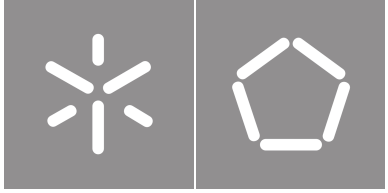
**Universidade do Minho**
Escola de Engenharia

Alexandre Esteves Miranda

# Realistic Fault Assessment in SPDK-enabled Storage Stacks

February, 2023



**Universidade do Minho**
Escola de Engenharia

Alexandre Esteves Miranda

# Realistic Fault Assessment in SPDK-enabled Storage Stacks

Master's Dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
**João Tiago Medeiros Paulo (advisor)**
**Ricardo Gonçalves Macedo (advisor in Research Center)**

February, 2023

# Acknowledgements

This Master's dissertation brings to an end a very long, tough and yet amazing journey at the University of Minho. This 5 years ride would not be possible without the presence of all the people that stood by my side providing all the help and guidance I needed to achieve my goals.

First, I want to thank to both supervisors of this project. To João Tiago Paulo, I am grateful for the opportunity you gave me to develop such interesting project and for all your consistent guidance and concern. To Ricardo Macedo, thank you for all your always instructive advices and sharp corrections. And thank you both for including me in the HASlab family where I met so many colleagues that I will never forget.

Thank you to all my University colleagues who shared this journey with me. To Alexandre Ferreira, a big thank you for putting up with me since we were boys, your help and support throughout the past 18 years was key for me to get to where I am today. Thank you to João Azevedo for the help in solving the problems I could not solve alone, not only in this dissertation but throughout all my academic journey. To Paulo Araújo, thank you for making boring moments always fun and for helping me getting to the end always with a smile in my face.

To my family, specially my parents and brother, I am so grateful for all you support and guidance throughout my academic and personal life. Without your education and encouragement to face my challenges throughout my life I certainly would not be where I am today.

Lastly, I would like to present my gratitude to the Portuguese Foundation for Science and Technology (FCT), for sponsoring the development of this dissertation.

## STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

# Resumo

## Avaliação Realista de Faltas em Stacks de Armazenamento SPDK

A eficiência e desempenho das operações de Entrada/Saída(E/S) são aspetos fundamentais na implementação de sistemas de armazenamento. A maioria das soluções atuais são implementadas em kernel, obrigando a trocas de contexto entre espaço de utilizador e kernel por parte das aplicações. Estas trocas de contexto são custosas e, por isso, limitam o desempenho do sistema de armazenamento. A plataforma Storage Performance Development Kit (SPDK) disponibiliza uma forma de construir estes sistemas evitando o acesso a kernel, realizando todas as operações de E/S necessárias diretamente do espaço de utilizador para o disco físico.

Contudo, os dados continuam a ter que ser guardados com garantias de persistência. Assim sendo, os sistemas construídos com SPDK devem se tolerantes a faltas para garantir resiliência em cenários de falta. A inexistência de uma ferramenta capaz de testar essa resiliência em sistemas de armazenamento construídos com SPDK é um problema para os programadores que querem testar a resiliência dos seus sistemas.

De forma a resolver este problema, esta dissertação propõe o Fault Injector in SPDK (FISPDK), uma ferramenta que estende o SPDK e fornece injeção de faltas determinística ao nível do block device. Para injetar faltas deterministicamente, FISPDK utiliza diferenciação de pedidos E/S de forma a identificar quais pedidos devem (ou não) ser injetados com uma falta. Para isso, o FISPDK implementa mecanismos de propagação de contexto, que permitem passar informação da aplicação para os níveis mais baixos das pilhas de E/S, e é baseado numa extensão da block device Application Programming Interface (API) original do SPDK. Para providenciar injeção de faltas, o FISPDK apresenta um block device virtual que interceta pedidos E/S e injeta corrupção de dados ou atraso neles. O block device virtual pode ser configurado pelos utilizadores para apontar os tipos de faltas e quais os pedidos que devem ser injetados com essas faltas.

Uma avaliação compreensiva do FISPDK demonstra que a nossa solução consegue injetar faltas de forma determinística e avaliar a tolerância a faltas de sistemas de armazenamento que usam SPDK, sem adicionar uma sobrecarga significativa à pilha de armazenamento.

**Palavras-chave:** Armazenamento, Espaço de Utilizador, Tolerância a Faltas, SPDK, Propagação de Contexto

# Abstract

## Realistic Fault Assessment in SPDK-enabled Storage Stacks

The efficiency and performance of Input/Output (I/O) operations are two of the most important aspects in the implementation of a storage system. Most of the current solutions are implemented in kernel, forcing context switches between user-space and kernel by applications. These context switches are costly and, therefore, limit the performance of the storage system. The SPDK framework provides a way to build these systems by bypassing all kernel components, performing all necessary I/O operations from user-level applications directly to the physical device.

However, data still needs to be stored persistently. Therefore, systems built with SPDK must guarantee fault tolerance, to ensure resilience under fault scenarios. The nonexistence of a tool that is able to test the fault tolerance of SPDK-compliant systems is a problem to developers that want to test the resilience their systems.

To solve this problem, this dissertation presents FISPDK, a tool that extends SPDK and provides deterministic fault injection at the block device level. To inject faults in a deterministic way, FISPDK resorts to I/O differentiation to be able to identify which requests must be (or not) injected with a fault. For that, FISPDK implements a context propagation mechanisms,which enables passing application-level information to the lower-levels of the I/O stacks, and is based on an extension of the original block device API of SPDK. To provide fault injection, FISPDK presents a virtual block device that intecepts I/O requests and injects data corruption and delay into these. The virtual block device can be configured by users to pinpoint the type of faults and the I/O requests targeted by these.

A comprehensive experimental evaluation of FISPDK shows that our solution is able to deterministically inject faults and evaluate the fault tolerance of the SPDK-compliant storage systems, without adding significant performance overhead into the storage stack.

**Keywords:** Storage, User Space, Fault Tolerance, SPDK, Context Propagation

# Contents

# List of Figures

# List of Tables

# List of Listings

# Glossary

**stale data** is data that is in cache but is not in the most recent version committed to the data source. 12

# Acronyms

| | |
|---|---|
| $B^3$ | Bounded Black-Box 2, 15 |
| **ACE** | Automatic Crash Explorer 15, 16 |
| **API** | Application Programming Interface vii, viii, 3, 6, 9, 10, 21, 22, 23, 28, 31, 32, 33, 35, 43, 44, 45 |
| **CPU** | Central Processing Unit 5, 6, 14, 41, 43, 44 |
| **DPDK** | Data Plane Development Kit 1 |
| **FERRARI** | Fault and Error Automatic Real-Time Injection 14 |
| **FIO** | Flexible I/O tester 3, 35, 36, 40, 41, 44 |
| **FISPDK** | Fault Injector in SPDK vii, viii, xi, 3, 4, 21, 22, 23, 27, 31, 32, 33, 34, 35, 36, 37, 39, 40, 41, 42, 43, 44, 45 |
| **FTAPE** | Fault Tolerance And Performance Evaluator 14 |
| **I/O** | Input/Output viii, xi, 1, 2, 3, 5, 6, 7, 8, 9, 10, 16, 17, 18, 20, 21, 22, 23, 24, 26, 28, 30, 31, 32, 44, 45 |
| **iSCSI** | Internet Small Computer System Interface 6 |
| **JSON** | JavaScript Object Notation 30 |
| **LoC** | Lines of Code 27, 28, 29, 33, 44 |
| **LSEs** | Latent Section Errors 11, 17, 18, 19, 45 |

**NVMe** Non-Volatile Memory Express 1, 5, 6, 9, 34, 35, 36

**PMDK** Persistent Memory Development Kit 1

**POSIX** Portable Operating System Interface 9, 10

**RAID** Redundant Array Of Independent Disks 2, 7, 11

**SCSI** Small Computer System Interface 9, 11

**SPDK** Storage Performance Development Kit vii, viii, ix, xi, xii, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 19, 20, 21, 22, 23, 24, 27, 28, 30, 31, 32, 34, 35, 37, 41, 43, 44, 45

**SSD** Solid State Drive 1

**SUT** System Under Test 32, 34

**TLS** Thread Local Storage 32

**UDEs** Undetected Disk Errors 11, 12, 17, 18, 19, 20, 23, 30

**UREs** Undetected Read Errors 11, 12

**UWEs** Undetected Write Errors 12

# 1 Introduction

Storage systems are used by a vast group of applications to efficiently store and access digital information. Most storage systems use the traditional multi-layered architecture composed of both user and kernel space [22, 13, 28]. This results in a complex Input/Output (I/O) stack that every I/O request must go through. In user space one has applications, such as databases, the local storage (like the memory heap), while, in kernel space, there are several layers like the file system, the page cache, the block layer (which includes the I/O scheduler with several queues that manage the scheduling of the I/O), and the disk driver. An I/O request that is submitted by an application must pass through all these layers including a context switch, between user and kernel space, before it gets to the physical disk. Going through all these stacks adds extra latency to requests [37, 35] and with the advances in storage technologies and the appearance of faster storage devices (e.g. Solid State Drive (SSD) and Non-Volatile Memory Express (NVMe)), these problems gained more importance. With faster physical disks the context changes became the main bottleneck in the storage systems.

This problem led to the appearance of user-level development kits like Storage Performance Development Kit (SPDK) [37] for storage, Data Plane Development Kit (DPDK) [8] for network and Persistent Memory Development Kit (PMDK) [30] for persistent memory. These tools have the ability to bypass the kernel enabling the applications that use them to access directly the storage devices, saving the time of context switching in every I/O operation, and so having a positive impact by reducing the latency of storage requests.

In particular, SPDK is a framework that allows one to build a fully customized storage stack for user-space applications. The framework offers the possibility of creating custom virtual block devices, allowing users to arrange the different block devices the way they want, creating a fully customized flow for their I/O requests. For instance it is possible to create a block device that encrypts the data of the requests and below that it is possible to have one that has cache functions and so on.

Despite of being a very important aspect, good performance is not the only crucial requirement in today's storage systems. Reliability is also a key requirement for today's storage solutions [31]. Indeed, failures at these systems can compromise the durability and availability of stored critical data. In traditional systems, fault tolerance is ensured in many different levels, for example, at the disk level there is the usage

of Redundant Array Of Independent Disks (RAID), at the file system level there is the usage of checksums to ensure the correctness of the persisted data, the usage of log files at the data bases level is another example of ensuring fault tolerance. With reliability being a main factor in storage systems, the existence of tools that are able to simulate and inject faults in the storage systems and evaluate their behavior is key for having a way to validate that a system is indeed fault tolerant. Moreover, these tools allow us to identify potential bugs and even performance limitations for solutions implementing fault tolerant features and corresponding recovery mechanisms. Reliability remains crucial in SPDK [37] storage systems, however, the tool's main focus has been the performance enhancement and there are no tools capable of testing reliability in these systems.

## 1.1 Problem Statement and Objectives

Production-ready storage systems provide fault-tolerant designs including different mechanisms to ensure their reliability, such as the use of redundancy approaches (e.g., use of RAID disks). However, this does not exclude the need to have appropriate benchmarking tools that can help validating these fault-tolerant features. Namely, validating the impact of failures and the performance of recovery from these, is an important aspect that this kind of tools can help with. Storage devices can have multiple failures like checksums mismatches, identity discrepancies, parity inconsistencies [3] and many others, while users' data can be at risk if the storage systems do not have mechanisms capable of overcoming these problems. While existing tools that can simulate and inject faults in kernel-level storage systems like AchillesBench [7], FTAPE [36] or Bounded Black-Box ($B^3$) Crash Testing [23], these are not applicable with SPDK-based storage solutions, which bypass the kernel. In fact, many questions are yet to be answered with regard to the fault tolerance mechanisms of the systems and applications that resort to SPDK. However, there is not any framework or tool that is able to simulate and inject faults in this new type of user level storage solutions. The creation of such a tool is not trivial, and raises several challenges that need to be addressed:

- **Performance overhead:** The tool must not add much overhead to the system not only from the usability point of view (i.e., users should be able to test their solutions with a high load of I/O requests in a reasonable time window), but also to avoid masking the potential performance overhead in I/O requests caused by fault-tolerant mechanisms.

- **Fault injection:** Defining where exactly the tool will perform the fault injection is another challenge. For instance if users want to inject faults in I/O requests of a specific file and the fault injection tool works the block device level, normally the tool does not have access to what file each request is related to at the block device level, because block devices only work with disk offsets, memory blocks and request sizes. The file-system, at the upper level, is the one that makes the correspondence between in memory positions and the files.

- **Loss of context between I/O layers:** This challenge is related to the previous one. It is not possible to know where to inject a fault at the block device layer if there is no additional information about the I/O. The tool must have some mechanism to allow that additional information can be propagated to the level where the fault injection takes place.

The main goal of this thesis is to allow users to inject faults in kernel-bypass storage systems, in this case in systems that use SPDK. We intend to offer the possibility to test the resilience to storage faults of SPDK-compliant systems. To do so we developed a tool that is able to fulfill all the aforementioned challenges.

## 1.2 Contributions

This thesis presents three main contributions to achieve the aforementioned goals.

**Fault injection tool for SPDK stacks.** As the first contribution of this thesis, we propose the design and implementation of Fault Injector in SPDK (FISPDK), a fault injection tool for user-level storage stacks built with SPDK. This tool is able to intercept write and read operations and it is able to inject content corruption and delay in these requests. The framework is also composed of an extension of the SPDK's block device Application Programming Interface (API) that allows the applications to propagate context to the block device layer. This design was implemented as an SPDK virtual block device that can be customized with a configuration file to inject faults. The injection of the faults is provided by another thesis contribution: the faults library.

**Integration of the prototype with real systems.** The second contribution is the integration of the prototype with other SPDK applications, namely the Blobstore[1] and BlobFS[2]. This implied the extension of the Blobstore API in order to be able to propagate context and also some changes in the BlobFS code, to propagate context in the write and read operations, in this case the file name to which the requests are related to. With this changes we intend to be able to test the resilience of BlobFS to either content corruption and slow disk.

**Experimental Evaluation.** Finally, we conducted an experimental performance evaluation of FISPDK by using the Flexible I/O tester (FIO) plugin for SPDK. The results show that the tool presents good performance and does not add significant overhead into the SPDK storage stacks. On top of that, FISPDK was used to test the fault tolerance of BlobFS in order to prove the utility of the tool. The results indicate that the system tested is not fault tolerant.

[1] Blobstore Programmer's Guide. url: `https://spdk.io/doc/blob.html`
[2] BlobFS Getting Started Guide. url: `https://spdk.io/doc/blobfs.html`

## 1.3 Outline

This document is organized as follows. In Chapter 2 we provide a background on the SPDK platform, present different types of storage faults and discuss how faults can be injected with different techniques. Further, this chapter also presents related work on fault injection over storage systems. Chapter 3 presents the designed tool for fault injection in SPDK. It displays both architecture and implementation decisions. On top of that, this chapter presents how it is possible to adapt SPDK-compliant systems to use FISPDK and also how to adapt a virtual block device for it to be compatible with FISPDK. Chapter 4 evaluates the system under different testing scenarios. The chapter begins with the methodology used to validate the fault injection tool, and then it presents the results of applying that methodology. Finally, Chapter 5 presents the final remarks of this thesis and discusses possible future work.

# 2

# Background

This chapter presents background knowledge that is key to understand the topics discussed in this thesis. It starts with an overview of the SPDK framework, detailing its main architecture, how to build custom storage systems with SPDK, how its different components act and how can an application be SPDK-compliant.

