Realistic Assessment of Faults in Storage Systems

César Borges
INESC TEC and U. Minho
Email: cesar.a.borges@inesctec.pt

João Paulo
INESC TEC and U. Minho
Email: joao.t.paulo@inesctec.pt

***Abstract*—With the emergence of new data-centric applications (e.g., data analytics, artificial intelligence) and larger quantities of data to manage, both academia and industry have been actively proposing novel optimizations to improve the speed and reduce costs of current storage solutions. While existing benchmarking frameworks are able to efficiently assess the performance benefits of such optimizations, we argue that these can be improved for evaluating the dependability and reliability of storage systems.**

Therefore we propose ACHILLESBENCH[1], a novel benchmarking framework for assessing the reliability and performance of storage systems. ACHILLESBENCH supports the injection of several types of faults (i.e., data corruption, I/O operations timeouts, and errors) while assessing performance across local storage systems exposing commonly used interfaces (i.e., block device, POSIX) and resorting to complex optimizations such as data deduplication.

Our preliminary experimental results corroborate that ACHILLESBENCH's open-source benchmark is indeed able to combine stress I/O workloads with fault-injection in order to assess simultaneously the reliability and performance of different storage systems and optimizations.

I. INTRODUCTION

Storage systems are essential to persist and ensure fast access to applications' data. These are typically composed of storage devices, such as Hard Disk Drives (HDDs) or Solid State Drives (SSDs), and by additional software layers including different optimizations like caching, deduplication, redundancy, for increased performance, storage space reduction, and reliability purposes [1], [2].

Several studies show that failures can occur frequently at these systems and may lead to downtime and data loss [3], [4]. Even though current solutions provide fault-tolerant designs, it is important to evaluate how reliable these are when faced with different kinds of faults and the impact that these may have on the system's performance and recovery mechanisms.

This paper focuses on evaluating storage performance while facing faults that may occur at the block device layer due to errors on the software managing it or the underlying storage devices supporting it (e.g., HDD or SSD disks). We choose block devices given their high popularity and widespread usage in traditional I/O stacks [5]. Namely, block devices can be accessed directly by applications and file systems (e.g., Ext4, ZFS). Also, the goal of the paper is to understand how faults propagate (e.g., number of corrupted blocks or files), and the impact these have (e.g., in I/O throughput or latency), for applications using directly the block device or through a file system mounted on top of it.

Current fault-injection benchmarking tools can be classified into two major groups [3]. The first includes simulation-based approaches that resort to high-level models of storage systems and corresponding optimizations, that mimic the parameters found in real world deployments [6]–[8]. While these tools are interesting to evaluate an abstract representation of a storage system under different types and distributions of faults, these require accurate input from users when building the model to ensure that it provides an accurate representation of software and hardware stacks found in real world deployments.

The second group includes prototype-based fault-injection benchmarks that aim at evaluating storage solutions in real world deployments [7], [9]–[19]. These benchmarks inject faults at the System Under Testing (SUT) hardware or software layers while monitoring and observing the fault's impact on reliability. Therefore, these types of solutions can be further divided into two sub-groups. The first, targets the hardware layer for fault-injection which can be a costly task, as injecting failures at the hardware level can be more complex and even damage the hardware being assessed [3], [20]. This is why, in recent years, researchers began to explore cheaper and more flexible software-based fault-injection approaches [3].

This paper proposes ACHILLESBENCH, a prototype- and software-based (i.e., independent of underlying hardware) fault-injection benchmark for local storage systems. Briefly, and unlike previous work, our solution is designed to support the evaluation of realistic storage deployments when subjected to different types of failures at the block device layer. Moreover, ACHILLESBENCH aims at assessing the impact of faults in both the performance and reliability of SUTs that resort to commonly used storage interfaces, namely file system (POSIX) and block device APIs. In more detail, the paper provides the following contributions:

- ACHILLESBENCH supports the injection of several types of faults, namely: data corruption, I/O requests delays, and I/O errors. This is possible by intercepting I/O requests from the SUT to the underlying block device layer and injecting these different faults at run time.
- The proposed design enables the evaluation of applications that use directly a block device or, indirectly, through a POSIX file system. The support for the latter interface is challenging since it requires knowing how failures injected at the block device layer propagate to the file system layer (e.g., which file(s) are affected by the corruption of a given block at the storage backend). This is made possible due to a novel content-based fault-

[1]https://github.com/CesarAugustoBorges/AchillesBench

injection algorithm proposed in the paper.

