2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

Towards Securing Data Transfers Against Silent Data Corruption

Batyr Charyyev
Computer Science and Engineering
University of Nevada, Reno
bcharyyev@nevada.unr.edu

Ahmed Alhussen
Computer Science and Engineering
University of Nevada, Reno
aalhussen@nevada.unr.edu

Hemanta Sapkota
Computer Science and Engineering
University of Nevada, Reno
hsapkota@nevada.unr.edu

Eric Pouyoul
Energy Sciences Network
Lawrence Berkeley National Lab
lomax@es.net

Mehmet H. Gunes
Computer Science and Engineering
University of Nevada, Reno
mgunes@unr.edu

Engin Arslan
Computer Science and Engineering
University of Nevada, Reno
earslan@unr.edu

***Abstract*—Scientific applications generate large volumes of data that often needs to be moved between geographically distributed sites for collaboration or backup 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. It 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. In this paper, we investigate the robustness of existing end-to-end integrity verification approaches against silent data corruption and propose a Robust Integrity Verification Algorithm (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 increasing the robustness to silent data corruption. We also implemented dynamic transfer and checksum parallelism to overcome performance bottlenecks and observed more than 5x increase in RIVA's speed.**

I. Introduction

Large scientific experiments such as environmental and coastal hazard prediction [1], climate modeling [2], genome mapping [3], and high-energy physics simulations [4], [5] generate data volumes reaching petabytes per year. This massive amount of data often needs to be moved for various purposes including processing, collaboration, and archival. While most of earlier works focused on optimization of data transfers [6]–[8], the integrity of transfers are also critical for many applications such as Dark Energy Survey [9] and Sky Survey Simulation [10] as they rely on correctness of data to operate.

As data transfer rates are rapidly increasing, legacy integrity verification mechanisms fall short to detect corruption. Although some of data transfer components have built-in integrity verification mechanisms, they are either weak or applicable to a subset of available systems. For example, TCP uses 16-bit checksum to capture data corruption but it fails to detect errors once in 16 million to 10 billion packets [11], which is not rare for big scientific data transfers.

In addition to network, data corruption can also happen at 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) [12]–[14]. UDEs occur mainly due to firmware or hardware malfunctions in disk drives, and silently corrupt data without being detected by the disk. 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 [13]. 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 [12].

Li et al. estimate that the probability of an UWE is between 10^{-12} and 10^{-14}, once in every 1-100 terabytes [15]. 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 [12]–[14]. 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 [16]. 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 [17].

Application-layer end-to-end integrity check is proved to

978-1-7281-0912-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CCGRID.2019.00040

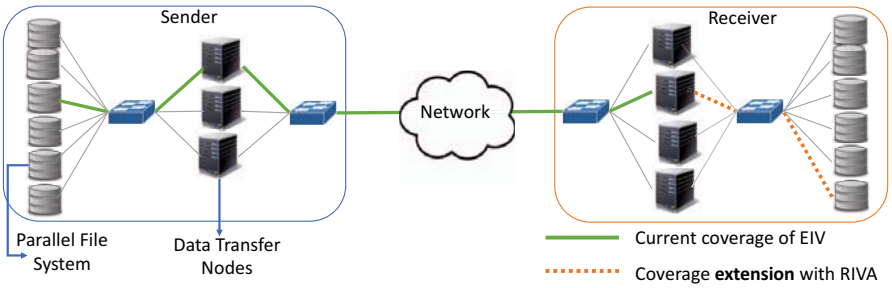


Fig. 1: RIVA extends end-to-end integrity verification coverage to ensure data integrity between receiver memory and disk.

be a robust solution to detect and recover from UDEs as it covers complete path of operations [18]–[22]. 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.