Further we also present some background on fault injection, discussing different types of faults that can happen in storage systems and how to inject them.

Finally, we survey current work on storage fault injection.

## 2.1 Storage Performance Development Kit

SPDK consists in a set of tools and libraries for writing high performance, scalable, user-level storage applications [37]. The main goal of the framework is to offer higher performance for the I/O requests than the traditional kernel storage solutions by moving storage requests handling to user-space, removing unnecessary context-switching between kernel and user-space and reducing the amount of I/O layers that a given request needs to traverse before reaching the storage device.

This tool is divided in four main components: drivers, application scheduling, storage devices and storage protocols [37], as depicted in Figure 1.

- **Drivers**: is the main component, and enables the operating system to communicate with a device. In a traditional infrastructure, the disk drivers are implemented in kernel, forcing the context switching between user-space and kernel for all disk operations. SPDK provides zero-copy, highly parallel and direct access to NVMe disk for user-space applications via the user-space polled mode (which means that is the Central Processing Unit (CPU) job to see if the component requires attention) NVMe driver.

- **Storage services**: it is a layer that provides higher level abstractions to mask the lower level interface exported by the drivers. For instance it provides the user-space block I/O interface to storage applications above the SPDK platform. As depicted in Figure 1, this component already includes different block devices abstractions (e.g. NVMe bdev, passthrough), and also enables

the integration of third party block devices. This means that in these layers, users of the platform can create their own block devices with their own logic implemented. SPDK also enables users to mount a stack of block devices creating a fully personalized I/O stack. Complementary, SPDK also provides BlobFS, a simple file system, and Blobstore, a persistent and power-fail safe block allocator designed to be used as the local storage system backing a higher level storage service like BlobFS for instance [37].

- **Storage protocols**: contains the accelerated applications implemented upon the SPDK framework to support various different storage protocols like Internet Small Computer System Interface (iSCSI) target for iSCSI service acceleration.
- **App scheduling**: this component provides an application event framework for writing asynchronous, polled-mode, shared-nothing server applications by leveraging its libraries. It is an optional component to have in an SPDK storage stack, however, it is very helpful for external applications that will use SPDK because it handles all the thread and CPU management of the system.

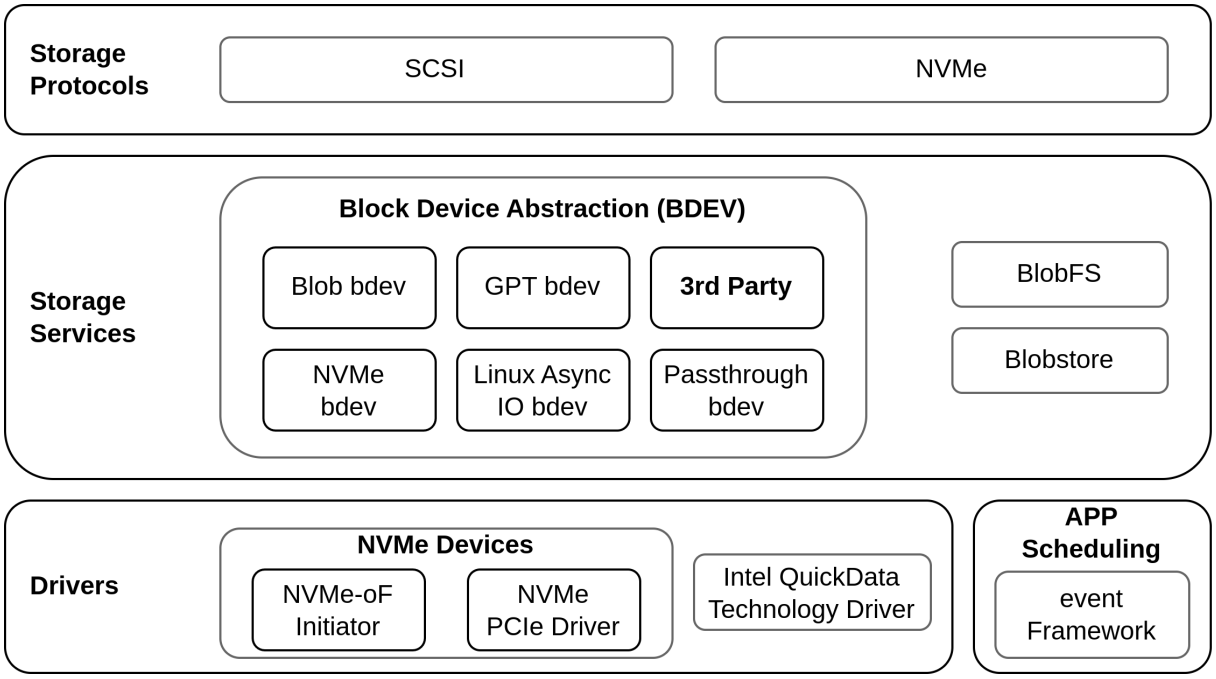



Figure 1: SPDK main components [37].

To understand how all these components work together, it is better to analyze a simple example of an SPDK I/O stack composed of an application, an NVMe block device, an NVMe driver and the NVMe device. Naturally the driver connects to the physical disk and exports a low level interface. To mask this interface, the block device connects with this driver and exports an API more intuitive and easy to use, the SPDK block device API. Using this API, the application connects to the block device creating an I/O flow between the application and the physical storage device. To better understand the difference between this storage

stack and a more traditional kernel one, we are going to analyze the I/O flow of the same application and storage device, but now with a traditional kernel stack as depicted in Figure 2. In the kernel stack, requests pass through several layers like the file system, the page cache and the mapping layer, and all of them have their purpose, but in cases like this where the application is simple and just wants to connect with the storage device directly, all of these steps might be unnecessary. It is also important to mention the context switching element that brings extra latency to the requests. It is obvious that more complex systems will require more than the simple stack of SPDK presented, however, the fact that the framework allows users to build a customized stack accordingly to their needs is a huge advantage when comparing to traditional kernel stacks that have a much more restrict configuration. With SPDK, users are able to create shorter paths for the I/O requests and eliminate the context switching overhead resulting in less latency for I/O requests.

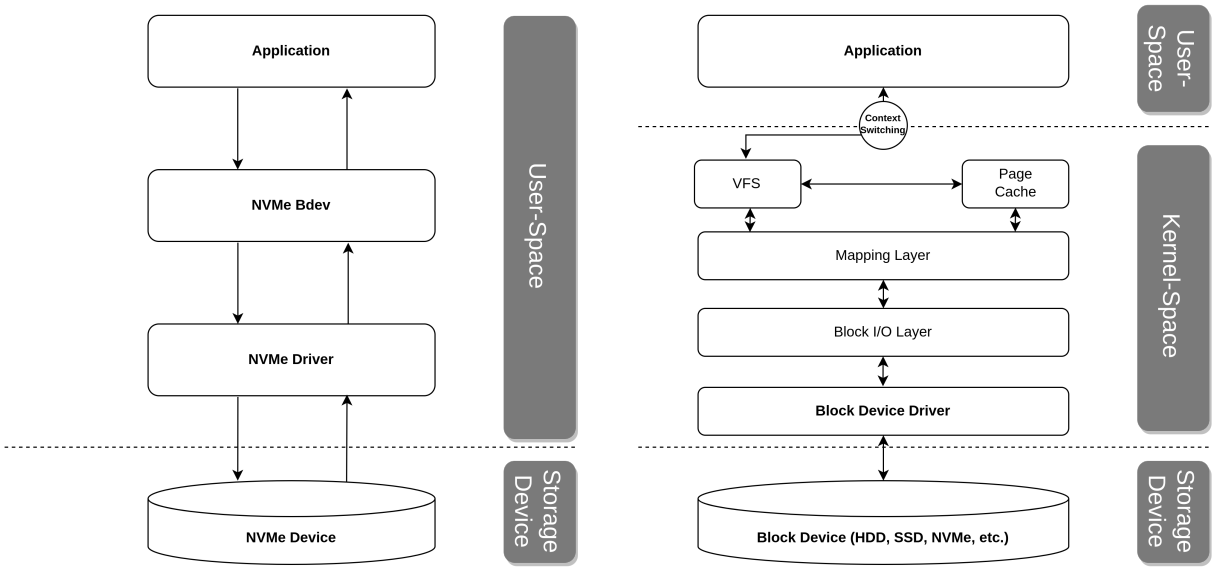



Figure 2: SPDK I/O flow vs traditional I/O flow.

### 2.1.1 Building an SPDK Block Device

The storage services component enables users to build custom block devices, as well as fine-tuned I/O stacks, made of different block devices (each with a different functionality to be applied over I/O requests). Firstly is important to understand that is possible to create two kind of block devices: a **real block device**, that receives I/O requests and passes them to the drivers component, and **virtual block devices**, that route I/O requests to another block device. A canonical example of a virtual block device would be a block device that implements RAID [34]. The process to build them is pretty similar, however, a virtual block device must claim another block device where it will forward I/O requests.

**Creation of the module** To create a new block device module, the first interface that users must interact with is in the file *bdev_module.h* [1]. In this header file there is a macro defined that is used to register a new bdev module. This takes an argument of the type *spdk_bdev_module*, a structure that defines a block device module, where all the initialization functions and information about the module are detailed, as depicted in Table 1.

| Data Fields | Description |
|---|---|
| module_init | Initialization function for the module |
| init_complete | Optional callback for modules that require notification of when the bdev subsystem has completed initialization. |
| fini_start | Optional callback for modules that require notification of when the bdev subsystem is starting the fini process. |
| module_fini | Finish function for the module. |
| config_json | Function called to return a text string representing the module-level JSON RPCs required to regenerate the current configuration. |
| name | Name for the modules being defined. |
| get_ctx_size | Returns the allocation size required for the backend for uses such as local command structs, local SGL, iovecs, or other user context. |
| examine_config | First notification that a bdev should be examined by a virtual bdev module. |
| examine_disk | Second notification that a bdev should be examined by a virtual bdev module. |
| async_init | Denotes if the module_init function may complete asynchronously. |
| async_fini | Denotes if the module_fini function may complete asynchronously. |
| async_fini_start | Denotes if the fini_start function may complete asynchronously. |

Table 1: spdk_bdev_module structure documentation.

**Creation of the block device** After the specification and creation of the module, the block device is created within the module. The block device must implement the interface described in Table 2. This is the most important aspect of the block device, because all of the behavior of the block device is implemented in this functions. All these functions, with the exception of the optional ones, are fundamental to the correct operation of the block device and all the environment that surrounds it. The function *submit_request* is the one that is used to pass I/O requests to the block device and it is where users can define what will happen with the I/O request, how it will be processed, how or when it will be passed to the lower levels and so on. In the *destruct* function, the developer defines what may be important to do when the block device is being discarded and is not needed anymore, this can imply freeing memory or shutting down a piece of hardware etc. [34]. About the *io_type_supported* field, we first need to understand what I/O types are available in the SPDK framework. They are depicted in Table 3. Simple block devices only support *SPDK_BDEV_IO_TYPE_READ* and *SPDK_BDEV_IO_TYPE_WRITE*, however, more complex block devices need more than just write and read operations. For instance the *SPDK_BDEV_IO_TYPE_FLUSH* operation is important for block devices that do any type of caching. With this is possible to understand that to have

[1] SPDK: bdev_module.h file reference. url: https://spdk.io/doc/bdev__module_8h.html

| Data Fields | Description |
|---|---|
| destruct | Function that destroys the backend block device object. |
| submit_request | Function that processes the I/O. |
| io_type_supported | Function that checks if the block device supports a specific I/O type. |
| get_io_channel | Function that gives an I/O channel for the block device for the calling thread. |
| dump_config_json | (Optional) Function that outputs the configuration of the block device to a JSON stream. |
| get_spin_time | (Optional) Function that gives the spin-time per I/O channel in microseconds. |

Table 2: spdk_bdev_fn_table structure documentation [34].

a good functioning of the I/O stack, the block device must provide a function (*io_type_supported*) that indicates the supported I/O types, so that the upper layers know what kind of request must be sent (or not) to the block device. For instance, if a layer wants to send an *SPDK_BDEV_IO_TYPE_WRITE_ZEROES* request and the block device does not support that I/O type but it supports *SPDK_BDEV_IO_TYPE_WRITE*, the upper level can submit a write request with a buffer full of zeroes.

| IO Type | Description |
|---|---|
| SPDK_BDEV_IO_TYPE_READ | Read data from the device. |
| SPDK_BDEV_IO_TYPE_WRITE | Write data to the device. |
| SPDK_BDEV_IO_TYPE_WRITE_ZEROES | Similar to write but has no data buffer, if it did it would be a buffer full of zeroes. |
| SPDK_BDEV_IO_TYPE_UNMAP | Marks a set of blocks as no longer containing valid data. |
| SPDK_BDEV_IO_TYPE_FLUSH | Makes all previously completed writes durable. |
| SPDK_BDEV_IO_TYPE_RESET | Aborts all I/O and returns the device to its initial state. |
| SPDK_BDEV_IO_TYPE_NVME_ADMIN | Mechanism to pass raw NVMe commands. |
| SPDK_BDEV_IO_TYPE_NVME_IO | Mechanism to pass raw NVMe commands. |
| SPDK_BDEV_IO_TYPE_NVME_IO_MD | Mechanism to pass raw NVMe commands. |

Table 3: I/O types in SPDK[34].

The *get_io_channel* function provides each thread a gateway into the block device. I/O channels in SPDK are important to manage thread concurrency and message passing between different layers.

### 2.1.2 Making an Application SPDK-compliant

While SPDK enables building custom storage stacks, it does not follow a standard I/O API. This means that the integration of the toolkit with systems may not be as straight forward as it is with using a traditional file system that uses Portable Operating System Interface (POSIX).

As depicted in Figure 1, it is possible to use storage protocols, like Small Computer System Interface (SCSI) or NVMe, with SPDK, so applications that are currently using this type of storage protocols to perform their storage operations, will not need any drastic changes to their source code. All that is needed to do is to build an SPDK environment below these applications and direct the requests from the application to the SPDK running environment.

Applications that are not compatible with the aforementioned storage protocols, require changes to their source code. An option is to start using the storage protocols mentioned above, another is to build the application to communicate directly to the storage services component of the SPDK framework.

To use directly the storage service component, SPDK offers different options: the **block device layer**, **Blobstore**, or **BlobFS**. The **block device layer** can be used by applications as a block device, which differs from using traditional file systems. To use a block device, the application does not work with files, but instead blocks and the physical mapping to the block device (SPDK offers both ways of sending I/O, with blocks, or simply with memory offsets and sizes). On top of that, applications must be aware that SPDK block device API is asynchronous, so they must be ready to have callback functions and wait for the actual response to arrive. All of the I/O is done with direct memory access buffers, which means that there is no copy of memory during the whole process [2].

**Blobstore** is a persistent, power-fail safe block allocator designed to be used as the local storage system backing a higher level storage service [37]. It presents a different interface than the block device layer, in fact, in an SPDK stack with Blobstore we see that below the Blobstore comes the block device layer, which means that Blobstore acts like an application using the block device layer. A higher level application must understand this new interface that is based on blobs. To understand what blobs are, we have to analyze the hierarchy of storage abstractions implemented by Blobstore:

- **Logical Block:** exposed by the disk itself, which are numbered from 0 to N with N being the number of blocks in the disk.
- **Page:** defined to be a fixed number of logical blocks defined at Blobstore creation time.
- **Cluster:** is a fixed number of pages also defined at creation time.
- **Blob:** is an ordered list of clusters , and are manipulated by the upper level application (they can be created, sized, deleted, etc.).