- Furthermore, we show that the previous content-based algorithm is valuable for assessing the reliability and performance of other storage optimizations. Namely, we showcase this benchmark's feature for storage systems contemplating data deduplication, a widely used technique to reduce redundant data at storage solutions.
- ACHILLESBENCH's open-source prototype is built on top of the DEDISBench benchmark, while leveraging the I/O workloads, integrity checker, and performance assessment mechanisms provided by it. Also, our prototype resorts to the BDUS – block-device in user space framework – for implementing the proposed fault-injection features.
- Our preliminary evaluation, contemplating experiments on a raw block device, the VDO deduplication solution [21] and the Ext4 file system, shows that ACHILLESBENCH is able to simultaneously evaluate the performance (i.e., stress the evaluated system in terms of I/O load) and reliability (i.e., inject different faults) of the different SUTs. Moreover, the results highlight the impact of failures, and consequently, the need for ACHILLESBENCH, when deploying storage optimizations such as deduplication, where a single failure (i.e., on a single block) may lead to the corruption of large quantities of data to the application [4].

II. BACKGROUND AND RELATED WORK

Three main categories of faults are typically found in local storage mediums, namely, whole disk failures, Latent Section Errors (LSEs), and Undetected Disk Errors (UDEs). The last two are the most frequent in modern storage systems [22].

LSEs refer to unreachable device sectors that cannot be read or written when a given application accesses them. UDEs, unlike LSEs, can not be repaired by the disk and are only detectable when a read is issued for the affected sector [4].

All these errors have an important aspect in common. Namely, when these occur, the block-device driver accessing (e.g., reading) the storage medium will receive an I/O error response (LSEs) or corrupted data (UDEs). Therefore, the software managing this driver, or the software running on top of it (e.g., application, file system), must include appropriate fault-tolerant mechanisms (e.g., RAID, replications, erasure coding) to handle these failures and avoid problems such as performance degradation or even data loss [4], [22]. Besides that, these fault-tolerant mechanisms must be validated before being deployed in production, which has motivated the emergence of different fault-injection benchmarking tools.

A. Related work

Current fault injection tools can be classified as simulation- or prototype-based, while the latter can be further divided into hardware- and software-based.

Simulation- [6]–[8] and hardware-based [14]–[16] tools are orthogonal to our solution. Our objective is to develop a software-based (i.e., independent of the underlying hardware) benchmark that assesses the performance and reliability of storage systems while injecting faults at run time, and to obtain reproducible results.

Software-based: Recently, researchers have been shifting towards software-based fault injection tools due to their lower complexity, ease of development and use [3]. Examples of such frameworks are depicted in Table I, which also shows the components (i.e., CPU, RAM, Network, Storage) that can be assessed by these frameworks with fault-injection techniques.

Tool	CPU	RAM	Net	Disk
[17], [23]–[28]	-	-	-	yes
[10]	-	yes	-	-
[9], [11]	yes	yes	yes	-
[13]	yes	yes	-	yes
[19], [29]	yes	yes	-	-
[12], [18], [30]	yes	yes	yes	-

TABLE I: Overview of software-based fault-injection tools.

Most tools support fault injection for local storage systems and evaluate the impact of faults at run time [13], [17], [27], or at the offline recovery mechanisms [23]–[26], [28]. Although both approaches inject faults at run time, the first group assumes that the system recovers from faults continuously, assessing the SUT only when the experimental workload is completed. The second group pauses the SUT to impose an offline recovery process, assessing the SUT state after the recovery process. We focus on evaluating how the SUT performs at run time when facing faults that, if not handled correctly, turn into failures or, even worse, errors or crashes.

[13], [17], [27] are able to inject faults in specific storage systems at runtime. However, the first solution requires the virtualized Emulab testbed, the second focuses only on corruption-based faults and requires specific hardware environment for running the targeted SUT, while the third is limited to SCSI devices.