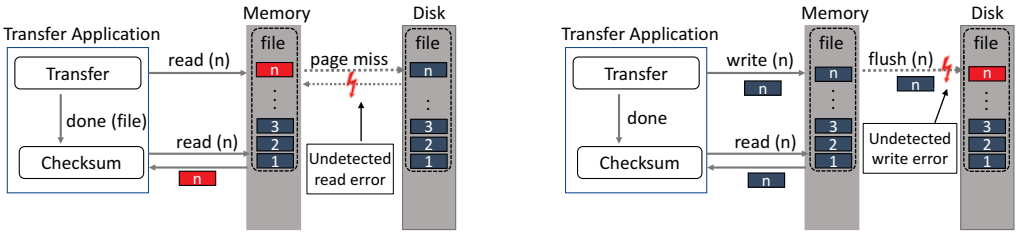
Several approaches are proposed to optimize the execution time of transfers when end-to-end integrity verification is enabled including file-level pipelining [23], block-level pipelining [24], and FIVER [25]. However, we find that existing implementations of end-to-end integrity verification for data 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 paper, 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. It is important to note that by invalidating cached copy of files before end-to-end integrity verification, *RIVA extends the coverage of end-to-end integrity verification mechanism without losing existing error-detection capabilities (e.g., capturing network errors)* as showed in Figure 1. We conducted extensive experiments using different network, storage subsystem, and dataset configurations and observed that unlike state-of-the-art solutions, RIVA always captures faults and offers robust end-to-end integrity verification of file transfers. On the other hand, invalidating memory copy of the files and forcing the OS kernel to read data from the disk increases the execution time of transfers. RIVA minimizes its overhead by overlapping transfer and checksum operations for different blocks of files and offloading memory unmapping (i.e., cache eviction) to a separate thread.

Contributions of this paper are as follows:

- We introduce RIVA to enhance resilience of end-to-end integrity verification for data transfers against UWEs by enforcing checksum calculations to read files directly from disk. Although RIVA can also capture errors that may happen while transmitting data from memory to disk, this work, without loss of functionality, only evaluates its capability in detecting silent disk errors.
- We introduce a fault injection approach to reproduce undetected write errors and evaluate state-of-the-art integrity verification approaches in terms of the reliability against data corruption.
- We introduce dynamic parallelism to identify and mitigate performance bottlenecks in integrity verification enabled file transfers.
- We conduct extensive experiments using variety of network, dataset, and fault injection scenarios to evaluate robustness and performance of RIVA.

The rest of paper is organized as follows: Section II describes integrity verification of data transfer and silent corruption scenarios. Section III presents related work and Section IV details design principles of the proposed algorithm. Section V discusses experimental results and Section VI concludes the paper with the summary.



(a) Undetected Read Error (URE)

(b) Undetected Write Error (UWE)

Fig. 2: Undetected disk errors could happen during read (a) and write (b) operations. When undetected read errors happen at sender or receiver, it will cause checksum mismatch and re-transfer of the file. On the other hand, when undetected write errors happen for small files, it will be missed since checksum will be computed based on memory copy of pages.

II. Background and Motivation

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

A. Undetected Disk Errors

There are two classes of undetected disk errors; undetected read error (URE) and undetected write error (UWE) as shown in Figure 2. UREs causes applications to see a different version of data than the one stored on the disk. In Figure 2(a), 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 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, UWEs could easily pass integrity verification 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(b) 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. Then, when a client accesses the file after some time, correct file copies wont 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 the disk errors [12], [13], [15] explain that ECC and S.M.A.R.T are not guaranteed to capture all silent errors.

B. 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.

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 3. 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 average file sizes of scientific data transfers are in the order of megabytes [26]. 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.

III. Related Work

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

High-performance data transfers: High-speed data transfer studies focus on transfer scheduling [27], throughput optimization [7], [28], [29], and power consumption optimization [30]. Globus [23] offers data transfer and sharing services and is well adopted by research community. Yun et al. proposed ProbData [29] 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. [31] presented stochastic gradient descent based solution to tune the number of parallel flows. HARP [28] 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. [30] 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.