The interface that Blobstore exports is asynchronous, so in order to use it, an application must be ready to use asynchronism.

**BlobFS** is a very simple file system, however, it does not follow a POSIX API. To use BlobFS an application must be aware that it has a synchronous API and despite of being a file system, it does not have directories, it only has files. As it was mentioned above, BlobFS uses Blobstore to manage the memory blocks, so an SPDK stack with BlobFS must have Blobstore and block device layer in the service components layer.

There already are some systems that are SPDK-compliant, some of them already existed and now have an SPDK version, like RocksDB that has a version that uses BlobFS. Then there are some systems that were originally designed already using an SPDK storage stack like MicroFS [19], a semi-microkernel user-level filesystem that uses the block device layer.

[2] Stated in Block Device Layer Programming Guide. url: `https://spdk.io/doc/bdev_pg.html`

## 2.2 Fault Injection

Fault injection consists of injecting faults in systems and check their reaction to them in order to see if the systems are tolerant to them or if it triggers a failure.

First we need to understand what a fault is and learn about the different types of faults. A fault takes place when a defect appears in a system and it could be dormant or active. It becomes active whenever an error happens. Then, if the error is visible from the outside world, we have a failure [7].

Secondly we need to isolate storage faults. With SPDK being the main focus of this thesis, and knowing that it is a framework used to build storage systems, we need to understand what storage faults there are and how storage systems behave in the presence of such faults.

### 2.2.1 Types of Faults

We want to focus on faults that are related to storage systems like disk failures. These faults are normally divided in three categories: whole disk failures, Latent Section Errors (LSEs) and Undetected Disk Errors (UDEs), with the last two being the most common in the modern storage systems [7, 29].

**Whole disk failures -** these are the failures that occur when the disk does not work as a whole. These can be caused by different problems like: **networking problems** - when a system is communicating with a remote disk and the network is not working resulting in the inability of the system to reach the disk; **hardware damage** - when the disk is physically damaged and is not able to work in a proper manner; **misconfiguration** - when the disk is not properly configured for certain environments, it will not behave as expected.

**LSEs -** refer to the situation where particular sectors on a device become inaccessible. These errors cannot be detected with read or write operations, they can only be detected by the disk drive itself, by its inability to read or write sectors [3]. For example, when the sector is accessed in SCSI and it returns a Medium Error (error that a storage device can experience, which implies that a physical problem was encountered when trying to access the device). These errors can lead to the loss of data, for example, when a group reconstruction is being made in a RAID architecture [32]. The probability of LSEs occurrences differs according to some variables [4]:

- **Storage capacity**: the greater the storage capacity of a disk, the greater the probability of LSEs occurrences.
- **LSEs frequency**: a disk that has some LSEs is more likely to get another than a disk that has none.
- **Location in time and space**: it is more likely to an LSE occur in a certain window of time and nearby storage location of the previous error.

**UDEs -** very different from LSEs, these are impossible to detect by the device driver [3]. Even RAID architectures might not be able to detect them. UDEs are subdivided into Undetected Read Errors (UREs)

and Undetected Write Errors (UWEs). The biggest difference between the two is that UREs cannot change the state of the disk and UWEs can. UREs can retrieve corrupted or stale data but because they cannot change the state of the storage device, UREs are considered transient faults. However, UWEs are persistent errors that can only be detected after a read operation, which retrieves corrupted data [7].

UDEs can have multiple outcomes depending on the operation completeness in near or far off disk tracks:

- **Near-off track UWE**: can create stable data on further reads.
- **Far-off track UWE**: creates stable data and corrupts the written track.
- **Near-off track URE**: can retrieve stale data.
- **Far-off track URE**: retrieves corrupt data.
- **Dropped write operation**: if it never reaches the disk, results in stale data.

For the detection of these errors, storage systems add their own higher-level checksum for each disk block, which is validated on each block read [3]. Then some storage systems use more than this because the simple use of checksum is no sufficient to detect all data corruptions. For example, in [3], their storage system also uses file system-level disk block identity information to detect previously undetectable corruptions. After the techniques of storing metadata regarding the data stored in the disk blocks, storage systems tend to use corruption detection mechanisms like data integrity segments or data scrubbing [3]. There are several classes of data corruption that differ in the way that they are detected and also the way they are generated.

- **Checksum Mismatches** : can be caused by components within the data path, an incomplete write, where only part of a data block is written successfully, or a misdirected write. These are the most common because they can be detected in any simple read.
- **Identity Discrepancies** : detected by a file system read and can be caused by a lost write or a misdirected write where the original disk location is not updated.
- **Parity Inconsistencies** : only detected during data scrubs and occur when a mismatch between the parity computed from data blocks and the parity stored on disk despite the checksums being correct. These can be caused by lost or misdirected writes, in memory corruptions, processor miscalculations and software bugs.

All of these are explored more deeply in [3]. There it were found 400000 checksum mismatches in 1,53 million disk drives during 41 months. With these numbers we can see that data corruption is a key concern that must be considered in today's storage systems.

### 2.2.2 Hardware vs Software fault injection

Now that we know what storage system faults are and the different type of faults there are, it is necessary to understand how these faults can be injected in storage systems.

There are two different approaches to inject faults, hardware and software based. Choosing between hardware and software fault injection depends on the type of faults that one is interested in and the effort required to create them [14]. Despite of this thesis being focused in software fault injection, we will discuss both approaches to see why the software approach is the best one for our case.

**Hardware fault injection -** is done by using additional hardware to introduce faults into the targeted hardware. Hardware-implemented fault injections are divided into two categories [14]:

- **With contact**: where the injector uses direct contact with the target system. These can be done using the two following strategies:

  1. *Active probs*: this consists in adding an electric current via the probes attached to the pins, altering their electric currents. This should be carried out very carefully because an inordinate amount of current can damage the hardware.
  2. *Socket insertion*: this one consists of inserting a socket between the target hardware and its circuit board. The socket inserts logic faults into the hardware. The pin signals can be inverted ANDed, or ORed with adjacent pin signals or even with previous signals on the same pin.

- **Without contact**: here there is no direct contact with the pins, this is done by creating heavy-ion radiation making the ions pass through the depletion region of the device generating current. It can also be done by placing the hardware near an electromagnetic field.

**Software fault injection -** consists in injecting faults in the software. This way of injecting faults is now the more commonly used because it is not as expensive and is more flexible, enabling the injection of faults in different layers of a software system (application, operating system, etc). This is quite difficult to achieve with hardware implemented fault injection.

Looking now to how one can inject software faults there are two different techniques:

- **Compile-time**: this technique consists in modifying the source code of the program, altering the instruction of the system before it is compiled. When the program runs, hard-coded faults will be injected and there is no additional runtime required [14].

- **Runtime**: this technique consists in the injection of faults during the system runtime. There are three main mechanisms to do it:

1. *Time-out*: consists in specifying a timer that, when it expires, triggers an injection of a fault. This mechanism can have unexpected effects and does not affect the workload of the program. With this approach it can be a bit more difficult to catch faults, and when faults are caught it is impossible to know where the bug is in the program.
2. *Exception/trap*: similar to the previous one, but instead of the fault being injected in a certain time, it is injected upon a certain event. This makes the effect more predictable since it is known the instruction that has already been executed and the ones that have not. In that sense, it is easier to understand where the possible bugs are in the code because we know that the failures caught were generated by injecting faults in a specific event.
3. *Code insertion*: code insertion does not modify instructions, it only adds some more, with the objective of injecting faults, and it is done during the program runtime. This approach provides more information about where the possible bugs are in the code because one knows exactly where fault injection is being performed.

## 2.3 Related Work

In this section we discuss related work on fault injection in storage systems. As it was already referenced in Subsection 2.2.2, hardware fault injection is too expensive and can cause permanent damage to the target hardware. On top of that, it is more challenging to reproduce more specific types of faults. However, there are some tools that do it (e.g, MARS [18], FTMP [11] or MESSALINE [1]), but this topic is out of scope of this thesis.

Regarding software fault-injection, there are several tools that can inject faults in the most different places like storage, memory, network or even CPU. Some tools are able to inject in many of them (e.g, Fault and Error Automatic Real-Time Injection (FERRARI) [16], Fault Tolerance And Performance Evaluator (FTAPE) [36]). These are more complex and generic, being able to affect many aspects of the whole system. However, the way that they inject faults in different places can differ, for example, FERRARI injects faults by corrupting data in general, but FTAPE injects faults in the CPU by corrupting its registers. The fact that both of these tools use kernel shows us that they are not usable in the SPDK applications use-case.

Now we will be looking to different approaches of fault simulations and inspect different tools that follow them. Those approaches are **system crash simulation**, **system call fault injection** and **content-based fault injection**.

### 2.3.1 System crash Simulation

Some fault injection tools test the storage system by simulating a crash in the system. This is what characterizes system crash simulation, a fault injection methodology that is based in simulating crash situations, like power loss or network failure.

One of the main references in system crash simulation is **Chaos-Engineering** [5]. It is not a fault injection tool, it is more like a methodology of fault injection that some tools follow to do system crash simulation. This methodology consists in making as many random experiments possible to the systems, blindly, and crash them along the way. Examples of such systems are Chaos Monkey [6], which tests distributed storage systems and randomly turns off one of the instances and analyzes the behavior of the system as a whole, and Chaos Kong [5], that takes this approach to a higher scale, turning down entire regions of nodes, which means that a higher number os instances is turned down at once.

The biggest advantage of this approach is the fact that crashing instances or killing random process is simple. Automating that to be done thousands of times can bring up many hidden failures without using a more complex tool.

One downside of this strategy is that it may give the developers a false sense of resilience and robustness. Many failures are only found if a certain combination of events happens and that can not be controlled using random testing due to its nondeterministic nature. Chaos-Engineering uses a nondeterministic approach because the state of the system is not evaluated using this methodology, taking the same steps does not mean that we are simulating the same events. Consequently, possibly detected failures may not be easily reproducible. Despite of the disadvantages, several companies (e.g., Netflix) use Chaos Engineering to test their systems.

With the random crashing being an issue, one tool that deterministically simulates crashes upon specific workloads, is $B^3$ Crash Testing [23]. It is a file system crash consistency tester that follows a black-box approach. The black-box approach consists in only evaluating the inputs and outputs of a program, this way it does not need to know the code of the target software. This brings the advantage of being compatible with different systems, in this case, file systems. It is only needed to evaluate the expected output and the real one and compare the two. If they match, then the program does not have a bug when the injected crash occurs. However, if they do not match, the program has a bug. We rapidly acknowledge a big disadvantage in this approach that is, when a bug occurs, it is challenging to identify in the source code of the targeted file system which part was responsible for that particular bug.

$B^3$ was designed to solve two main problems, the lack of an automated infrastructure to systematically test file systems, and the large number of workloads that need to be tested. For that, this tool has two different components, each designed to tackle one of the aforementioned problems:

- **CrashMonkey**: created to solve the first of the referenced problems, *CrashMonkey* is a framework that simulates faults, normally crash faults simulating system failures like power loss, in different places of a given workload. Then it tests the correctness of the file system after each injection.

- **Automatic Crash Explorer (ACE)**: created to solve the second of the referenced problems, ACE is a tool that generates workloads that satisfy certain bounds specified by the developer.

Looking now more deeply to both components of $B^3$, the *CrashMonkey* is the fault injection component of the tool. It works by following three main phases. The first one consists in collecting information about

the file system operations and I/O requests made during the workload.

In the second phase, the framework replays all the I/O requests until it reaches a persistence point and creates a crash state. A persistence point consists in the occurrence of a synchronization (e.g.,*fsync*). The crash state is the representation of the storage's state if the system had crashed after the persistence point. Also in this phase, *CrashMonkey* captures a reference file system image, by safely unmounting it so the file system completes any pending operations, these images are referred to as *oracle*. The *oracle* represents the expected state of the file system after a crash.

Lastly, the third phase is where a comparison of the oracle and the crash state, after the recovery mechanisms are done, is made by CrashMonkey's AutoChecker. If there are no bugs, both states should be similar. If the persisted files and directories differ in both states, we are in the presence of a potential bug.

About the ACE component, this is the one where the workloads are generated according to some bounds specified by the user. ACE has two sub-components [23]:

- **Workload synthesizer:** generates workloads according to the bounds defined by the user. These workloads come in a high-level language.
- **CrashMonkey Adapter:** converts the workloads that come in a high-level language so that CrashMonkey can work with them.

Looking now to how ACE generates the workloads, it is done following 4 phases. The first consists in the selection of the operations according to the user specifications. If the user specified a sequence of $n$ operations and also specifies $k$ different operations for ACE to work with, then in this phase we wil have $k^n$ different workload skeletons. For example, if we specify 3 operations to build sequences of 2 operations then there will be $3^2 = 9$ skeletons generated in phase one.

In phase two the selection of parameters for each phase one skeleton takes place. By default, two files and two sub-directories are used, which can result in multiple workloads for each skeleton. ACE has a semantics knowledge of file system operations, so if two operations lead to the same result, then one of them is discarded. For example, the operation of linking two files has the same outcome no matter the arguments order. The operation $link(f1, f2)$ has the same result than $link(f2, f1)$. This way, phase two can reduce the number of workloads for each skeleton that come from phase one, which is good for performance purposes. Another feature in this phase is the fact that, according to the studies made in [23], when data operations overlap, there are more occurrence of crash-consistency bugs, so the selection of parameters tend to create situations where data operations overlap so it can generate a larger number of bugs.

Phase three consists in the addition of persistence points. Once again, the number of workloads can grow depending on where the persistence operations will be added. The way that the files are persisted may vary, for example, ACE has the option of persisting the files after every operation. The only insurance is that every file is persisted at the end of each workload.

The final phase is where the dependent operations are added, for example, the creation of the directories or even the files that are being manipulated by the already specified operations. It is also in this phase where the *CrashMonkey Adapter* is used, transforming every resultant workload in the language of the *CrashMonkey* framework.

### 2.3.2 System Call level Fault Injection

Another case study is to analyze how a system reacts to faults injected in certain system calls. This is a very important issue that storage systems need to address properly. The *fsync* system call is crucial to ensure that data was in fact persisted, however, due to performance reasons, storage systems tend to streamline its usage and that can generate data loss problems.

CuttleFS [27] is a tool that has the objective of finding this type of problems. The tool uses its own page cache and has total control over the write operations to disk. The cache is built in user-space using in-memory buffers, allowing to mark the different pages as dirty or clean and keep the current state of files and reverse them to their previous state. This way it can simulate the real file systems scenarios of *fsync* failures. On top of that, CuttleFS can be run in two modes: trace mode, where it tracks the disk writes and identifies which blocks are flushed to disk; and fail mode where it allows to fail the $i^{th}$ write to a sector or block of a certain file.

To test the applications, CuttleFS must be run in trace mode and then, after analyzing the information provided by the trace run, it has to be run in fail mode, choosing the requests that should fail to catch a potential bug.

The big disadvantage of this tool is the fact that it uses FUSE which implies the use of kernel, making it impossible to use with kernel-bypass environments.

### 2.3.3 Content-based Fault Injection

Content-based fault injection consists in injecting faults in the content of I/O requests. This can be done at any level in the stack where I/O requests pass, and can generate UDEs. And just like we have seen above, UDEs are very recurrent and for that is important to be able to emulate these kinds of faults to see how the storage systems behave in the presence of such faults.