ACHILLESBENCH provides a fault-injection and performance assessment framework for different storage systems without requiring them to run on a specific environment or hardware. The main difference between ACHILLESBENCH and these three solutions is provided by our novel content-aware fault-injection mechanism. Namely, this new feature enables users to inject faults at specific and critical points of I/O workloads. For example, it becomes possible to target specific blocks of a given file or highly duplicated blocks at a given storage backend, which is key to better assess fault-tolerant mechanisms of current file systems and storage optimizations (e.g., data deduplication).

III. ACHILLESBENCH'S DESIGN AND IMPLEMENTATION

ACHILLESBENCH provides a Linux-based benchmarking framework to simultaneously evaluate the performance and reliability of storage systems. The proposed solution is applicable to any Linux-based machine supporting in-kernel block devices or file systems that use block devices as the underlying storage layer. As design principles, fault-injection is conducted

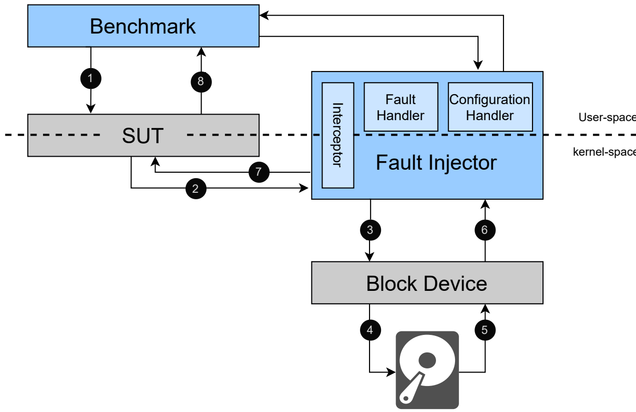

Fig. 1: ACHILLESBENCH's architecture. Blue boxes correspond to contributions of the paper.

at the block device layer and should be transparent to the SUT, thus not requiring any modifications to its source code or mode of operation. Also, the supported I/O workloads must consider both block device and file system-oriented interfaces to enable the validation of a wider range of storage solutions. Examples of SUTs that can be validated with ACHILLESBENCH include block-based middleware solutions residing on top of block devices and providing features such as data deduplication, or even file systems such as Ext4.

Figure 1 depicts the main architecture for ACHILLESBENCH. The *Benchmark* component, residing at the top of the testing software stack, is responsible for mimicking the behavior of an application, namely, the storage I/O operations that a real application would issue to the SUT. Also, it is responsible for injecting faults on some of these requests to validate the SUT's fault-tolerant capabilities.

The previous faults cannot be injected directly at the I/O operations being sent from the *Benchmark* to the SUT (1), as one expects these to manifest only at the underlying block device. Therefore, ACHILLESBENCH contemplates a *Fault Injector* that intercepts I/O operations being issued by the SUT to the block device (2) and, based on user-defined I/O workloads and fault-injection configurations, decides what faults should be applied to what requests.

For example, the *Fault Injector* may force the corruption of a given data block, by intercepting the corresponding write request and writing different content (with the same size) to the block device (3), that is then persisted at the local disk (4). Note that the response for this request then follows the reverse path, being acknowledged back to the SUT and *Benchmark* without showing any visible error (5 to 8). However, if later, the *Benchmark* checks the integrity of written data through the integrity checker, one can validate if the SUT detects data corruption or not (i.e., if it retrieves the correct content back to the benchmark).

As another example, the *Fault Injector* may intercept and delay a read I/O request issued by the SUT, while the *Benchmark* can then assess the performance impact of this type of error for the I/O workload being tested.

A. Benchmark

The *Benchmark* component is based on DEDISbench [1], an open-source synthetic benchmark for storage systems, and leverages the following features:

- Support for POSIX and block-based storage interfaces;
- Workloads with different I/O access patterns (e.g., sequential, random) and operation types (i.e., read, write);
- Write operations follow a realistic content distribution, thus mimicking redundant content found at real storage systems while enabling a more accurate evaluation of storage optimizations such as data deduplication;
- Monitoring and collection of SUT's performance statistics such as I/O throughput and latency.
- Data integrity validation to ensure that the benchmark's write workloads are persisted correctly at the SUT.

