, A. Ghafoor, and C. Kundu, "Efficient and scalable integrity verification of data and query results for graph databases," *IEEE*

Transactions on Knowledge and Data Engineering, vol. 30, no. 5, pp. 866–879, 2018.

[34] *Globus*, https://www.globus.org/.

[35] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, and 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.

[36] E. Arslan and 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.

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

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

[39] A. Alhussen, B. Charyyev, and E. Arslan, "Is end-to-end integrity verification really end-to-end?" In *International Workshop On Parallel Data Storage & Data Intensive Scalable Computing Systems*, IEEE, 2018.

[40] E. W. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. Rao, and P. Zhou, "Evaluating the impact of undetected disk errors in raid systems," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, IEEE, 2009, pp. 83–92.

[41] M. Li and P. P. C. Lee, "Toward i/o-efficient protection against silent data corruptions in raid arrays," *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, 2014.

[42] R. Kettimuthu, Z. Liu, D. Wheeler, I. Foster, K. Heitmann, and F. Cappello, "Transferring a Petabyte in a Day," *Future Generation Computer Systems*, 2018.

[43] Z. Liu, R. Kettimuthu, I. Foster, and 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.

[44] T. Kosar, E. Arslan, B. Ross, and 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.

[45] E. Arslan, B. A. Pehlivan, and T. Kosar, "Big data transfer optimization through adaptive parameter tuning," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 89–100, 2018.

[46] E. Arslan, K. Guner, and T. Kosar, "HARP: Predictive Transfer Optimization Based on Historical Analysis and Real-Time Probing," in *Proceedings of SC'16*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, 25:1–25:12, ISBN: 978-1-4673-8815-3.

[47] D. Yun, C. Q. Wu, N. S. Rao, Q. Liu, R. Kettimuthu, and E.-S. Jung, "Data transfer advisor with transport profiling optimization," in *Local Computer Networks (LCN), 2017 IEEE 42nd Conference on*, IEEE, 2017, pp. 269–277.

[48] I. Alan, E. Arslan, and T. Kosar, "Energy-aware data transfer algorithms," in *Proceedings of SC'15*, ser. SC '15, Austin, Texas: ACM, 2015, 44:1–44:12, ISBN: 978-1-4503-3723-6.

[49] I Alan, E Arslan, and T Kosar, "Energy-performance trade-offs in data transfer tuning at the end-systems," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 4, pp. 318–329, 2014.

[50] I. Marincic and I. Foster, "Energy-efficient data transfer: Bits vs. atoms," in *2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, IEEE, 2016, pp. 1–6.

[51] R. Kettimuthu, G. Vardoyan, G. Agrawal, and P. Sadayappan, "Modeling and optimizing large-scale wide-area data transfers," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, 2014, pp. 196–205.

[52] N. S. Rao, Q. Liu, S. Sen, G. Hinkel, N. Imam, I. Foster, R. Kettimuthu, B. W. Settlemyer, C. Q. Wu, and D. Yun, "Experimental analysis of file transfer rates over wide-area dedicated connections," in *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, IEEE, 2016, pp. 198–205.

[53] Y. Zhu, H. Hu, G.-J. Ahn, and M. Yu, "Cooperative provable data possession for integrity verification in multicloud storage," *IEEE transactions on parallel and distributed systems*, vol. 23, no. 12, pp. 2231–2244, 2012.

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

[55] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. Rosenthal, and M. Baker, "The LOCKSS peer-to-peer digital preservation system," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 1, pp. 2–50, 2005.

[56] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. K. Mckusick, "Ffsck: The fast file-system checker," *ACM Transactions on Storage (TOS)*, vol. 10, no. 1, p. 2, 2014.

[57] Y. Zhang, D. S. Myers, A. C. Arpaci-Dusseau, and 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.

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

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

[60] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population.," in *FAST*, vol. 7, 2007, pp. 17–23.

[61] S. Shah and J. G. Elerath, "Reliability analysis of disk drive failure mechanisms," in *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*, IEEE, 2005, pp. 226–231.

[62] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?" In *FAST*, vol. 7, 2007, pp. 1–16.

[63] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *ACM SIGMETRICS Performance Evaluation Review*, ACM, vol. 35, 2007, pp. 289–300.

[64] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, *IRON file systems*, 5. ACM, 2005, vol. 39.

[65] S. O.K. J. Amy Tai Andrew Kryczka and A. C. Michael J. Freedman, "Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy," in *2019 USENIX Annual Technical Conference*, IEEE, 2019.