There are some tools that can simulate these faults. One of them is AchillesBench [7], a software based fault injector, that injects real faults, like corrupting reads and writes by changing some bits in the content of the operations, simulating UDEs. It also has the ability of simulating LSEs by returning the application a medium error when it tries to access a certain memory block. This is all possible because, this framework intercepts the requests at a lower level and injects faults in them. To do this, this tool uses *BDUS* [10], a framework that enables the development of block device drivers in user space, to build a user-level layer where the requests can be intercepted before reaching the block device. The original kernel

layer continues to exist, but when the requests reach the kernel these are redirected to the user-level layer that applies its functionalities and the request is sent back to the kernel.

Looking to the architecture of the system, this is composed of two main modules. First a benchmark, where the workload is generated and sent to the application. Here is possible to configure the type of faults the user wants to inject, when to inject them, if the fault is persistent or transient and even configure the block that will be affected by the fault. This was achieved by extending an already existing benchmark called *DEDISbench* [26].

The second module is a fault injector. It is built over *BDUS*'s virtual block device and is separated in several components:

- **Configuration Handler**: receives the information about the fault configuration from the benchmark.
- **Interceptor**: intercepts the I/O operations at kernel space, sending them to user-space where they will be processed synchronously.
- **Fault Handler**: merges the work made by the other two modules, being responsible for checking all the requests and injecting faults in them according to the information given by the configurations.

Looking to how this prototype inject faults and the kinds of faults that it supports, currently these are the ones that the platform provides:

- **Bit flip**: consists in flipping a bit in the content of a block before it is written in the underlying block device or read by the application. This will generate a so called corruption creating an UDEs.
- **Medium error**: consists in making a certain block inaccessible, injecting a LSEs.
- **Slow disk**: consists in delaying an I/O operation by a certain amount of time.

The tool enables the configuration of which faults to inject by providing a configuration file that users can edit to define the politics of fault injection in each workload.

Another fault injection tool that is oriented to corrupt content is called *CORDS* [12]. It is a fault injection framework that was built to test how distributed systems react to local file systems faults.

*CORDS* framework is composed of two main components:

- ***errfs***: a user-level faulty file system that is responsible to implement the faults. This file system was build using FUSE.
- ***errbench***: a workload suite responsible for the creation of two workloads (read data and insert or update data) per system that are able to exercise all the code paths that interact with the local file system.

About the types of faults that the framework supports, it is able to corrupt blocks by inserting zeros or random bytes in them. This is done by *errfs* in read operations (e.g., read, pread) by changing the marked block content before it is sent to the application. The tool also supports bit-flip, similar to AchillesBench, however, this is done with more precision, because the bit that is corrupted must be properly marked, and this is only supported for read operations. This is all of the UDEs that the framework supports. In the field of LSEs, this tool is able to inject those both in writes and reads. Note that this tool could also be included in the previous Subsection 2.3.2, because all the faults are injected in system calls.

*CORDS* strategy to inject faults is simple, it only injects one single fault in one node of the distributed storage system. The objective is to see how much of an impact one simple error can cause in an entire storage infrastructure. In other words, if the recovery and redundancy systems are able to overcome these errors.

To do this, it is needed to analyze both local and global behavior. Local being the behavior of the faulty node and global, the behavior of the all system. The local analysis is done by inspecting the local file system logs and outputs. The global analysis is done by inspecting if the distributed protocols are correct. For instance making sure if all data that was committed is not lost or if the read operations are returning any errors.

Despite of being a good tool for many distributed systems, the fact that uses FUSE in the construction of their main fault injection component, implies the use of kernel, making it not viable to use in SPDK applications. Another downside to this tool is the fact that for each test it only injects one fault. Sometimes a sequence of faults or the occurrence of many faults is what can make some storage systems break, and for that scenarios this tool is not that strong.

## 2.4 Discussion

After analyzing the SPDK framework and all of its functionalities in Section 2.1, we can conclude that it enables system designers to build a fully customized user-space storage stack. It gives better performance than traditional storage stacks, due to the fact that it bypasses the kernel and avoids context switching, making the framework very enticing to use. However, to ensure that storage systems built upon an SPDK environment are fault tolerant is important and for that matter, we need a tool that successfully assesses this.

Regarding how fault injection is achieved in existing storage systems (2.2), we can conclude that hardware fault injection is not an option to achieve our goals due to the fact that it is very costly in terms of hardware damage, and it is challenging to inject faults with precision and determinism (i.e., where we want and when we want). On the other hand, software fault injection is very flexible, allows one to inject faults in any place in the system's stack and is way less costly than hardware fault injection, and for that reason it is the fault injection approach followed in this thesis.

After analyzing the related work, it is possible to understand that there are many possible approaches

for storage fault injection (system crash simulation, system call level fault injection, content-based fault injection). However, there is no tool that does the work for storage stacks built with SPDK. For instance, AchillesBench uses *BDUS* that requires the use of kernel, making it impossible to test SPDK systems. Chaos Engineering is a possible approach, in terms of interface. However, it presents two problems: first, it only enables random crashes, limiting the domain of faults to inject (e.g., data corruption); second, it does not enable deterministic and reproducible fault injection.

Looking to AchillesBench, it does not allow to test kernel-bypass storage systems. However, it has some good ideas that can be taken in consideration. The idea of using a user-space layer on top of the block device in order to intercept and inject faults in the I/O requests is a very interesting concept that can easily be adapted to an SPDK environment, given the fact that the platform permits the construction of virtual block devices. Another good idea that we can take from this tool is the use of configuration files and I/O request identifiers to inject the fault that we want in the request that we want.

About the types of faults, the objective of this thesis is to inject faults in the content of the I/O requests to emulate UDEs, and we can see that the bit flip is present in the current literature because of its simpleness and effectiveness [12, 7]. This shows that it would be interesting to have this methodology in a tool that evaluates faults in systems built with SPDK. We can also see that there are many different definitions of the amount of faults to inject and when to inject them. CuttleFS, for instance, has a very particular one where there is possible to inject in the $i^{th}$ request of a certain file. Then there is the others like CORDS or AchillesBench that can inject faults in more than one request. This grouping of different ideas can result in other strategies like inject faults in a defined interval of requests (e.g, inject in the fifth, the tenth, the fifteenth...).

Another key point that we can take from this study is to inject faults in a deterministic way. This facilitates the reproducibility of bugs which is very important if we want to see what patterns of faults are generating bugs, allowing us to find where the problems might be.

Concluding, there are many fault injection tools that can do the job in many different ways, however, nothing is adapted to be used in an SPDK environment. So the purpose of this thesis is to adapt some of the ideas present in the related work, in order to be possible to evaluate the correctness of storage systems that use SPDK.

# 3

# FISPDK

The main objective of this thesis is to be able to test the fault tolerance of kernel-bypass storage systems built with SPDK. To achieve this goal, we designed FISPDK, a fault injection tool for kernel-bypass storage systems. FISPDK allows to test the resilience of the storage systems that are built with SPDK, using I/O differentiation to inject faults deterministically. It also allows the faults caught to be reproducible.

In this chapter we will discuss how FISPDK is designed and its core features. Then the implementation decisions will be presented, including how to make existing applications and virtual block devices FISPDK-compliant.

## 3.1 Architecture

FISPDK is designed as an extension for SPDK and offers a way to test the resilience of the storage systems. FISPDK allows that by performing deterministic fault injection at kernel-bypass storage systems, at the block device level. In order to achieve deterministic fault injection, FISPDK allows I/O differentiation, more specifically relative to write and read operations. I/O differentiation is important to give extra information about a request that normally does not reach the lower levels of the storage stack. To be able to do that, FISPDK enables context propagation (i.e., pass application level information to the lower levels of the I/O stack) [21, 20]. Figure 3 depicts a general view of a system that uses FISPDK to test the application resilience.

FISPDK fits in the block device layer of SPDK, where the **context propagation** is achieved with an extension of the original SPDK's block device API. The interception of the I/O requests is made with a new block device module, the **faulty block device**, where the requests are injected with the faults according with the request differentiation. This module has a sub component to help it achieve fault injection: a **faults library**, that provides a way to inject different types of faults.

The next sections discuss in more detail each component of FISPDK.

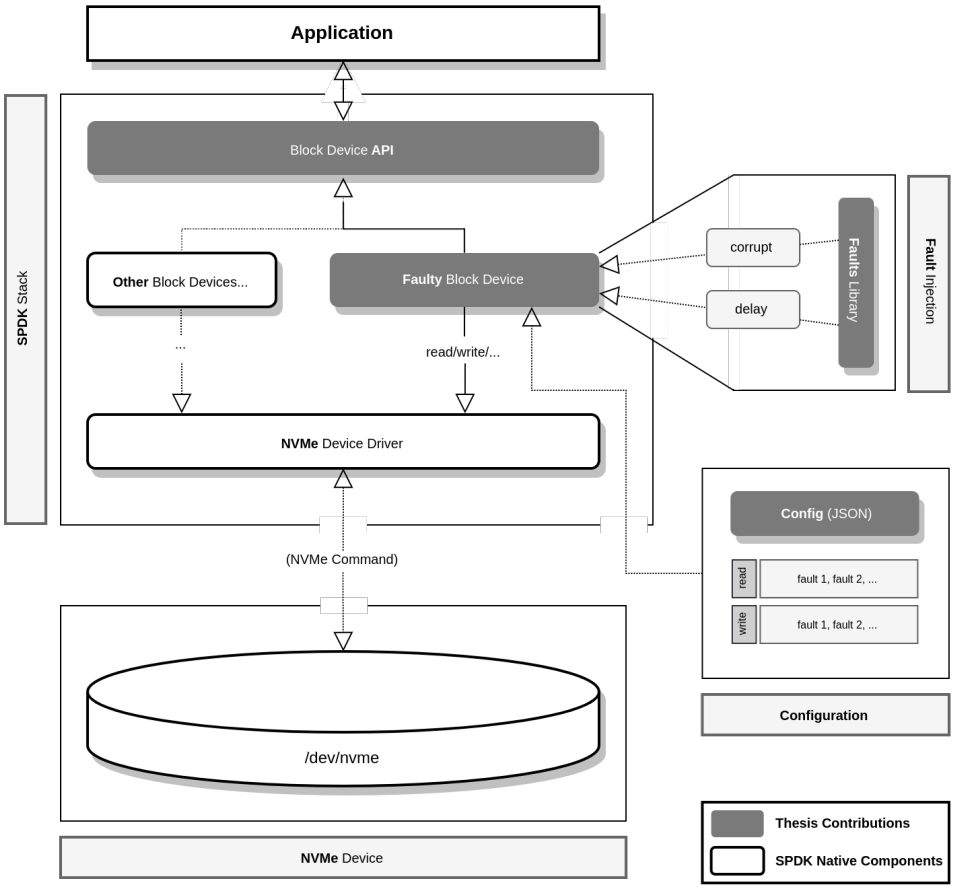



Figure 3: High-level architecture of FISPDK.

### 3.1.1 Context Propagation and Block Device API

The original block device API provided by SPDK allows performing read and write operation in a given place of the disk with a given content size. However, it would be better if it allowed to pass extra information to the block device layer in order to differentiate the I/O requests, because I/O differentiation is fundamental to deterministically inject faults in SPDK devices. For example, let us consider a scenario where a user wants to inject faults in log files that typically have the extension *.log*. To do this, we need to know the name of the file. The file system may have this information, but for the block device layer, write or read operations are related to a position in the disk where it should write to or read content from. With I/O differentiation, we would be able to propagate the name of the file to the block device enabling a differentiated treatment of each request. To do this we need to propagate context throughout the storage layers. Context propagation consists in passing extra information (context) about a request throughout the layers that the request passes through. These contexts, can be used for example, to capture causal relationships between events [20]. Context propagation is used both in research and practice [9, 17, 24, 25, 21]. Looking back to the previous example, if the context that is propagated contains the name of the file to which that I/O request is related to, the block device layer will have access to the file name.

To propagate context in the block device layer of SPDK, we had to make some changes to the framework. An I/O request in SPDK's block device layer is represented by a structure called `spdk_bdev_io` [1] so the strategy is to add a field in that structure that stores the context that the user wants to propagate. However, this is not enough, because this structure is not accessible to the applications. As such, we created a new interface so the application can indicate the context that it wants to propagate throughout the block device layer. This led to changes in the original Block Device API [2] so that applications have the possibility to propagate context in a practical way. We provide more implementation details in Section 3.2.

With this solution, if a virtual block device is above another block device that needs the context to work properly, it needs to propagate the context to the lower layers. To do so, these block devices must also use the new extended version of the block device API every time that there is any context to propagate. In the case of FISPDK, only the faulty block device will give usage to the context. However, the block device layer can have more than one virtual block device that needs the context to work properly. As such, even if the context is used by a given virtual block device, it also has to propagate the context to ensure that any lower layer that might need the context will have access to it.

### 3.1.2 Faults Library

Now that we have a way to propagate context to the block device layer allowing block devices to perform differentiated processing of I/O requests, it is needed to have a way of injecting faults in those differentiated requests. For that reason, we designed an independent library, based in the literature that was explored in Section 2.3, that provides to block devices a way to inject UDEs faults in I/O requests. In particular, it enables corrupting the data of I/O requests, and injecting delay to simulate slow storage mediums (as [7]).

Regarding data corruption faults, the library provides four mechanisms for corrupting buffers:

- **Random Bit Flip:** memory bit flips are errors that can be costly in terms of system failures that they cause and the repair costs associated with them [33]. So the library is able to randomly pick one bit of a given buffer and flip it, creating a corrupted state.

- **Custom Bit Flip:** following the same philosophy of the previous fault, but instead of being a randomly picking a bit to be flipped, the user can specify which bit to flip. This can be handy in certain situations like flipping the signal of an integer for instance. Despite of the corruption in the real world being random, the library gives the option to see the reaction of the system if a certain bit is flipped.

- **All Zeros:** like it was stated in [12], in the ext family of file systems or in the XFS, it is possible to read zeros when there is a misdirected or lost write. For that matter we simulate this scenario by zeroing all the bits in the buffer of an I/O request.

[1] SPDK: Spdk_bdev_io struct reference. url: `https://spdk.io/doc/structspdk__bdev__io.html`
[2] SPDK: Bdev.h file reference. url: `https://spdk.io/doc/bdev_8h.html`

- **All Ones:** in the previous scenario, [12] also states that is possible to find junk instead of zeros, so the library also gives the option of turning all the bits of the buffer to 1, to simulate junk.

About the slow disk, it is important to emulate because some multi-threaded applications may not be protected against the possibility of one request taking longer time than another and this can cause some inconsistent state. What the library does is to inject some delay in the request. More details about how this is done are present in the implementation section (3.2).

With context propagation and the faults library, a block device is now able to inject faults according to the context that is propagated with each request.

### 3.1.3 Faulty Block Device

The faulty block device is a virtual block device, built in SPDK, that can be used to inject faults in differentiated I/O requests, more specifically in write and read operations. It allows users to create custom fault injection policies, and by leveraging from the data corruption and delay injection mechanisms provided by the faults library. Figure 4 depicts the key components of the faulty block device and the flow of an I/O request (more specifically a write operation).

As we can see, the faulty block device is divided in 4 components:

- **I/O Receiver**: is the component that receives the I/O requests from the upper layers.
- **Context Checker**: is the component that checks the context that characterizes the I/O requests that arrive to the I/O receiver.
- **Configuration**: is the component where the user configures the fault injection tool, where it is possible to define how the fault injector must inject the faults, when and where.
- **Fault Injector**: is the component responsible for injecting faults in I/O requests, by resorting to the faults library. According to the information that comes from the context checker and from the configuration, the fault injector asks the faults library to inject the fault.

We will now discuss in more detail each component.