To support fault-injection workloads, our work extends DEDISbench by proposing two main changes. Firstly, more configuration parameters are added to the definition of I/O workloads so that users can also specify the type of faults (e.g., I/O corruption, error, delay); when to inject these faults; the targeted I/O operation (e.g., write or read); if the fault is persistent or transient; and the block that will be affected by the fault. This configuration is defined by the user in a *yaml* file and it is loaded before the benchmarking process. This specification allows us to restrict variability and precisely control which blocks are faulty using the different access and I/O patterns. Additionally, with the resources DEDISbench offers, we can reproduce the same workload with minimal randomness, guarantying the same written content and faulty blocks. Secondly, DEDISbench is modified to communicate these fault parameters to a new *Fault Injector* component, which is described next.

B. Fault Injector

The Fault Injector is responsible for intercepting and mediating I/O requests, and corresponding responses, between the SUT and the underlying block-device. Also, it must inject the faults specified by the *Benchmark*'s I/O workloads. This is achieved with three main modules:

The **Configuration Handler** receives information from the *Benchmark* component stating the types of faults to be injected and the I/O requests that must be targeted by these.

The **Interceptor** module intercepts I/O operations at kernel space and redirects these to user-space to be processed synchronously. Requests are intercepted at the block granularity along with their content as well as information about their type (read/write), size and offset[2].

Then, the **Fault Handler** bridges the two previous modules and is responsible for checking each intercepted I/O operation, injecting a fault if desired, and submitting requests to the underlying block device. Note that the responses from submitted operations are then received by the Fault Handler, passed back

[2]The position at the block device where a request is going to be done.

to the Interceptor, forwarded to the SUT, and, finally, passed back to the *Benchmark*.

Our current design supports three main types of faults:

1) **Bit flip:** Flips a bit of the block's content before being written at the underlying block device or read by the application [22]–[24], [27]. Our algorithm flips the rightest bit of the first byte of a block, thus corrupting data.
2) **Medium error:** Makes a given block inaccessible, thus returning an I/O error back to the SUT when reading or writing such block [22], [27].
3) **Slow disk:** Delays a write or read operation by X milliseconds[3] to a given block [17].

C. Selecting I/O requests for fault-injection

The Fault Handler must be able to distinguish what I/O requests are intended for fault-injection and what requests are just forwarded to the underlying device without any changes (regular non-faulty operations). Our current solutions support two complementary solutions.

The first follows an **offset-based** approach meaning that a given fault is only injected for I/O requests being issued at a specific offset of the underlying block device. In this case, the Fault Handler just needs to match the offset and operation type (read/write) of intercepted requests with the fault rules kept at the Configuration Handler.

The second approach is **content-based** and ensures that the Fault Handler only injects a given fault for I/O operations manipulating blocks with specific content. The module first divides the intercepted I/O request's content[4] into fixed size blocks, computes a hash sum for each block and, then, checks the Configuration Handler for faults for those given hash sums (content summaries). If a matching block is found, the fault is applied to the full I/O request. In the case of a corruption fault (bit flip), only the content of the matching block is targeted.

Note that fault-injection rules stored at the Configuration Handler can specify either persistent or transient faults. Therefore, if an offset-based rule is defined to be transient, it is only enforced for the first corresponding I/O request. In terms of applicability, the offset-based approach is useful for scenarios where users know the offset at the block device that should be compromised. We next detail two scenarios where using the content-based approach can be more useful.

Starting with the case where a file system is the SUT, when a file is being written by the *Benchmark*, it is impossible to know the exact offset where the file's blocks will end up at the underlying block device. But, since our framework is based on DEDISbench and, therefore, it supports realistic content generation for written blocks, it becomes possible to track the content of a specific file's block at the block device. Namely, the user can specify a fault injection rule for *corrupting the block being written at offset X of file A*. Internally, since the Benchmark knows *a priori* the content that is going to be written for that file's block, it just needs to calculate the corresponding hash sum, transparently update the fault policy to *corrupt a block with hash sum H*, send it to the Configuration Handler, and then issue the SUT's I/O request.

