Extending a File System to Support NVMe Devices with SPDK

Ao Li
University of Toronto

Geoffrey Yu
University of Toronto

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse egestas arcu sit amet libero imperdiet fringilla. Nunc tempus libero vitae sapien pretium, id gravida tortor vestibulum. Suspendisse ut velit sed mi vehicula sollicitudin eu quis urna. Nulla sed libero eu enim fringilla eleifend ac at augue. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nunc et dapibus turpis. Vestibulum fringilla nibh a enim tempus lobortis. Ut posuere risus eu mi consequat, eget pretium nibh varius.

1 Introduction

Lorem ipsum [1] dolor sit amet, consectetur adipiscing elit. Suspendisse egestas arcu sit amet libero imperdiet fringilla. Nunc tempus libero vitae sapien pretium, id gravida tortor vestibulum. Suspendisse ut velit sed mi vehicula sollicitudin eu quis urna. Nulla sed libero eu enim fringilla eleifend ac at augue. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nunc et dapibus turpis. Vestibulum fringilla nibh a enim tempus lobortis. Ut posuere risus eu mi consequat, eget pretium nibh varius.

Donec maximus purus nec elit dapibus ultricies nec vitae mi. Pellentesque vel turpis vitae mi sodales volutpat. Nullam vehicula neque eu ipsum pretium, eget aliquet nibh convallis. Proin iaculis, tortor pulvinar ultrices dictum, magna massa imperdiet arcu, ac bibendum velit arcu eu odio. Vivamus vitae lectus porta, tristique diam eu, vestibulum dolor. Maecenas dapibus porttitor cursus. Nunc vel quam eu mi fringilla congue. Vestibulum fermentum sagittis arcu, ut pharetra eros eleifend nec. Nam nisl augue, efficitur in porta eget, dapibus eget mauris. Vivamus sit amet sapien faucibus, elementum odio et, pulvinar purus.

2 Background

In this section we present the background of NVMe interface and SPDK framework.

2.1 NVMe and I/O Queue

NVMe [2] is an open interface specification designed to allow host software to communicate with a non-volatile memory subsystem (NVM) via a peripheral component interconnect express (PCIe) bus. Previous standards such as serial-attacked SCSI and serial advanced technology attachment can handle queue depths of 254 and 32 respectively. NVMe is able to handle queue depths of up to 65535 I/O queues with up to 64 Ki outstanding commands per I/O queue, which allows an NVMe device to support parallel operations. An I/O queue is composed of a submission queue and a completion queue. Host software issues I/O commands to a submission queue and completions are placed into the associated completion queue by the controller. Note that the order of completions is not determined by the submission oder of the commands.

2.2 SPDK and Block Devices

SPDK [1] is an open source library that allows developer to implement high performance, scalable, user-mode storage applications. A block device in SPDK is an abstraction of all block devices, where I/O commands are processed and issued to corresponding physical block devices such as NVMe devices and Malloc devices. SPDK provides a event framework, where different threads to exchange data through passing messages to one another. It allows a user to build asynchronous, lockless, and high performance applications.

For each thread, SPDK uses an io channel to represent the channel for accessing an I/O device. I/O requests issued to the block device will be forwarded to the underlying physical device. In our implementation, io channel corresponds to I/O queue of the underlying NVMe device, the framework builds I/O commands based on I/O requests and submits them to submission queue. It then polls for I/O completion on each queue pair with outstanding I/O to receive completion callbacks. Figure 1 provides a graphical representation of a host application using SPDK block devices to interact with an NVMe device. In the host application, each thread submit

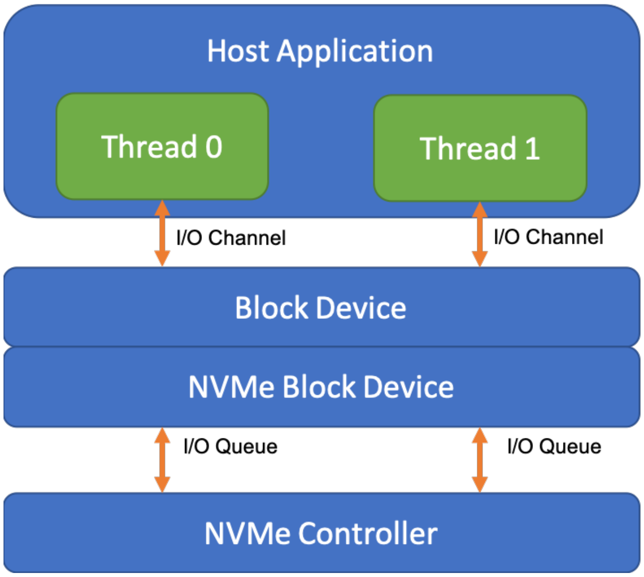

Figure 1: Example of an SPDK application.