#### 3.1.3.1 Configuration

The configuration is useful for users to define which faults they want to inject, within the options given by the faults library, and when to inject. The user can define when a fault injection will occur by defining which requests will be injected with a fault by indicating what the faulty block device must do to requests that propagate a certain context. The block device will only inject faults when it receives I/O requests that bring contexts matching the ones configured by the user. For instance, if a file system is being tested and it propagates the file name as context, the user can configure the faulty block device to inject faults in

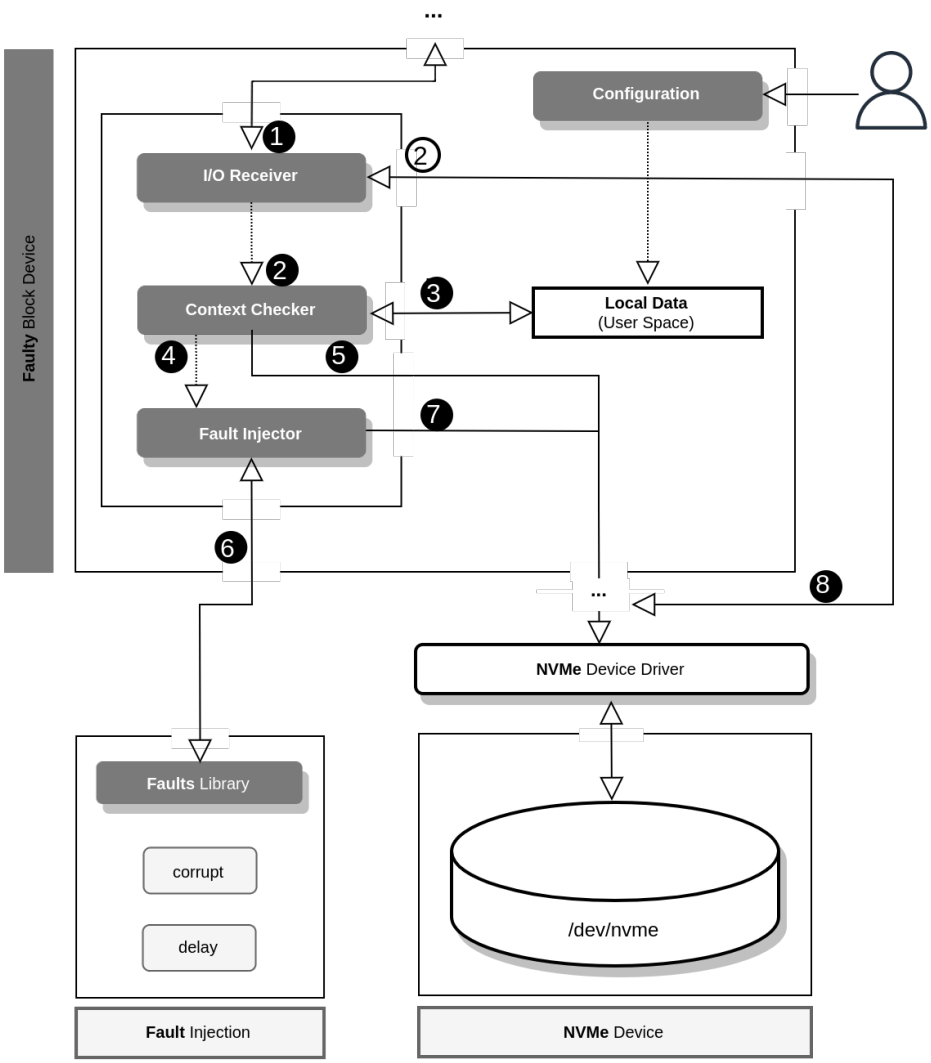



Figure 4: Faulty Block Device Architecture and Flow of a Write Request.

requests related to a certain file. Besides that, the user can define how often he wants the block device to inject faults regarding a specific context. For that, there are three different frequencies:

- **ALL:** the block device will inject the specified fault in every request regarding that file.
- **ALL_AFTER:** the block device will inject the specified fault in all requests after the first *x* requests, with *x* being a variable that also is configured by the user.
- **INTERVAL:** the block device will inject the specified fault in all requests that are multiple of *x*.

Because the fault injection tool is able to inject faults in write and read operations, the configuration module allows the user to inject different faults in the same file, one for write operations and another for read operations. The tool also allows to inject faults only on writes or only on reads with a certain context

(e.g., file name). This gives flexibility and fine granularity to the framework on how to inject faults, which is good to evaluate specific behaviors of certain operations.

### 3.1.3.2 I/O Receiver and Context Checker

The **I/O Receiver**, is a common component in all block devices. It consists in the reception of the I/O requests that come from the upper layers (Figure 4.❶), and their further processing according to the logic of that block device. For each I/O request, it validates its type (read or write), and treats it accordingly. Specifically, Figures 4 and 5 depict how the faulty block device handles write and read requests, respectively.

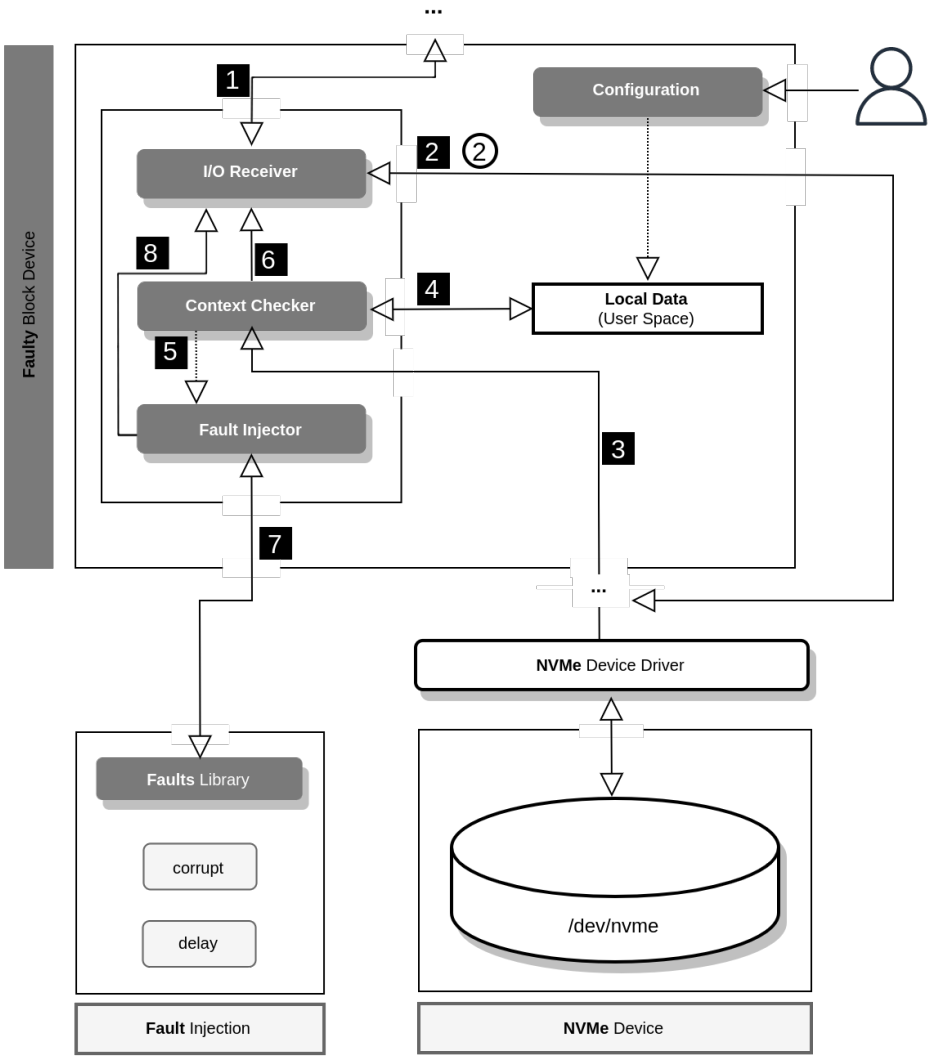



Figure 5: Faulty Block Device Architecture and Flow of a Read Request.

**Write** If the request is a write, the tool passes the request, that in the SPDK's block device layer is represented by the structure spdk_bdev_io [3], to the context checker (Figure 4.❷). Otherwise, it passes the request to the lower layers (Figure 4.②) just like a passthrough. The **context checker** analyzes the `spdk_bdev_io` structure to check if there is any context regarding the fault injection tool. The context checker expects to find a file name that represents the file to which that request is related. After that, the context checker accesses the local memory of the faulty block device (Figure 4.❸), where the information from the configuration is stored, and checks if there are any fault to inject in that request. If that request does not have any fault to be injected, then it is forwarded to the lower layers of the SPDK stack (Figure 4.❺), otherwise it is passed to the fault injector (Figure 4.❹).

**Read** If the request is a read, the context checker will receive the request from the lower levels of the SPDK storage stack, on its way back (Figure 5.3), because it is needed to have the content already read from the disk in order to perform content corruption. Then it does the same thing that it does for a write request, it analyzes if there is a context and checks if there is any fault to be injected or not. If there is, it passes the requests to the fault injector (Figure 5.5), but if the request does not require fault injection it forwards the request back to the upper layers (Figure 5.6) instead of sending it to the lower layers because now the request is flowing in the opposite direction. Regarding the slow disk fault injection, it could be injected before the request reached the disk, however, it would have the same effect, so to simplify the logic, slow disk fault injection follows the same process order as the content corruption.

#### 3.1.3.3 Fault Injector

The fault injector is responsible to inject the faults. If the request is a **write**, the faults library is called (Figure 4.❻) and injects the fault that was specified. After that the request is forwarded to the lower levels (Figure 4.❼).

In case it is a **read request**, the fault injector behaves the same way as for the write request (Figure 5.7), but once again, because the read request is flowing in the opposite direction, it forwards the request to the upper layers (Figure 5.8).

## 3.2 Implementation

In this section, we discuss all the implementation decisions regarding FISPDK and how to adapt an SPDK environment in order to use our fault injection tool.

FISPDK was implemented on top of SPDK, more specifically the version (22.01) of the framework. It was implemented in C and it required a total of 1748 new Lines of Code (LoC) and the modification of

[3]SPDK: Spdk_bdev_io struct reference. url: `https://spdk.io/doc/structspdk__bdev__io.html`

5 LoC which will be explained further down the discussion. The implementation of the whole framework counted with the help of two external libraries, namely the **GLIB** [4] and **JSON-C** [15].

### 3.2.1 Context Propagation and Block Device API

As discussed in Section 3.1.1, it was important to have a way to provide additional context to the I/O requests. After studying how I/O requests are implemented in the block device layer of SPDK, it was decided that the data structure `spdk_bdev_io`, which is the one that represents an I/O request in this layer, was the best option to hold the context that we wanted to add. An object of this structure can only be used by one block device in the stack. This means that if a block device layer is composed of *x* virtual block devices, then there will be *x* different objects of `spdk_bdev_io`. This happens because the virtual block devices also use the block device API to communicate with each other, passing the elements present in the structure to the functions instead of propagating the same structure throughout the stack.

With this fact in mind, it was imperative to extend the block device API so we could have functions that can create instances of `spdk_bdev_io` with the additional context that we want to propagate.

#### 3.2.1.1 I/O data structure

The context field was added to the data type `spdk_bdev_io` which implied the modification of two source code files of SPDK: the `bdev_module.h` [5] file, where the data structure is defined and a field called *flag* was added to it, and the `bdev.c` [6] file, where the initialization function of the data structure is implemented, where the only change was defining the field `flag` as NULL. The `flag` is of the `void*` type, which enables defining contexts in a generic way. In the initialization function we started the field as NULL to not interfere with the regular behavior of the SPDK framework and the field will only be allocated and used when the new created context propagation functions are used.

#### 3.2.1.2 Read and Write Requests

An `spdk_bdev_io` instance is only used by one virtual block device in the stack, and new instances are created when block devices communicate with each other via the bdev API. To propagate the context, it was needed to extend the API with new functions that are able to create instances of `spdk_bdev_io` with the context in it. We decided to create new functions instead of modifying the original ones as the changes would imply breaking existing function signatures, making it incompatible with every software that uses the original API.

In Table 4 we can see the list of functions added to the file `bdev.c` of SPDK, where the original block device API is implemented, for enabling context propagation in write and read requests.

[4] Glib-2.0. https://docs.gtk.org/glib/

[5] bdev_module.h source code: https://github.com/spdk/spdk/blob/master/include/spdk/bdev_module.h

[6] bdev.c source code: https://github.com/spdk/spdk/blob/master/lib/bdev/bdev.c

The new interface allows context propagation in write operations with the method `spdk_bdev_write_flag` and, for scatter-gather described buffers, the `spdk_bdev_writev_flag`. For read requests we followed an identical strategy. These implementations required some extra lines of code for auxiliary functions for

| Function | Description | LOC |
|---|---|---|
| *spdk_bdev_write_flag* | Submission of a write request with context | 14 |
| *spdk_bdev_writev_flag* | Submission of a write request with context where the buffer can be described in a scatter-gather list | 15 |
| *spdk_bdev_read_flag* | Submission of a read request with context | 14 |
| *spdk_bdev_readv_flag* | Submission of a read request with context where the buffer can be described in a scatter-gather list | 36 |

Table 4: Write and read functions added to `bdev.c`.

both write and read operations. Specifically, to enable context propagation in read and write requests, we added 152 and 164 LoC respectively, which is minimal when comparing to the original SPDK framework (approximately 2.1M LoC).

### 3.2.2 Faults Library

The faults library is a C library that is able to create and inject faults (3.1.2), namely data corruption and slow disk.

#### Data Corruption

For data corruption, the library provides the `corrupt_buffer` method as follows:

```
int corrupt_buffer (void *buffer, size_t size, BufferCorruptionPattern
                    corruptionPattern, int customOffset, int customIndex);
```

Looking into the data types of the fields, is important to point out that the BufferCorruptionPattern data type is an enumeration defined in the library for the users to specify which type of corruption they want to use. For **All Zeros** and **All Ones** corruption patterns, it was used the `memset` function with the integer values 0 and 255 respectively.

For the **Bit Flip** corruption type, the only thing that is done is to flip a random bit from a random byte (in the case of random bit flip) or to flip a specified bit from a specified byte (in the case of the custom bit flip).

#### Slow Disk

Regarding the **Slow Disk** fault, it was created the function `operation_delay` with the following signature:

```
int operation_delay (double time);
```

The field *time* is the amount of time in seconds that the user wants to delay the operation.

### 3.2.3 Faulty Block Device

The faulty block device is based on SPDK's passthrough, as it will directly forward to other layers requests that are neither read or write, or when the context is not propagated.

#### 3.2.3.1 Configuration

For the configuration components, it was decided to use a configuration file just like in [7]. For the file format it was chosen the JavaScript Object Notation (JSON), because it is self-describing, compact and easy to read and understand. The configuration file is composed of two lists, one list is dedicated to the write requests and the other for the read requests. Each object of a list defines the file name, the fault to inject and the frequency as explained above. This means that if a user wants a file to be injected with faults not only in the write requests but also in the read requests, he needs to indicate the file name in both lists. After parsing the configuration file, it is needed to store the information parsed and for that we used two hash tables, one for the writes faults and another for the reads. For the hash table implementation we used the **GLIB** implementation, the `GHashTable`[7]. Annex I exemplifies the organization of a configuration file for the following example.