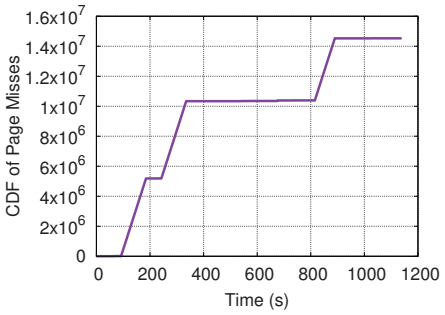


Fig. 3: Transferring a mixed dataset with sequential integrity verification approach reveals that only files that are larger than memory size contribute to page misses during the checksum computation.

Integrity verification: Researchers studied integrity verification in the context of storage outsourcing [20], [32], [33], long term archiving [21], [34], file systems [35]–[37], databases [22], provenance [38], and data transfer [24]. Zhang et al. [37] 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 [23] 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 [24]. 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 [25]. FIVER outperformed state-of-the-art solutions by reducing the overhead of integrity verification from up-to 60% to less than 10%.

Data corruption in storage systems: Studies on disk fault analysis investigates drive failures [39]–[41], latent sector errors [42], and data corruption [13], [14], [37]. 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 [40]. Schroeder et al. [41] analyzed data from 100,000 disks over a five- year period 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 [18], [43]. Bairavasundaram et al. monitored 1.53 million disk drives over 41 months and observed more than 400,000 checksum mismatches [13]. They also found that

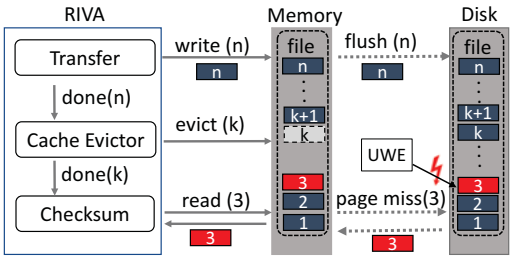


Fig. 4: System architecture of RIVA. When *Checksum* thread attempts to read UWE-exposed page-3, it will trigger a page miss since *Cache Evictor* has evicted it. Consequently, page-3 will be read from the disk and the UWE will be captured.

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 [42]. 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 [14].

IV. System Design

RIVA consists of three threads as shown in Figure 4. *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, *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, *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. 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, 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.

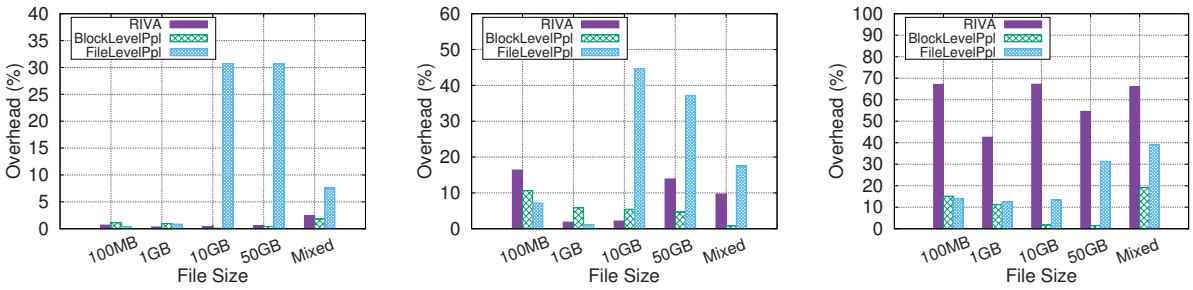
Performance overhead of RIVA relies on the speed of cache eviction and disk read operations. Cache eviction is a quite lightweight operation as it only updates virtual memory mappings of a file. In the worst case, cache eviction for a file chunk takes 0.5 seconds as multiple `munmap` calls are made to 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.

V. Evaluations

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-I. In HPCLab, we have two sets of servers; workstations (HPClab-WS) and data

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 (Emulated)
ESnet	RAID-0	12 x Intel Xeon E5-2643 @3.40GHZ	128	100	89

TABLE I: System specification of networks.