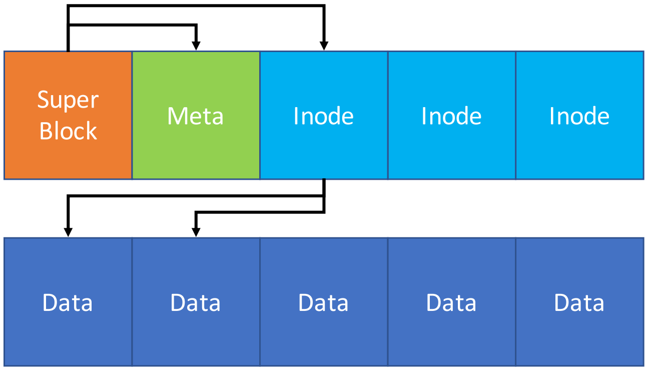

Figure 2: The layout of blocks with different types in testFS.

I/O requests to its corresponding I/O channel and the I/O channel forward the I/O requests to the actual physical device based on the implementation of the physical block device. The framework invokes the callback function when the I/O request is finished.

2.3 TestFS

TestFS [3] is a user space file system which is similar to EXT3, a journaled file system that is commonly used by the Linux kernel. TestFS has three levels of indirections, which is illustrated in Figure 2. A super block points to meta data and inode block of the root directory. The meta data store the freemap of blocks as well as the checksum of the data blocks. Inode blocks point to data blocks where the actual data are stored. In testFS, both directory data and indirect inode data are stored in data blocks. The space for inode blocks and data blocks are pre-allocated and fixed during the life cycle of testFS.

Note that our file system is based on testFS because testFS is well maintained, user level, and well documented file system, that allows us to build a proof of concept quickly. We believe our modifications to testFS can be applied to most of file systems.

3 Project Overview

In this section we provide an overview of our project by enumerating our goals, describing the primary challenges we faced, and highlighting the key ideas behind our approach to addressing these challenges to meet our goals.

3.1 Project Goals

The high level goal of our project is to explore the feasibility and performance implications of enabling parallel access to NVMe storage devices in a user space file system. We work toward this goal by integrating SPDK with testFS. More concretely, in our project we aim to:

1. Demonstrate the feasibility of using SPDK to interact with NVMe devices in testFS.

2. Modify the testFS write path to leverage the ability to make asynchronous block writes with multiple threads.

3. Quantify the performance improvements of an asynchronous write path by benchmarking our implementation.

3.2 Key Challenges

Challenge 1: Callbacks and Synchronous Code. Reads and writes to block devices with SPDK are asynchronous operations. SPDK uses callback functions as the mechanism for scheduling code that should run after an asynchronous operation has completed. However testFS is designed with synchronous operations in mind. The file system code is written with the assumption that, after a thread makes an I/O request, the thread will wait until the request completes before executing any additional code. Ultimately, this combination of synchronous code and a library that uses asynchronous callbacks makes it difficult to integrate testFS with SPDK without making invasive changes throughout the testFS codebase.

Challenge 2: Lock Contention. Our project is a proof of concept for next generation file systems that may use hundreds of threads. To ensure the scalability of our ideas, we need to avoid designs that have multiple threads acquiring the same locks to prevent contention among the threads.

3.3 Our Approach: Key Ideas

The key idea behind our approach is to leverage a multi-threaded architecture where each thread has a specific purpose. Specifically, we use three threads: (i) a control thread responsible for executing the file system logic and for interacting with the user through the testFS REPL, (ii) a metadata thread responsible for submitting I/O requests for blocks that are used for file system metadata, and (iii) a data thread responsible for submitting I/O requests for data blocks. This architecture allows us to achieve asynchronous writes with multiple threads without incurring a significant engineering overhead.

Reconciling Callbacks and Synchronous Code. A multi-threaded architecture allows us to use existing thread synchronization techniques as a way to be able to wait for asynchronous requests to complete. The threads responsible for issuing I/O requests are distinct from the control thread, which means callbacks only need to be registered on the lowest level code responsible for issuing read/write requests with SPDK. When asynchronous requests complete, the callback will execute on the I/O request thread and can then "signal" the control thread as needed. We discuss two techniques for thread synchronization in Section 4.1. Ultimately this approach allows us to avoid adding callback functions throughout the entire testFS codebase and therefore allows us to avoid making invasive changes to the file system.

Avoiding Lock Contention. To avoid lock contention, we use thread synchronization techniques that either (i) do not require locks, or (ii) have a lock per thread, to prevent contention among I/O threads. We discuss our proposed thread synchronization techniques in greater detail in Section 4.1.

4 Implementation Details

To implement asynchronous writes with multiple threads in testFS, we added utilities to facilitate thread synchronization and we modified the file system's write path to make it more amenable to asynchronous writes. For a mapping of the key changes to the relevant source code files, please see Appendix B.