Let us consider that a user wants to simulate UDEs in two files: the `focal.img` and `bionic.img`. The faults injected must be of type `CORRUPT_CONTENT` which will inject content corruption. Regarding the `focal.img` file, the user wants that all of the write requests to be injected with a bit flip in the first bit of the first byte in the content of the request, however, he does not want to inject faults in read operations of this file. For the `bionic.img` file, the user wants that both read and write operations to be injected with faults. In the case of the writes he wants the first 3 to work properly, but all the requests following the third must have all the content zeroed, simulating that all of them were misdirected or lost [12]. Regarding the read requests, the user wants that in every 5 requests, one must have its first bit of the first byte flipped.

#### 3.2.3.2 I/O Receiver and Context Checker

The **I/O Receiver**, component is implemented just like the passthrough block device except when the requests are reads or writes.

If the request, represented by the structure `spdk_bdev_io`, is a **write**, it is passed to another component, the context checker. The context checker analyzes the structure and checks if there is any context. The main case study of this thesis is BlobFS, a file system compliant with SPDK, so we assume the context as a string that has the name of the file to which an I/O request is related to. If context checker finds a context, it means that there was context propagation and so the context checker assumes that the

[7] Documentation in `https://docs.gtk.org/glib/struct.HashTable.html`

context is a file name. The context checker searches for that file name in the write hash table, the one that has the configuration data stored. If the file name is not in the hash table, then there is no fault to inject, otherwise, it needs to be injected with a fault.

In the case of the request being a **read**, the requests are checked only when it is on its way back, like it was mentioned in Section 3.1.3.2. After that, the process is exactly the same where the context checker uses the read hash table instead, and only sends the request to the fault injector in the case there is a fault configured to inject in that request.

To determine if a fault should be injected at a given request, the context checker verifies the fault injection frequency associated with that file.

- If the frequency is **ALL** then the context checker just directs the request to the fault injector.
- If the frequency is **ALL_AFTER** then the context checker only forwards requests (i.e., read or write) to the fault injector after N requests of the same type have already been handled by the block device.
- If the frequency is **INTERVAL** the context checker must check how many requests of that type were made and check if the number of that requests is a multiple of the one configured.

#### 3.2.3.3 Fault Injector

The fault injector implementation is a bridge between the faulty block device and the faulty library. All the requests that reach this component will be injected with a fault. All that the fault injector does is to inspect the information retrieved from the hash table earlier by the context checker, and use that data to call the correct functions from the faults library.

## 3.3 Making virtual block devices compatible with FISPDK

Block devices that are above of the faulty block device in the stack must propagate the context to the lower layers, so that FISPDK can operate properly. Virtual block devices must be able to propagate the contexts according with their logic.

There are some block devices modules already provided by SPDK (e.g., passthrough, crypt (which encrypts content of I/O requests), compress (which compresses the content of I/O requests),etc). To make these block devices compliant with FISPDK, the block devices must expose the extended API and be able to propagate context to the lower layers.

To show that it is possible to make other block devices FISPDK-compliant we took the simpler block device, the passthrough, and created a new version that is FISPDK-compliant. To do that we added a context checker component that, like the one implemented in the faulty block device, needs to inspect the context of the request but only if it is a read or a write operation. If there is any context, all that this version of passthrough needs to do is to use the new API and use the context, retrieved from the

`spdk_bdev_io`, as an argument of the new read and write functions. This way the API will create a structure `spdk_bdev_io` with the context on it. This way, all the information that comes from the upper levels, like an application for instance, will be delivered to the lower level. If in the lower level we have a faulty block device, it will be able to work properly, because it will have the context to be able to inject the faults. If the context field has nothing, then the passthrough block device does its normal job of directing the request to the lower level.

## 3.4 Making Applications FISPDK-compliant

An application that is SPDK-compliant and uses the block device interface, exposed by the SPDK environment, can easily be adapted to use FISPDK. For that it only needs to add the faulty block device to the stack and ensure that all of the block devices that are above the faulty one are compatible with the framework which requires minor code changes. More specifically, these changes consist in replacing the original write and read calls with those that support context propagation, as well as adding the necessary context to be propagated down the I/O stack. For example, if an application wants to inject faults in all write requests that are made by a specific thread, users can propagate the name of the thread and configure it in the configuration file.

Like it was already mentioned, BlobFS is the main System Under Test (SUT) of this thesis, so we had to adapt it to use our fault injection tool. An important thing to keep in mind is that BlobFS uses Blobstore, so to adapt BlobFS, first we have to make Blobstore ready to propagate context.

### 3.4.1 Blobstore

The **Blobstore** is a storage service of SPDK (2.1) that applications can use to store their data in an SPDK storage stack. To enable Blobstore to use FISPDK is straightforward, we only had to change the read and write functions that it uses from the original block device API to the new ones that allow context propagation. However, this does not allow applications that use Blobstore to propagate their own context to the block device layer, because there is no path throughout Blobstore for the context to pass. So we need to instrument a way for Blobstore to pass extra information from applications to the block devices.

We followed a similar approach to the one taken in the block device API, namely extend the write and read Blobstore API to propagate application-level information down the I/O stack. However, Blobstore is a complex component and to propagate context throughout all its functions like we do in the block device layer would require significant code refactoring. To overcome this, we noticed that the request flow of Blobstore is conducted by the same SPDK thread (i.e., there are no delegations), and thus, that same thread can also carry the context information along the request.

To achieve this, we first extended the original Blobstore API to include the context as an additional argument of write and read requests. Then, we extended the SPDK threading library to include an additional `context` field in the internal thread structure, which is similar to what is made in the OS's Thread Local

Storage (TLS) mechanism [21]. When the request is about to leave the Blobstore to go to the lower levels, the thread storage is checked and if there is any context, then the extended block device API is used to propagate that context.

The extension of the Blobstore was implemented using 139 LoC (15 for the new write functions, 15 for the new read functions, 90 to make Blobstore compatible with the extended block device API and 19 for the new thread functionalities).

### 3.4.2 BlobFS

BlobFS is the application that we will test, so unlike Blobstore, the objective is not to offer applications that use BlobFS a way to propagate their own context throughout the file system and into the lower layers. Instead, BlobFS will pass its own context that, in this case, is the name of the file to which the requests are related to. So every write or read request from the BlobFS to the Blobstore must be replaced with the new functions referenced in Section 3.4.1.

Despite of having two different API's, one synchronous and another asynchronous, BlobFS only ensures that the synchronous works properly, so we will be focusing in the synchronous operations. Because BlobFS has a cache system, a request made to BlobFS does not necessarily needs to reach the block device layer to be considered completed. The requests might be answered only reaching the cache. This means that for a user of BlobFS, the frequency policies defined by FISPDK can appear wrong, because one request to BlobFS does not necessarily means that one request was made to the block device layers. For write requests, this can be solved by forcing a flush after every write, however, for read requests it is a more complex task. For read requests, if the data that we are trying to retrieve is in cache, the faults that we want to inject are not injected because the request never reaches the faulty block device. This does not mean that FISPDK is unable to corrupt read requests, it just means that FISPDK can only inject faults in requests that in fact reach the faulty block device.

The implementation of this mechanism was straightforward, where every read and write requests made by BlobFS to Blobstore that are related to a BlobFS file were replaced with the new functions with context propagation. This implied the modification of 5 LoC and the addition of 20 LoC.

# 4 Evaluation

In this chapter, we conduct a thorough experimental evaluation to assess FISPDK performance and usefulness. With this evaluation we intend to be able to answer the following questions:

- What is the overhead that FISPDK inputs in a storage stack?
- What is the performance impact of doing context propagation?
- Is FISPDK able to inject all the faults that were proposed?

## 4.1 Methodology

To validate our tool of fault injection in an SPDK environment it is important to evaluate two main aspects:

1. **Performance**: to measure the performance of FISPDK, we considered two main metrics, the throughput and the latency. Measuring these metrics is very important to validate if the system is able to be used in real scenarios and if it can inject faults in a fast and efficient way. It is important that FISPDK does not add significant overhead to the end-to-end performance of the overall system stack, because if it does, it can give the false idea that a potential SUT has recovery politics that are very slow when, in reality, it is the fault injection tool that is causing overhead in the system.
2. **Utility**: testing FISPDK's feasibility is important to prove that it can be a solution to test fault tolerance of SPDK-compliant storage systems. To do that we will use FISPDK to test the fault tolerance of BlobFS.

### 4.1.1 Experimental Setup

All tests were conducted using identical machines with the operating system Linux Ubuntu 18.04 LTS 64 bits. They were configured with an Intel(R) Core(TM) i5-9500 CPU with 6 cores clocked at 3.00GHz, 16GB of memory, and an NVMe with 250GB.

### 4.1.2 Microbenchmark Experiments

For the performance tests, we used FIO [2], a benchmarking and workload simulation tool, which is the typically used tool to evaluate block devices, and SPDK brings a plugin of FIO, compatible with the block device layer API. FIO provides 4 main types of workloads:

- **Sequential read** - this workload consist on reading sequential positions of disk blocks.
- **Random read** - this workload consists in reading the same data than the previous one but instead of every read being sequential, the disk blocks positions from where the read is performed are random.
- **Sequential write** - this workload consists in writing data to sequential disk blocks positions.
- **Random write** - this workload consists in writing data to random disk blocks positions.

With FIO we can retrieve important metrics like latency and throughput. To measure resource usage, including CPU and memory, we used the dstat[1] monitoring tool.

#### 4.1.2.1 Experimental Setups

To understand how FISPDK compares in terms of performance with different SPDK alternatives, experiments were conducted under five setups:

**SPDK Vanilla (baseline)-** To serve as comparison, first we ran FIO workloads in the original SPDK stack. This environment consisted in a simple stack of just one original SPDK's NVMe block device serving has the block device layer and the SPDK NVMe driver.

**Context Propagation Only (context propagation) -** Because FISPDK is composed of multiple components, to evaluate the overhead imposed by each one of them is needed. With this setup we intend to understand the cost of the context propagation mechanism of FISPDK. This setup has a storage stack similar with the previous one, but FIO uses the extended version of the block device API instead of the original one.

**Faulty Block Device Without Context Propagation (no propagation) -** In this setup, we added the faulty block device on top of the original stack, but it is in passthrough mode, because there is no context propagation, disabling the faulty block device from injecting faults.

**Faulty Block Device With Context Propagation and Without Fault Injection (no fault injection) -** In this setup we have again the faulty block device in the stack in passthrough mode, but this time with context propagation. It is in passthrough mode by not having any configuration to inject faults.

**Faulty Block Device With Context Propagation and Fault Injection (FISPDK) -** This setup represents the complete version of FISPDK where faults are configured related to the context that FIO propagates. Because there are several types of faults, we considered 3 sub-setups, namely a setup were

[1] Dstat manual page: https://linux.die.net/man/1/dstat

the faulty block device injects *custom bit flip* faults (more specifically flips the first bit of the first byte of every request), a setup where the faulty block device injects *random bit flips* faults, and lastly a setup where *all the bits are flipped to zero*. We did not considered the flip of all bits to one because of the similarity with the flip all bits to zero.

#### 4.1.2.2 Workloads and testing methodology

We conducted experiments using the FIO benchmarking tool. We considered sequential read, random read, sequential write and random write workloads under a single-threaded setting. We only use single-threaded settings because FIO plugin is not ready for multi-threading testing.

For each workload, we used two different block sizes: 512B, which respects to the default size of block devices (including SPDK), and 4KB, the default block size of the file system.

For each experiment (combination of workload and block size), we conducted 3 runs, each of which was executed whether the total amount of bytes read/written reached 128GiB or reached the 20 minutes of runtime.

### 4.1.3 BlobFS Experiments

To understand the performance and feasibility of injecting faults with FISPDK, we conducted experiments over realistic applications, namely BlobFS.

#### 4.1.3.1 Experimental Setups

To test BlobFS fault tolerance and to compare the performance of BlobFS with and without FISPDK, experiments were conducted under two setups:

**Without FISPDK** - To serve as the baseline, the benchmark of BlobFS was run in a simple setup with BlobFS and Blobstore with a vanilla version and the block device stack with only the NVMe block device.

**With FISPDK** - In this setup, we have BlobFS and Blobstore with their extended versions (Section 3.4), and with the faulty block device on top of the NVMe block device. We will be testing the different frequencies of fault injection to prove that FISPDK is capable of injecting faults in the configured frequencies, so this setup is composed of 3 different sub-setups. The fault type used in all the test is the custom bit flip content corruption.

#### 4.1.3.2 Benchmark and Dataset

It is important to have realistic data that have significant size and are frequently written to storage. For datasets, it was used 3 different linux images that have significant size and are used to test many different types of systems. These images can be downloaded from https://cloud-images.ubuntu.com/. The images used were the bionic server image with the size of 1.208 GB, the focal server image with the size of 1.4883 GB and the jammy server image with the size of 1.542 GB.

The conducted workloads consists of three main steps:

1. First, write all images to BlobFS. To prevent dirty data to be cached at BlobFS's internal cache, each write is followed by a flush (as discussed in 3.4.2). Further, all writes are made with 4KB block size.

2. Then, all files are read, firstly the bionic image, then the jammy image and lastly the focal image.

3. Finally, we checksum each file with the md5 hashing scheme to validate if the written files are equal to the original ones.

Under the FISPDK setup, the faulty block device was configured to inject faults in one of the dataset files (the bionic image). Runtime will be the only parameter to be measured because it is simpler to measure and the purpose is just to have a comparison between the two setups. The runtime is only related to the write phase of the workload, because it is when FISPDK takes action.

Once again we resorted to the strategy of running each experiment 3 times.

## 4.2 Experimental Results

In this section we start by showing and discussing the microbenchmarks results (4.2.1) and then we present and discuss the results of the BlobFS tests (4.2.2).

### 4.2.1 Microbenchmarks Results

This section presents the microbenchmark experimental results for all SPDK setups under both 4KB and 512B block sizes.

#### 4.2.1.1 Experiments under 4KB block size

Tables 5 and 6 depict the throughput and latency results, respectively, obtained with FIO benchmark under the 7 setups discussed in 4.1.2.1 with a block size of 4KB.

| | | Throughput(MB/s) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Baseline | Context Propagation | No Propagation | No Fault Injection | FISPDK (Custom Bit Flip) | FISPDK (Random Bit Flip) | FISPDK (Flip All Zeros) |
| **seq-read** | AVG | 1168.0 | 1081.667 | 1072.0 | 1052.333 | 1053.63 | 1081.667 | 1072.0 |
| | STDEV | 82.274 | 29.956 | 24.331 | 1.155 | 11.015 | 0.832 | 24.331 |
| **rand-read** | AVG | 323.333 | 323.0 | 323.0 | 323.0 | 323.0 | 320.667 | 320.0 |
| | STDEV | 0.577 | 0.0 | 0.0 | 0.0 | 0.0 | 0.577 | 0.0 |
| **seq-write** | AVG | 438.667 | 449.0 | 432.0 | 427.333 | 427.667 | 417.333 | 405.667 |
| | STDEV | 24.826 | 1.0 | 7.81 | 2.309 | 1.155 | 19.858 | 0.577 |
| **rand-write** | AVG | 436.333 | 446.333 | 449.0 | 449.667 | 449.667 | 430.333 | 417.667 |
| | STDEV | 22.811 | 4.619 | 0.0 | 0.577 | 0.577 | 30.172 | 1.528 |

Table 5: Throughput results (4KB).

| | | Latency(usec) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Baseline | Context Propagation | No Propagation | No Fault Injection | FISPDK (Custom Bit Flip) | FISPDK (Random Bit Flip) | FISPDK (Flip All Zeros) |
| **seq-read** | AVG | 28.027 | 30.177 | 30.437 | 31.02 | 30.977 | 30.177 | 30.437 |
| | STDEV | 2.073 | 0.832 | 0.692 | 0.044 | 0.316 | 0.832 | 0.692 |
| **rand-read** | AVG | 101.133 | 101.173 | 101.207 | 101.26 | 101.41 | 102.107 | 102.123 |
| | STDEV | 0.047 | 0.025 | 0.05 | 0.01 | 0.017 | 0.134 | 0.012 |
| **seq-write** | AVG | 74.76 | 72.883 | 75.793 | 76.523 | 76.47 | 78.447 | 80.673 |
| | STDEV | 4.417 | 0.145 | 1.35 | 0.367 | 0.202 | 3.618 | 0.14 |
| **rand-write** | AVG | 75.127 | 73.26 | 72.77 | 72.757 | 72.753 | 76.237 | 78.37 |
| | STDEV | 4.099 | 0.832 | 0.01 | 0.021 | 0.015 | 5.114 | 0.262 |

Table 6: Latency results (4KB).