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

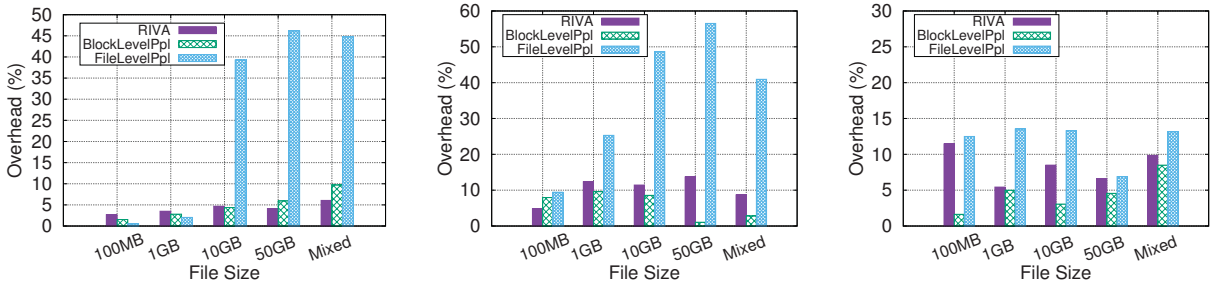
Fig. 5: Performance comparison of algorithms in LAN experiments. While RIVA is able to keep its overhead below 17% in HPCLab and Pronghorn networks, slow disk speed and large memory size in Chameleon Cloud deteriorates its performance significantly.

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.

We compare RIVA against file-level pipelining (FileLevelPpl), block-level pipelining (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 transfer 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 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.

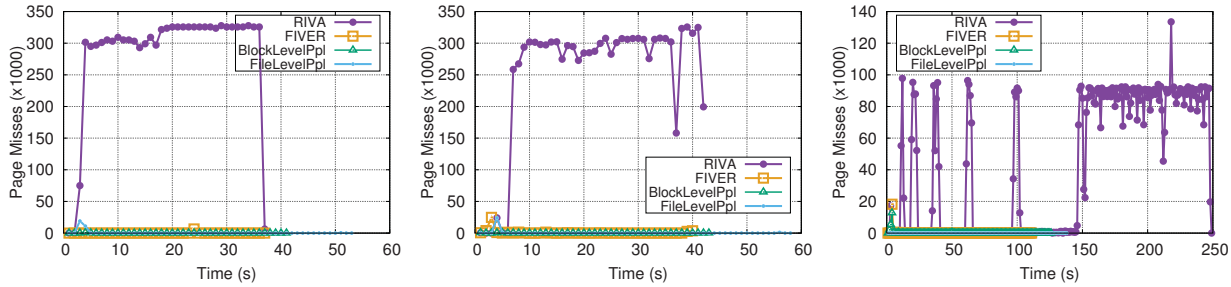
$$Overhead = 100 * \frac{t_{algorithm} - t_{FIVER}}{t_{FIVER}} \quad (1)$$

We conducted extensive experiments in six different networks which are grouped as local-area network (Figure 5) and wide-area network (Figure 6) results. The overhead of RIVA is always less than 5% in HPCLab-WS transfers (Figure 5(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 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(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 overlaps 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



(a) HPCLab-DTN (b) ESnet (c) Pronghorn-Chameleon

Fig. 6: Performance comparison of algorithms in WAN experiments. RIVA is able to keep its overhead below 12% in all networks.



(a) HPCLab-DTN (b) ESnet (c) Chameleon Cloud

Fig. 7: RIVA yields high page miss rates as it evicts file pages from memory and reads directly from disk to capture UWEs.

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 6. 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.

We also collected the page misses (i.e. disk reads rates) during the transfer of a 10GB file in WAN as given in Figure 7. 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 7(a) and 7(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 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 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.