Another interesting feature of basing our framework in DEDISbench is that it has access to other runtime statistics, namely, the number of times that a given content (hash sum) is being written to the SUT during the experiments (duplicate blocks). Therefore, with the content-based algorithm, it becomes straightforward to support new user policies that, for instance, *corrupt the block with the highest number of duplicates*. Again, internally, the *Benchmark* translates this rule into a concrete hash sum that is sent to the Configuration Handler. To the best of our knowledge, this is a novel feature provided by our work that can be very valuable to assess the dependability of data deduplication. Briefly, block-based deduplication systems reduce redundant storage space by transparently pointing several write requests (logical blocks), operating over duplicate content, to a single physical block at the underlying block device [31]. However, the corruption of a physical block can lead to the loss of several logical blocks (or files) sharing the same content.

D. Implementation

ACHILLESBENCH is implemented in C and the *Benchmark* component is based on DEDISBench v1.1.0 [32]. BDUS v0.0.9 [33], [34], a framework for implementing block devices in user-space, is used to implement the I/O interception and processing logics of the *Fault Injector*. The communication between the *Benchmark* and *Fault Injector* is done with Unix Domain Sockets. Hash sums are calculated by resorting to the XXH3_128 hashing algorithm v0.8.0 [35], while the block size is configurable by users.

IV. EXPERIMENTAL METHODOLOGY

Our preliminary experimental evaluation aims at validating the following research questions:

- Is ACHILLESBENCH capable of assessing simultaneously the performance and reliability of storage systems?
- What is the overhead of fault-injection on the benchmark's performance?
- Is ACHILLESBENCH able to support different SUTs and storage interfaces?
- What is the impact of faults, in terms of performance and reliability, for the different SUTs?

To answer these questions we devised the following experimental methodology.

Experimental environment. Experiments ran on identical servers equipped with one Intel(R) Core(TM) i5-9500 CPU (6 cores), 16GB of RAM, an NVMe SSD disk with 250 GB, and a WDC HDD disk with 500 GB. Software-wise, servers ran Ubuntu 20.04 LTS with kernel 5.4.0-71. To avoid interference, the NVMe disk was used to support the experimental workloads, while another HDD disk was used to run the operating system and to store benchmarking logs and collected metrics.

[3] The delay parameter is defined by users.

[4] For read operations it is the I/O response's content.

I/O workloads. The benchmark was configured to run sequential write tests, with a single process and a block size of 4KiB. Also, each test wrote 64 GiBs to the NVMe device.

We used the DEDISbench's *dist_kernels* distribution for realistic content generation. After each experiment, a full integrity check was done for the written content to check for potential data corruption.

Fault-injection workloads. For the experiments considering faults, these were injected at the last block, chosen at runtime, of each GiB written to the SUT by our benchmark. Therefore, each experiment includes the injection of 64 faults and the overhead of choosing which block is being injected with the fault.

Three types of faults were addressed in our evaluation, namely, bit-flips, medium errors, and slow disk faults. The latter type was configured to slow I/O requests by 1 second.

Collected metrics. From ACHILLESBENCH, we collected the SUT's **I/O throughput and latency** to evaluate performance, as well as the **number of faults injected** and the corresponding **number of failures detected**[5] to assess reliability. Also, Pidstat v12.2.0 [36] was used to observe **CPU** and **memory** usage at the servers running the experiments. The values shown for each experiment in the next section refer to the average and standard deviation of 5 independent runs.

Systems We considered three different SUTs:

- **Bdev.** A standard block device using the NVMe disk as storage backend.
- **Ext4.** An Ext4 file system, that only journals metadata (data=ordered), mounted on top of the previous block device.
- **VDO.** The Virtual Data Optimizer block-based deduplication system (version 6.2.5.11) [36], using also the block device. VDO was configured with asynchronous deduplication, a logical storage size of 700 GiB (as suggested by the documentation), and the compression feature was disabled.

Note that neither these systems nor our deployments consider fault-tolerance mechanisms. The goal of this preliminary evaluation is to measure the impact of faults without them.

V. EXPERIMENTAL RESULTS

We now show and discuss the results obtained for the different experiments.

A. Throughput and latency analysis

Table II depicts the I/O throughput and latency for the three SUTs (Bdev, Ext4, and VDO) when assessed with different types of failures and configurations.

The *Bdev-baseline* setup corresponds to a baseline deployment of our benchmark without any fault-injection mechanisms. The reason for evaluating this setup is that it performs similarly to the original DEDISbench benchmark and, thus, can be used to assess the performance impact of ACHILLESBENCH's fault-injection mechanisms, which are only used at the other setups.