**Sequential read** Looking to the baseline results of the sequential read workload we have an average throughput of 1168 MB/s and a latency averaging 28.027 microseconds per operation. The context propagation only setup values (1081.667 MB/s for throughput and 30.177 for latency) and the faulty block device without context propagation values (1072 MB/s for throughput and 30.437 for latency) are very similar, presenting a small overhead when compared with the baseline. That was expected because in the first there is the propagation of context logic added to the workload and in the second there is an extra virtual block device in the block device layer. These values show that the overhead of having context propagation or an extra virtual block device in the stack impose at most 8.5% overhead when compared with

the baseline. Looking now to the results from the faulty block device with context propagation and without fault injection setup (1052.333 MB/s for throughput and 31.02 for latency), it does not exhibit significant overhead when compared with the baseline and even with the previous two setups. Analyzing the results for the three setups with FISPDK we can conclude the fault injection part does not have any impact in the storage stack performance, because it does not present worst results when compared with the previous setup for any fault injection type. It even has some minor improvements, that are not significant because we are talking about differences of 3% that can be related with variance.

**Random read** The baseline results for the random read workload are 323.333 MB/s and 101.133 microseconds per operation for throughput and latency respectively. This decrease when comparing with the sequential read workload was expected due to the nature of the workload. When we look to the tables we see that the results for all the different setups average around 320 MB/s of throughput and 101 microseconds per operations of latency. This shows that the workload itself is responsible for all the overhead in the system and limits the results to these values, which shows that using FISPDK in cases similar to this workload will not have any impact in the performance of the storage stack.

**Sequential write** For the sequential write workload we have 438.667 MB/s of throughput and 74.76 microseconds of latency per operation for the baseline setup. Analyzing the context propagation only setup, the values shows a slightly improvement to the baseline values, with 449 MB/s of throughput and 72.883 microseconds per operation. But once again these are non significant differences of around 2% that must be related with variance, and looking to the standard deviation values of the baseline setup (24.826 for throughput and 4.417 for latency) we can support this conclusion. The faulty block device without fault injection setup presents results (432 MB/s of throughput and 75.793 microseconds of latency per operation) that are very close to the baseline showing that for this workload, the addition of a block device to the block device layer does not input significant overhead to the storage stack. Looking now to the setup that groups both context propagation and the faulty block device but without fault injection, we have a slightly worst results (427.333 MB/s of throughput and 76.523 microseconds of latency per operation) than the baseline and the other two already discussed setups. Despite of not happening in both read workloads, this was expected because there are two joint factors that give extra work to each request, the fact that it has to pass through an extra block device and that has to propagate context simultaneously. However, the difference is less than 3% which is an affordable price to pay regarding the extra features that the setup enables. Regarding the setups with the full FISPDK, we can see that injecting the custom bit flip fault there is no extra overhead if we compare it with the previous setup, but this is not the case for the other two fault types. For the FISPDK setup with the random bit flip fault injection we have 417.333 MB/s of throughput and 78.447 microseconds of latency per operation, which are slightly worst results than the ones with the custom bit flip fault injection (427.667 MB/s of throughput and 76.47 microseconds of latency per operation). This can be explained by the extra processing of the randomize function that is used to determine the specific bit that must be flipped. With the FISPDK setup where the fault injected is the flip

all zeros, we have worst results than in the other two FISPDK setups (405.667 MB/s of throughput and 80.673 microseconds of latency per operation), which are caused by the full replacement of the content.

**Random write** Analyzing both Table 5 and Table 6, we can see that random write workload has the same pattern of results than the sequential read workload, which was expected due to the similar nature of both workloads.

Looking now to all the experimental results of the microbenchmarks with the 4KB block size we can conclude that the fault injection part of FISPDK has slightly more impact in the performance of the storage stack in write operations. However, the general analysis shows us that the overhead that FISPDK inputs into the storage stack is not significant, allowing us to conclude that the fault injection tool has a good performance.

#### 4.2.1.2 512 bytes Tests

Tables 7 and 8 depict the throughput and latency results, respectively, obtained with FIO benchmark under the 7 setups discussed in 4.1.2.1 with a block size of 512 bytes.

| | | **Throughput(MB/s)** | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Baseline | Context Propagation | No Propagation | No Fault Injection | FISPDK (Custom Bit Flip) | FISPDK (Random Bit Flip) | FISPDK (Flip All Zeros) |
| **seq-read** | AVG | 304.0 | 299.0 | 298.0 | 298.0 | 297.667 | 295.333 | 297.667 |
| | STDEV | 8.718 | 1.732 | 0.0 | 0.0 | 0.577 | 1.155 | 0.577 |
| **rand-read** | AVG | 41.3 | 41.3 | 41.3 | 41.3 | 41.267 | 40.5 | 40.4 |
| | STDEV | 0.0 | 0.0 | 0.0 | 0.0 | 0.058 | 0.0 | 0.693 |
| **seq-write** | AVG | 180.667 | 140.333 | 141.0 | 141.333 | 141.667 | 142.0 | 141.667 |
| | STDEV | 73.078 | 2.082 | 1.0 | 0.577 | 0.577 | 0.0 | 0.577 |
| **rand-write** | AVG | 21.633 | 20.933 | 22.233 | 21.533 | 20.833 | 21.6 | 21.3 |
| | STDEV | 0.635 | 0.153 | 2.065 | 1.021 | 0.058 | 0.0 | 0.52 |

Table 7: Throughput results (512B).

| | | Latency(usec) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Baseline | Context Propagation | No Propagation | No Fault Injection | FISPDK (Custom Bit Flip) | FISPDK (Random Bit Flip) | FISPDK (Flip All Zeros) |
| **seq-read** | AVG | 13.363 | 13.58 | 13.61 | 13.62 | 13.63 | 13.623 | 13.633 |
| | STDEV | 0.376 | 0.078 | 0.0 | 0.01 | 0.02 | 0.032 | 0.006 |
| **rand-read** | AVG | 98.933 | 98.977 | 98.977 | 99.0 | 99.07 | 100.833 | 101.237 |
| | STDEV | 0.006 | 0.006 | 0.012 | 0.0 | 0.017 | 0.006 | 1.822 |
| **seq-write** | AVG | 24.743 | 29.097 | 28.877 | 28.83 | 28.853 | 28.757 | 28.837 |
| | STDEV | 8.169 | 0.416 | 0.16 | 0.13 | 0.084 | 0.032 | 0.055 |
| **rand-write** | AVG | 189.153 | 195.563 | 185.127 | 190.39 | 196.297 | 189.467 | 192.46 |
| | STDEV | 5.531 | 1.516 | 16.336 | 8.947 | 0.717 | 0.355 | 4.915 |

Table 8: Latency results (512B).

Analyzing both tables, and comparing these results with the ones from the previous section, the only big difference that we see is in the proportion of the values. However, it is normal for this to happen, because in the previous tests FIO sends 4KB of content in each requests, and in these tests it only sends 512 bytes, making the performance of the system decrease drastically. However, the obtained results follow a similar pattern as those of 4 KB, and thus, we draw similar conclusions. This reinforces the idea that FISPDK does not add significant overhead to the storage stack.

#### 4.2.1.3 Resource Usage

Regarding the resource usage, FISPDK does not input significant overhead both in CPU and RAM usage, as depicted in Annex II. The CPU usage values are always around the **16%** mark in all the setups including the baseline. About the usage of memory, the usage in all the setups is always around 5300 MB in all setups. The RAM values were based in only one run, because the memory usage is the same for each run of each experiment and so we expect RAM usage to be similar between runs.

### 4.2.2 BlobFS Experimental Results

This section discusses the BlobFS experimental results under both vanillan SPDK and FISPDK setups. Table 9 depicts the results of corrupting only the bionic image by flipping the first bit of the first byte in the defined write requests. Here we discuss the experiments of the different frequencies of fault injection provided by FISPDK. Annex III shows the faulty block device configuration files used for all of these experiments.

Looking first to the performance evaluation of the workload, the results of the SPDK vanilla serve as baseline to compare with the other 3 setups. The write phase of the workload, which includes writing all

| | | Workload Time (sec) | Bionic Image Corruption | | |
|---|---|---|---|---|---|
| | | | Corrupted | All 3 Runs Match Hash | Number of Corrupted Blocks |
| **SPDK Vanilla** | AVG | 42.52 | - | - | - |
| | STDEV | 0.07 | | | |
| **Interval (500)** | AVG | 42.85 | ✓ | ✓ | 633 |
| | STDEV | 0.16 | | | |
| **All** | AVG | 43.53 | ✓ | ✓ | 316672 |
| | STDEV | 0.66 | | | |
| **All After (500)** | AVG | 42.88 | ✓ | ✓ | 316172 |
| | STDEV | 0.2 | | | |

Table 9: BlobFS Tests Results.

3 linux images, averages 42.52 seconds of run time. Looking now to the FISPDK setup where the fault injection tool injects a fault every 500 write requests at the bionic image, we have a slightly higher average run time (42.85 seconds). Those are good results, when we compare that to the baseline, knowing that in this workload there is context propagation for all writes of all three images and there is injection of faults in requests of one of the images. In FISPDK setup with the frequency ALL, that consists in injecting faults in every requests related to the bionic image file, despite of having the worst average result of them all (43.53 seconds), which is expected since it does everything that the other FISPDK setups and injects faults more frequently, we only have the increment of approximately 1 second which is good when comparing the amount of extra work that this workload implies. Regarding the FISPDK setup with the fault injection frequency ALL After (500), that consists of injecting faults to all the requests related to the bionic image file after the request 500, we once again have slightly higher average run time (42.88 seconds) than the baseline, which was expected regarding all the extra computation referenced above. These results support the conclusion taken with the microbenchmarks, that FISPDK does not add much overhead considering the extra computation that it involves. These also allow us to understand that both Blobstore and BlobFS extensions also have good performance.

Regarding the corruption information given by Table 9, we can immediately conclude that BlobFS is not tolerant to content corruption. We can say this because when we hashed the original bionic file and the one stored in BlobFS, the result was not the same. We did the same procedure to the other two images and the hash from the original files and the ones stored in BlobFS are equal, showing that FISPDK successfully differentiated the requests and managed to only corrupt the configured file.

The fact that the hash of the corrupted bionic image matches in the 3 runs over the same setup proves that FISPDK is able to deterministically inject faults and to reproduce the same failure.

Lastly looking to the number of corrupted blocks, we have a total of 316672 4KB blocks composing the bionic file. Analyzing every setup we have:

- **Interval (500) -** Here we have 633 corrupted blocks that is exactly what was expected of this

setup[2]. On top of counting the number of corrupted blocks, we also checked which ones were corrupted and we saw that the block 500 the 1000 and so on were the ones corrupted, proving that FISPDK does exactly what it was configured to do.

- **All -** Here we can see that all of the 316672 blocks are corrupts, showing that FISPDK was able to inject a content corruption fault in every single write request related to the bionic image file.
- **All After (500) -** Once again we have the number of corrupted blocks (316172) that we expected to have, and once again we checked that all the first 500 blocks of the bionic image file were the ones that did not suffered corruption.

With these BlobFS experimental results we can conclude that FISPDK was able to corrupt the bionic image file, while adding minimal overhead into the storage stack.

## 4.3 Discussion

With our evaluation, we measured the impact that FISPDK has on an SPDK storage stack. On top of that, we also tested BlobFS with our tool and concluded that the file system is not fault tolerant.

In the microbenchmarks experiments we tested five different setups: *SPDK Vanilla*, *Context Propagation Only*, *Fautly Block Device Without Context Propagation*, *Faulty Block Device With Context Propagation and Without Fault Injection* and *Faulty Block Device With Context Propagation and Fault Injection*. Having these different setups allows us to evaluate each component of FISPDK. The results showed that all of the components of FISPDK (the faulty block device, the faults library and the extended block device API) do not add significant overhead into the SPDK storage stack. With these experiments we also measured the usage of CPU and RAM, and we got around 16% for CPU usage and 5300MB for RAM usage in every conducted experiment.

With the BlobFS experiments, we tested the fault tolerance of BlobFS. We used an SPDK vanilla setup and a full FISPDK setup. For these tests we used a more realistic workload with a real dataset. Comparing the workloads runtime of both setups, we were able to validate that FISPDK performed well adding minimal overhead into the storage stack. The comparison of the files stored in BlobFS with the original ones allowed us to understand that the files targeted with fault injection were corrupted, showing that BlobFS is not fault tolerant.

This results show that FISPDK is a valid solution to inject faults deterministically in an SPDK environment with minimal overhead.

[2] 316672/500 = 633

# 5 Conclusion

Storage systems that use traditional multilayered storage stacks, require expensive context switching between user and kernel space which presents significant performance overhead for applications that resort to these. To solve this problem, SPDK enables building a full user space storage stack, bypassing the kernel. With performance being the main focus of this development kit, the correctness and reliability of the storage systems is often forgotten, and there are no tools that are able to test SPDK-compliant storage systems reliability.

Therefore, this thesis proposes FISPDK, a fault injection tool for SPDK that is able to inject data corruption and delay in I/O requests at the block device level. In order to provide deterministic fault injection, FISPDK enables I/O differentiation using context propagation throughout the SPDK storage stack, which is possible due to an extension of the original block device API. For the fault injection to happen at the block device level, FISPDK presents a virtual block device that is able to intercept read and write operations and inject several types of faults in them. The block device makes use of the contexts that are propagated to differentiate which requests must or must not be injected with a fault. By providing a configuration module, FISPDK allows users to configure what requests should be targeted with fault injection, and also with which type, regarding their propagated context.

Porting Blobstore and BlobFS to use FISPDK required less than 200 LoC. This shows that adapting a system to use FISPDK is straightforward.

A detailed experimental evaluation of FISPDK shows that the tool is able to inject faults deterministically and to reproduce failure scenarios, without imposing significant performance overhead into the storage stack. The microbenchmark experiments made with FIO and dstat show in detail that each individual component of FISPDK does not add significant overhead for different workloads (random read, random write, sequential read and sequential write), as well as it does not add any stress to the machine resources, presenting the same CPU and RAM usage as the SPDK vanilla stack. The BlobFS experiments show that FISPDK is able to test the fault tolerance of a storage system, while showing that BlobFS is not tolerant to data corruption. These tests also help proving the determinism of fault injection that the tool presents, as well as the ability to reproduce the same failed state when configured properly.

## 5.1 Future Work

This thesis opens several topics to explore in SPDK environments. Regarding the presented prototype, it would be interesting to extend the portfolio of faults that it allows to inject, for instance by supporting LSEs simulation. It would also be interesting to use FISPDK to test other storage systems that are built with SPDK. Currently there are some systems that could have their fault tolerance tested like the RocksDB extension that uses BlobFS as the base file system, or MicroFS [19] a user level file system built with SPDK.