A. 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

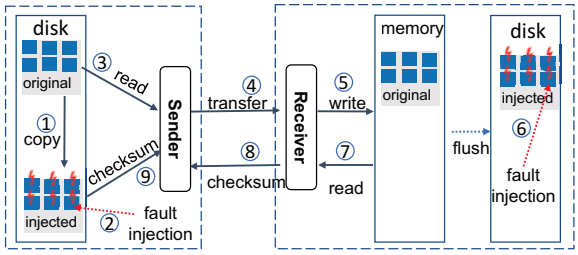


Fig. 8: Extreme-injection test involves injecting UWEs to all pages of files to determine if a transfer application reads any data from page cache.

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

TABLE II: 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.

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, the cached copy will not be updated, creating a similar impact of UWEs as shown in Figure 2(b).

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 8). 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 II 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

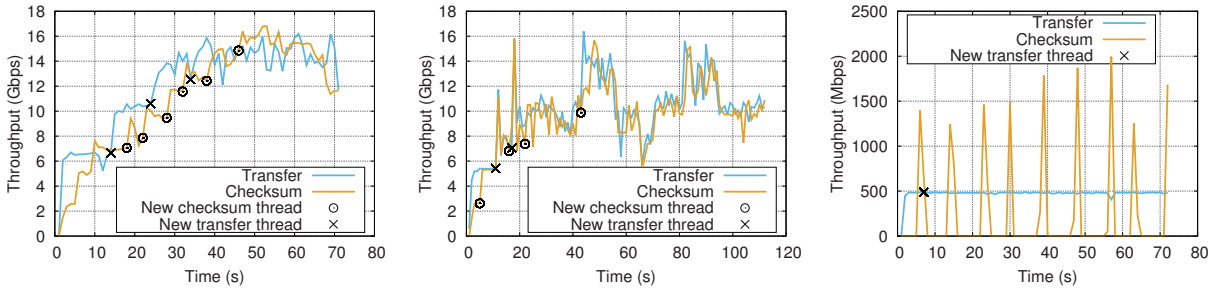
Algorithm 1: Dynamic parallelism of RIVA

```
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
```

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.

B. 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.



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

Fig. 9: RIVA can dynamically adjust the number of transfer and checksum threads to overcome bottlenecks.

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 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 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 ESnet, HPCLab-DTN, and HPCLab-WS networks as illustrated in Figure 9. Single checksum computation thread obtains around 2.4 Gbps speed and limits RIVA performance in HPCLab-DTN and ESnet networks. Dynamic parallelism is able to detect that checksum is the bottleneck and opens up new checksum threads at around 5s as shown in Figure 9(a) and 9(b). Creating more checksum threads eventually helps checksum speed 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 the observed transfer and checksum throughput values stop increasing 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 HPCLab-DTN and ESnet, leading over 5 times improvement over single threaded approach.

On the other hand, it is also possible adding new transfer and checksum threads may not improve performance when single checksum and transfer thread is sufficient to achieve maximum possible performance. For example, RIVA 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 9(c). After realizing that this change did not improve transfer performance, dynamic parallelism terminates its search phase immediately.

VI. Conclusion

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 paper, 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 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.

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] 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.

[2] 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.

[3] 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, October 1990.

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

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

[6] 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. Springer-Verlag, 2013, pp. 725–736.

[7] 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.

[8] 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.

[9] "Dark Energy Survey," https://www.darkenergysurvey.org/.

[10] 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.

[11] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," in *ACM SIGCOMM computer communication review*, vol. 30, no. 4. ACM, 2000, pp. 309–319.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google file system*. ACM, 2003, vol. 37, no. 5.

[20] 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.

[21] 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.

[22] 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.

[23] "Globus," https://www.globus.org/.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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. Piscataway, NJ, USA: IEEE Press, 2016, pp. 25:1–25:12.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population." in *FAST*, vol. 7, no. 1, 2007, pp. 17–23.

[40] 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.

[41] 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, no. 1, 2007, pp. 1–16.

[42] 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*, vol. 35, no. 1. ACM, 2007, pp. 289–300.

[43] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, *IRON file systems*. ACM, 2005, vol. 39, no. 5.