[5]Failures are reported at the integrity checking phase.

Setup	Fault Type	Throughput (MiB/s)	Latency (ms)
Bdev	baseline	465.39 ± 2.23	0.005 ± 0
Bdev	bit flip (O)	458.03 ± 3.51	0.005 ± 0
Bdev	bit flip (C)	462.45 ± 1.68	0.005 ± 0
Bdev	medium error (O)	450.59 ± 11.12	0.005 ± 0
Bdev	slow disk (O)	94.82 ± 0.18	0.036 ± 0
Ext4	bit flip (C)	99.34 ± 0.49	0.037 ± 0
VDO	bit flip (C)	38.81 ± 0.36	0.097 ± 0.001
VDO	bit flip (top C)	44.14 ± 0.33	0.086 ± 0.001
VDO	bit flip (unique C)	44.33 ± 0.40	0.085 ± 0.001

TABLE II: Throughput and latency for the different SUTs (Bdev, Ext4 and VDO), types of faults (bit flip, medium error, and slow disk) and for the offset- (O) and content-based (C) algorithms.

The *Bdev-bitflip (O)* setup does bit flip fault injection by following an offset-based approach, while the *Bdev-bitflip (C)* follows our novel content-based algorithm. Nonetheless, these two setups inject faults by following the same rule, i.e., at the last block of each GiB written by our benchmark.

When comparing the throughput of the baseline setup (465.39 MiB/s), with the *Bdev-bitflip (O)* (458.03 MiB/s) and *Bdev-bitflip (C)* (462 MiB/s) setups, we can see that the difference is minimal. The same is true for latency results, thus showing that the provided fault-injection mechanisms and the offset- and content-based algorithms perform similarly.

Before analyzing the next results, it is important to mention that the offset-based algorithm (O) was used for all the remaining experiments with the Bdev system, while the content-based algorithm (C) was used for the experiments with the Ext4 and VDO solutions. The reason for this choice was previously explained in Section III-C.

Moreover, for VDO, we have considered three different fault-injections scenarios. The first (*VDO-bitflip (C)*) follows an identical approach to the previously mentioned experiments since a bit flip fault is injected at the last block of each GiB written by our benchmark. The second (*VDO-bitflip (top C)*) injects a bit flip fault, at the end of every GiB written by the benchmark, but this fault is targeted towards a stored block with a high number of duplicates. The third scenario (*VDO-bitflip (unique C)*) injects a bit flip fault, at the end of every GiB written by the benchmark, but this fault is targeted towards a stored block with unique content. As discussed next, these different scenarios are important to validate the reliability of VDO's data deduplication feature.

Going back into the performance results, the *Bdev-medium error (O)* (450.59 MiB/s) and *Bdev-bitflip (O)* (458.03 MiB/s) setups exhibit similar performance, with a small drop in the average throughput and increase in variance for the former.

More interestingly, when a slow disk fault (*Bdev-slow disk (O)* is being used instead, there is a visible decrease in throughput (94.82 MiB/s). Of course, this decrease is related with the delay configured for the corresponding fault (1 second in our experiments). Nevertheless, it is important to note that this significant performance drop happens even when the benchmark is only delaying 64 I/O requests of the full storage

Setup	Fault Type	Injected faults	Reported failures
Bdev	baseline	0	0
Bdev	bit flip (O)	64	64 ± 0
Bdev	bit flip (C)	64	64 ± 0
Bdev	medium error (O)	64	8113.8 ± 46.6193
Bdev	slow disk (O)	64	0 ± 0
Ext4	bit flip (C)	64	64 ± 0
VDO	bit flip (C)	64	378.8 ± 14.7838
VDO	bit flip (top C)	64	11519.4 ± 2.2450
VDO	bit flip (unique C)	64	64 ± 0

TABLE III: Number of injected faults and reported failures (integrity check) for the different SUTs (Bdev, Ext4 and VDO), types of faults (bit flip, medium error, and slow disk) and for the offset- (O) and content-based (C) algorithms.

workload, which is writing more than 16 million blocks.