Regarding the I/O differentiation and context propagation, this thesis opens new possibilities, because fault injection is just one of many applications that may leverage from I/O differentiation. With the extended block device API proposed by FISPDK, we can propagate various contexts that can be used for different purposes. We have the example of making differentiated caching, creating a cache that only keeps in memory content from certain I/O requests regarding the context that they propagate to improve the cache hit ratio and its effectiveness. The same goes for differentiated encryption, where a virtual block device can have encryption logic (like the crypt module provided by SPDK), but now encrypting only content coming from differentiated requests that includes sensitive data. Another example is differentiated I/O scheduling where we could order the execution of requests regarding the context that they propagate to boost applications performance. As a final example we have differentiated compression. An example that could be similar in term of implementation logic to the faulty block device, where users could configure which requests should perform compression based on their compressibility.

# Bibliography


[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. "Fault injection for dependability validation: A methodology and some applications". In: *IEEE Transactions on software engineering* 16.2 (1990), pp. 166–182 (cit. on p. 14).

[2] J. Axboe. *Fio-flexible io tester*. https://fio.readthedocs.io. Accessed June 23, 2022 (cit. on p. 35).

[3] 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". In: *ACM Transactions on Storage (TOS)* 4.3 (2008), pp. 1–28 (cit. on pp. 2, 11, 12).

[4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. "An analysis of latent sector errors in disk drives". In: *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 2007, pp. 289–300 (cit. on p. 11).

[5] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. "Chaos engineering". In: *IEEE Software* 33.3 (2016), pp. 35–41 (cit. on p. 15).

[6] C. Bennett and A. Tseitlin. "Chaos monkey released into the wild". In: *Netflix Tech Blog* 30 (2012) (cit. on p. 15).

[7] C. Borges and J. Paulo. *Realistic Assessment of Faults in Storage Systems*. https://dsr-haslab.github.io/repository/bp21.pdf. Distributed Storage Research at HASLab. Accessed November 12, 2021. 2021 (cit. on pp. 2, 11, 12, 17, 20, 23, 30).

[8] *Data plane development kit*. Nov. 2022. url: https://www.dpdk.org/ (cit. on p. 1).

[9] C. Esposito, A. Castiglione, and K.-K. R. Choo. "Challenges in delivering software in the cloud as microservices". In: *IEEE Cloud Computing* 3.5 (2016), pp. 10–14 (cit. on p. 22).

[10] A. Faria, R. Macedo, J. Pereira, and J. Paulo. "BDUS: implementing block devices in user space". In: *Proceedings of the 14th ACM International Conference on Systems and Storage*. 2021, pp. 1–11 (cit. on p. 17).

[11] G. B. Finelli. “Characterization of fault recovery through fault injection on FTMP”. In: *IEEE transactions on reliability* 36.2 (1987), pp. 164–170 (cit. on p. 14).

[12] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. “Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to file-system faults”. In: *ACM Transactions on Storage (TOS)* 13.3 (2017), pp. 1–33 (cit. on pp. 18, 20, 23, 24, 30).

[13] C. Hellwig. “XFS: the big storage file system for Linux”. In: *; login:: the magazine of USENIX & SAGE* 34.5 (2009), pp. 10–18 (cit. on p. 1).

[14] M.-C. Hsueh, T. Tsai, and R. Iyer. “Fault injection techniques and tools”. In: *Computer* 30.4 (1997), pp. 75–82 (cit. on p. 13).

[15] Json-C. *GitHub Repository, JSON-C*. `https://github.com/json-c/json-c`. Accessed January 29, 2023 (cit. on p. 28).

[16] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. “FERRARI: A Tool for The Validation of System Dependability Properties”. In: *Digest of Papers: FTCS-22, The Twenty-Second Annual International Symposium on Fault-Tolerant Computing, Boston, Massachusetts, USA, July 8-10, 1992*. IEEE Computer Society, 1992, pp. 336–344 (cit. on p. 14).

[17] T. Killalea. “The hidden dividends of microservices”. In: *Communications of the ACM* 59.8 (2016), pp. 42–45 (cit. on p. 22).

[18] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. “Distributed fault-tolerant real-time systems: The Mars approach”. In: *IEEE Micro* 9.1 (1989), pp. 25–40 (cit. on p. 14).

[19] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. “Scale and Performance in a Filesystem Semi-Microkernel”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 819–835. isbn: 9781450387095 (cit. on pp. 10, 45).

[20] J. Mace and R. Fonseca. “Universal Context Propagation for Distributed System Instrumentation”. In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys ’18. Porto, Portugal: Association for Computing Machinery, 2018. isbn: 9781450355841 (cit. on pp. 21, 22).

[21] R. Macedo, Y. Tanimura, J. Haga, V. Chidambaram, J. Pereira, and J. Paulo. “PAIO: General, Portable I/O Optimizations With Minor Application Modifications”. In: *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 2022, pp. 413–428 (cit. on pp. 21, 22, 33).

[22] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. “The new ext4 filesystem: current status and future plans”. In: *Proceedings of the Linux symposium*. Vol. 2. Citeseer. 2007, pp. 21–33 (cit. on p. 1).

[23] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. “Finding crash-consistency bugs with bounded black-box crash testing”. In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 33–50 (cit. on pp. 2, 15, 16).

[24] Netflix. *Netflix Open Source*. Accessed October 18, 2022. url: https://netflix.github.io/ (cit. on p. 22).

[25] S. Newman. *Building microservices*. ”O’Reilly Media, Inc.”, 2021 (cit. on p. 22).

[26] J. Paulo, P. Reis, J. Pereira, and A. Sousa. “Dedisbench: A benchmark for deduplicated storage systems”. In: *OTM Confederated International Conferences”On the Move to Meaningful Internet Systems”*. Springer. 2012, pp. 584–601 (cit. on p. 18).

[27] A. Rebello, Y. Patel, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. “Can Applications Recover from fsync Failures?” In: *ACM Transactions on Storage (TOS)* 17.2 (2021), pp. 1–30 (cit. on p. 17).

[28] O. Rodeh, J. Bacik, and C. Mason. “BTRFS: The Linux B-Tree Filesystem”. In: *ACM Trans. Storage* 9.3 (Aug. 2013). issn: 1553-3077 (cit. on p. 1).

[29] E. W. D. Rozier. *Understanding the fault-tolerance properties of large-scale storage systems*. University of Illinois at Urbana-Champaign, 2011 (cit. on p. 11).

[30] S. Scargall. “Introducing the persistent memory development kit”. In: *Programming Persistent Memory*. Springer, 2020, pp. 63–72 (cit. on p. 1).

[31] R. Schiesser. *IT systems management*. Prentice Hall PTR, 2010 (cit. on p. 1).

[32] B. Schroeder, S. Damouras, and P. Gill. “Understanding latent sector errors and how to protect against them”. In: *ACM Transactions on storage (TOS)* 6.3 (2010), pp. 1–23 (cit. on p. 11).

[33] B. Schroeder, E. Pinheiro, and W.-D. Weber. “DRAM errors in the wild: a large-scale field study”. In: *ACM SIGMETRICS Performance Evaluation Review* 37.1 (2009), pp. 193–204 (cit. on p. 23).

[34] *SPDK: Writing a Custom Block Device Module*. url: https://spdk.io/doc/bdev_module.html (visited on 11/02/2022) (cit. on pp. 7–9).

[35] S. Swanson and A. M. Caulfield. “Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage”. In: *Computer* 46.8 (2013), pp. 52–59 (cit. on p. 1).

[36] T. K. Tsai and R. K. Iyer. “Measuring fault tolerance with the FTAPE fault injection tool”. In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 1995, pp. 26–40 (cit. on pp. 2, 14).

[37] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. “SPDK: A Development Kit to Build High Performance Storage Applications”. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2017, pp. 154–161 (cit. on pp. 1, 2, 5, 6, 10).

# I

# Annex 1 - Configuration File Example

```
{
  "write":{
     "files":[
       {
         "fileName": "focal.img",
         "faultInjection":{
           "frequency": "ALL",
           "faultType":{
             "type":"CORRUPT_CONTENT",
             "corruptionPattern":"BITFLIP_CUSTOM_INDEX",
             "offset": 0,
             "index": 0
           }
         }
       },
       {
        "fileName": "bionic.img",
        "faultInjection":{
          "frequency": "ALL_AFTER",
          "nRequests": 3,
          "faultType":{
            "type":"CORRUPT_CONTENT",
            "corruptionPattern": "REPLACE_ALL_ZEROS"
          }
        }
      }
     ]
  },
  "read":{
    "files":[
      {
        "fileName": "bionic.img",
        "faultInjection":{
          "frequency": "INTERVAL",
          "nRequests": 5,
          "faultType":{
            "type":"CORRUPT_CONTENT",
            "corruptionPattern":"BITFLIP_CUSTOM_INDEX",
            "offset": 0,
            "index": 0
          }
```

```
        }
      }
    ]
  }
}
```

Listing I.1: Configuration file example.

# II

# Resource Usage Results

| | | CPU(%) | | | | RAM(MB) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | SPDK Vanilla | Context Propagation Only | Faulty Bdev Without Propagation | Faulty Bdev Without Fault Injection | SPDK vanilla | Context Propagation Only | Faulty Bdev Without Propagation | Faulty Bdev Without Fault Injection |
| **seq-red** | AVG | 16.58 | 16.56 | 16.59 | 16.57 | 5345.13 | 5328.46 | 5266.99 | 5266.57 |
| | STDEV | 0.05 | 0.02 | 0.02 | 0.02 | - | - | - | - |
| **rand-read** | AVG | 16.66 | 16.66 | 16.64 | 16.64 | 5267.49 | 5266.65 | 5266.54 | 5266.65 |
| | STDEV | 0.03 | 0.03 | 0.01 | 0.0 | - | - | - | - |
| **seq-write** | AVG | 16.62 | 16.64 | 16.63 | 16.63 | 5266.76 | 5266.63 | 5266.69 | 5266.68 |
| | STDEV | 0.01 | 0.03 | 0.0 | 0.0 | - | - | - | - |
| **rand-write** | AVG | 16.64 | 16.63 | 16.63 | 16.63 | 5308.29 | 5267.09 | 5266.99 | 5267.16 |
| | STDEV | 0.02 | 0.01 | 0.0 | 0.01 | - | - | - | - |

Table 10: CPU and RAM results with no fault injection (4KB).

| | | CPU(%) | | | | RAM(MB) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | SPDK Vanilla | Context Propagation Only | Faulty Bdev Without Propagation | Faulty Bdev Without Fault Injection | SPDK vanilla | Context Propagation Only | Faulty Bdev Without Propagation | Faulty Bdev Without Fault Injection |
| **seq-red** | AVG | 16.71 | 16.87 | 16.72 | 16.73 | 5303.00 | 5267.50 | 5267.44 | 5266.88 |
| | STDEV | 0.11 | 0.37 | 0.11 | 16.73 | - | - | - | - |
| **rand-read** | AVG | 16.67 | 16.67 | 16.67 | 16.67 | 5267.07 | 5266.58 | 5266.87 | 5267.40 |
| | STDEV | 0.01 | 0.0 | 0.0 | 0.0 | - | - | - | - |
| **seq-write** | AVG | 16.72 | 16.7 | 16.67 | 16.71 | 5266.84 | 5267.22 | 5266.62 | 5266.73 |
| | STDEV | 0.12 | 0.06 | 0.02 | 0.06 | - | - | - | - |
| **rand-write** | AVG | 16.68 | 16.69 | 16.68 | 16.68 | 5266.81 | 5266.94 | 5281.86 | 5267.84 |
| | STDEV | 0.02 | 0.01 | 0.0 | 0.01 | - | - | - | - |

Table 11: CPU and RAM results with no fault injection (512B).

| | | CPU(%) | | | RAM(MB) | | |
|---|---|---|---|---|---|---|---|
| | | Custom Bit Flip | Random Bit Flip | Flip All Zeros | Custom Bit Flip | Random Bit Flip | Flip All Zeros |
| **seq-read** | AVG | 16.56 | 16.55 | 16.55 | 5264.92 | 5264.55 | 5264.15 |
| | STDEV | 0.01 | 0.02 | 0.02 | - | - | - |
| **rand-read** | AVG | 16.64 | 16.61 | 16.64 | 5264.19 | 5264.16 | 5264.70 |
| | STDEV | 0.0 | 0.04 | 0.01 | - | - | - |
| **seq-write** | AVG | 16.63 | 16.63 | 16.63 | 5265.29 | 5265.01 | 5264.49 |
| | STDEV | 0.0 | 0.0 | 0.0 | - | - | - |
| **rand-write** | AVG | 16.63 | 16.63 | 16.64 | 5264.62 | 5264.40 | 5264.14 |
| | STDEV | 0.01 | 0.01 | 0.01 | - | - | - |

Table 12: CPU and RAM tests results with fault injection (4KB).

| | | CPU(%) | | | RAM(MB) | | |
|---|---|---|---|---|---|---|---|
| | | Custom Bit Flip | Random Bit Flip | Flip All Zeros | Custom Bit Flip | Random Bit Flip | Flip All Zeros |
| **seq-read** | AVG | 16.72 | 16.64 | 16.64 | 5264.11 | 5266.67 | 5264.55 |
| | STDEV | 0.11 | 0.01 | 0.0 | - | - | - |
| **rand-read** | AVG | 16.69 | 16.67 | 16.69 | 5280.27 | 5280.65 | 5263.78 |
| | STDEV | 0.02 | 0.0 | 0.03 | - | - | - |
| **seq-write** | AVG | 16.67 | 16.66 | 16.65 | 5264.23 | 5267.46 | 5264.46 |
| | STDEV | 0.02 | 0.02 | 0.0 | - | - | - |
| **rand-write** | AVG | 16.68 | 16.67 | 16.67 | 5263.75 | 5266.79 | 5263.98 |
| | STDEV | 0.0 | 0.0 | 0.0 | - | - | - |

Table 13: CPU and RAM tests results with fault injection (512B).

# III

# Annex 3 - Faulty Configuration File For BlobFS Testing

```
{
  "write":{
    "files":[
      {
        "fileName": "bionic",
        "faultInjection":{
          "frequency": "ALL",
          "faultType":{
            "type":"CORRUPT_CONTENT",
            "corruptionPattern":"BITFLIP_CUSTOM_INDEX",
            "offset": 0,
            "index": 0
          }
        }
      }
    ]
  },
  "read":{}
}
```

Listing III.1: Configuration file used for FISPDK setup with frequency ALL.

```
{
  "write":{
    "files":[
      {
        "fileName": "bionic",
        "faultInjection":{
          "frequency": "INTERVAL",
          "nRequests": 500,
          "faultType":{
            "type":"CORRUPT_CONTENT",
            "corruptionPattern":"BITFLIP_CUSTOM_INDEX",
            "offset": 0,
            "index": 0
          }
        }
      }
```

```
    ]
  },
  "read":{}
}
```

Listing III.2: Configuration file used for FISPDK setup with frequency INTERVAL(500).

```
{
  "write":{
    "files":[
      {
        "fileName": "bionic",
        "faultInjection":{
          "frequency": "ALL_AFTER",
          "nRequests": 500,
          "faultType":{
            "type":"CORRUPT_CONTENT",
            "corruptionPattern":"BITFLIP_CUSTOM_INDEX",
            "offset": 0,
            "index": 0
          }
        }
      }
    ]
  },
  "read":{}
}
```

Listing III.3: Configuration file used for FISPDK setup with frequency ALL_AFTER(500).