For Ext4, the performance (99.34 MiB/s) is lower then using a standard block device due to the overhead of using a file system, while, again, the injection of bit flip faults does not affect the overall performance. On the other hand, we notice a significant decrease in throughput ($\approx$ 40 MB/s) for VDO under all fault-injection scenarios. This performance decrease is related with the I/O overhead introduced by the data deduplication feature supported by this system.

B. *Fault injection and failure propagation*

Table III illustrates the number of injected faults and failures generated from these for all the experiments.

As expected, the *Bdev-baseline* setup does not include any faults or failures. On the other hand, introducing bit flips generates 64 corrupted blocks at the Bdev and Ext4 SUTs. No failures are reported for the slow disk fault since the benchmark is just introducing a delay on I/O requests and, therefore, not compromising their reliability.

For the medium error experiment, there are 8113 failures reported. As explained in Section III-C, each I/O request intercepted by ACHILLESBENCH can include multiple blocks (contiguous offsets) to be written or read. Therefore, a medium error for a given block (offset) may be propagated to others if these are intercepted in the same I/O request, thus explaining the high number of reported failures.

Interestingly, the number of failures (corrupted data) also changes across the three VDO scenarios. For the *VDO-bit flip (c)* experiment, 378 failures were reported in average. This happens because the blocks targeted for bit flip injection will have different numbers of duplicate blocks already persisted at the storage medium which, in turn, will affect the amount of corrupted blocks reported.

To help understating this observation one can look at the other two experiments. On one hand, in the *VDO-bit flip (unique c)* experiment, the benchmark is only targeting blocks with unique content at the storage medium. Thus, the number of reported failures is always 64 (best-case scenario in terms of failure propagation). On the other hand, the *VDO-bit flip (top c)* is always targeting blocks with a high number of duplicates at the storage medium. Therefore, the number of reported failures greatly increases (11,519 blocks). These results showcase the need for using ACHILLESBENCH when evaluating the reliability of solutions with built-in data deduplication.

The variance in the number of reported failures for the medium error and VDO experiments is related with the non-determinism associated with our real deployment and experiments. This way, as each run of a given experiment may change in terms of I/O performance and in the way data is written to the SUT, the number of reported failures also changes accordingly.

C. *CPU and memory consumption*

The CPU ($\approx$ 59%) and memory ($\approx$ 2.3 GB) consumption is identical for all experiments with the BDev and Ext4 SUTs. On the other hand, VDO experiments have lower CPU ($\approx$ 8%) and memory consumption ($\approx$ 1 GB) due to the lower throughput observed at these.

VI. CONCLUSION AND FUTURE WORK

ACHILLESBENCH proposes a new benchmarking framework for evaluating the performance and reliability of storage systems. As main design principles, our solution is focused on the evaluation of multiple storage optimizations and interfaces (i.e., POSIX and block device), while supporting different types of faults (i.e., data corruption, I/O errors and delays).

The preliminary experimental results validate several key points about ACHILLESBENCH. Firstly, it can simultaneously evaluate the performance and reliability of storage solutions, while the proposed fault-injection mechanisms introduce low overhead in the I/O critical path.

Moreover, the results show that our prototype can indeed evaluate several storage solutions, exposing different I/O interfaces, and that different types of faults have specific impacts on the SUT's performance and reliability. As examples, medium errors can lead to the unavailability of several storage blocks, and a small number of I/O delays can have a deep impact on performance. Interestingly, the results motivate the case for using ACHILLESBENCH when evaluating storage optimizations such as data deduplication where bit flips can lead to extensive data corruption and have a deep impact on storage reliability.

As future work, it would be important to validate ACHILLESBENCH with other SUTs, fault-tolerant mechanisms, storage optimizations (e.g., data compression, caching), and types of faults (e.g., intermittent faults, misplaced operations). Also, it would be important to consider other I/O workloads (e.g., read workloads, different access patterns). Evolving ACHILLESBENCH to be able to evaluate distributed storage systems is also an interesting research path left open by this work.

ACKNOWLEDGMENT

This work is funded by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UTA-EXPL/CA/0080/2019

REFERENCES

[1] J. Paulo, P. Reis, J. Pereira, and A. L. Sousa, "Towards an accurate evaluation of deduplicated storage systems," *Computer systems science and engineering*, 2013.

[2] R. J. T. Morris and B. J. Truskowski, "The evolution of storage systems," *IBM Systems Journal*, 2003.

[3] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, 1997.

[4] E. Rozier, W. H. Sanders, P. Zhou, N. Mandagere, S. M. Uttamchandani, and M. L. Yakushev, "Modeling the fault tolerance consequences of deduplication," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, 2011.

[5] D. Meyer, B. Cully, J. Wires, N. Hutchinson, and A. Warfield, "Block mason," *First Workshop on I/O Virtualization*, 2008.

[6] V. Sieh, O. Tschache, and F. Balbach, "Verify: evaluation of reliability using vhdl-models with embedded fault descriptions," in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, 1997.

[7] C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, and H. Vierhaus, "A register-transfer-level fault simulator for permanent and transient faults in embedded processors," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 2001.

[8] L. Entrena, C. Ongil, and E. Olías, "Automatic generation of fault tolerant vhdl designs in rtl," *Forum on Design Languages*, 2001.

[9] S. Han, K. G. Shin, and H. A. Rosenberg, "Doctor: an integrated software fault injection environment for distributed real-time systems," in *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, 1995.

[10] A. Benso, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "Exfi a low cost fault injection system for embedded microprocessor based boards," *Association for Computing Machinery Transactions (ACM) on Design Automation of Electronic Systems (TODAES)*, 1998.

[11] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: a tool for the validation of system dependability properties," in *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, 1992.

[12] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Ysskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "Fiat - fault injection based automated testing environment," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*, 1995.

[13] T. Tsai and R. Iyer, "Measuring fault tolerance with the ftape fault injection tool," *Lecture Notes in Computer Science*, 1995.

[14] G. B. Finelli, "Characterization of fault recovery through fault injection on ftmp," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Reliability*, 1987.

[15] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: the mars approach," *Institute of Electrical and Electronics Engineers (IEEE) Micro*, 1989.

[16] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. . Fabre, J. . Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Software Engineering*, 1990.

[17] Y. Naik, M. Hibler, E. Eide, and R. Ricci, "Building a disk failure injection framework for fault-tolerant systems research."

[18] H. A. Rosenberg and K. G. Shin, "Software fault injection and its application in distributed systems," in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993.

[19] J. Carreira, H. Madeira, J. Silva, and D. Informática, "Xception: Software fault injection and monitoring in processor functional units," *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications*, 2001.

[20] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *The International Arab Journal of Information Technology*, pp. 171–186, 2004.

[21] "A look at vdo, the new linux compression layer," https://www.redhat.com/en/blog/look-vdo-new-linux-compression-layer, accessed: 7, 2021.

[22] E. Rozier, "Understanding the fault-tolerance properties of large-scale storage systems," Ph.D. dissertation, 2011, dept. Comput. Sci., Univ. Illinois Urbana-Champaign, Champaign, IL, USA.

[23] S. Jaffer, S. Maneas, A. Hwang, and B. Schroeder, "Evaluating file system reliability on solid state drives," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[24] A. Rebello, Y. Patel, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Can applications recover from fsync failures?" in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[25] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "All file systems are not created equal: On the complexity of crafting crash-consistent applications," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[26] O. R. Gatla and M. Zheng, "Understanding the fault resilience of file system checkers," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[27] L. Lu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Fault isolation and quick recovery in isolation file systems," in *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, 2013.

[28] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," in *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, 2004.

[29] W.-I. Kao, R. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *IEEE Transactions on Software Engineering*, 1993.

[30] W.-L. Kao and R. Iyer, "Define: a distributed fault injection and monitoring environment," in *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1994.

[31] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *Association for Computing Machinery (ACM) Computing Surveys*, 2014.

[32] "Dedisbench github," https://github.com/jtpaulo/dedisbench, accessed: 7, 2021.

[33] A. Faria, R. Macedo, J. Pereira, and J. a. Paulo, "Bdus: Implementing block devices in user space," ser. International Systems and Storage Conference (SYSTOR '21). Association for Computing Machinery, 2021.

[34] "bdus github," https://github.com/albertofaria/bdus, accessed: 7, 2021.

[35] "xxhash github," https://github.com/Cyan4973/xxHash, accessed: 7, 2021.

[36] "pidstat documentation," https://man7.org/linux/man-pages/man1/pidstat.1.html, accessed: 7, 2021.