4.1 Thread Synchronization

In our multi-threaded architecture, the control thread needs to be able to wait for asynchronous I/O requests to complete. We accomplish this through thread synchronization using shared objects: a primitive future [5] or several semaphores [4]. We describe both approaches, however for the remainder of the report we only reference the primitive future technique as the underlying ideas are similar.

4.1.1 Primitive Futures

We implemented a shared object that we call a future, similar in spirit to futures used in other concurrent programming languages [5]. Our future is a shared `struct` that contains monotonically increasing counter variables—one per thread. Listing 1 shows the definition of our future.

```
struct future {
  volatile size_t counts[NUM_THREADS];
  size_t expected_counts[NUM_THREADS];
};
```

Listing 1: Our primitive future

To use a future, the caller of an asynchronous function will create a future and then pass a pointer to it to the asynchronous function. The key idea is that the `expected_counts` variables keep track of the number of issued asynchronous requests and that the `counts` variables keep track of the number of completed asynchronous requests. The `counts` variable is incremented when an asynchronous request completes. When the two counters are equal, we know that the requests have completed. A thread can wait for a future's asynchronous operations to complete by spinning on the counters—essentially waiting for `expected_counts[i] == counts[i]` for all `i`. We defined a helper function called `spin_wait()` to do this; it takes a pointer to a future and spins on the counters.

Example. Suppose the control thread wants to call an asynchronous function that should be handled by worker thread `i`. The control thread will create a future and then increment `expected_counts[i]`. It then passes a pointer to the future to the asynchronous function. When the asynchronous operation completes on thread `i`, the worker thread will increment `counts[i]`.

Avoiding Locks by Design. Despite the presence of shared data, locks are not needed to use our future. The `counts` variables do not need to be guarded by locks because there is one counter per thread, meaning only one thread writes to a given counter. The `expected_counts` variables are not shared; they are only used by the asynchronous function's calling thread. The `counts` variables are marked as `volatile` to ensure the compiler does not generate code that caches the variables in a register.

4.1.2 Semaphores

Another approach for thread synchronization is to use POSIX semaphores—creating one per thread. The value of the semaphore represents the number of completed asynchronous requests on a given thread. Each time an asynchronous request completes on thread `i`, the thread will post to semaphore `i`. To wait for asynchronous requests to complete, the control thread can wait on the semaphores of the relevant worker threads.

Avoiding Lock Contention. We create one semaphore per thread to avoid lock contention. This ensures only one thread is posting to a given semaphore and only one thread is waiting on a given semaphore.

4.2 Asynchronous I/O

Interface. To implement asynchronous I/O, we added asynchronous analogues of the `read_blocks()` and `write_blocks()` functions called `read_blocks_async()` and `write_blocks_async()`. These `async` functions accept the same arguments as their synchronous counterparts. However callers of these `async` functions also need to pass in a pointer to a future and the thread ID on which the I/O request should be handled. This allows the control thread to direct metadata block requests and data block requests to distinct threads.

Implementation. When these asynchronous read/write functions are called, the future's `expected_count` for the handling thread is first incremented and then the request is sent to the handling thread using SPDK's message passing library. Upon receiving this message, the handling thread then submits the I/O request to the underlying NVMe device using its dedicated I/O channel. When the request completes, a callback is executed on the handling thread. The callback function increments the future's `count` for the handling thread to "notify" the calling thread that the request has completed. Since the I/O request is asynchronous, the calling thread is free to do other work while the I/O request is being handled. When the calling thread needs to wait for the I/O request to be completed it can call `spin_wait()` on the future.

Synchronous Comparison. To be able to compare asynchronous I/O requests and synchronous I/O requests, we modified `read_blocks()` and `write_blocks()` so that they can work with SPDK as well. These functions were implemented by calling their asynchronous counterparts and then immediately waiting on the future. This ensures that the I/O request completes before the function returns.

4.3 Write Path Modifications

Motivation. When we initially added asynchronous operations to file writes in testFS we discovered a number of inefficiencies in the code that led to repeated reads and writes of the same physical blocks. In the existing implementation: (i) an entire bitmap block is flushed to the underlying device each time a single block is allocated; (ii) entire checksum blocks are flushed for each modified checksum; (iii) an entire inode block is flushed for each modified inode; (iv) inode indirect blocks are read, modified, and written to the underlying device for every block appended to the file. Since a write transaction in testFS may consist of writing multiple blocks to more than one file, issuing entire metadata block writes for each modified data block results in unnecessary write amplification because a single metadata block may contain metadata for multiple data blocks. We discovered that these inefficiencies led to poor performance even in our asynchronous implementation. This led us to reimplement the file write path in testFS.

Bulk Metadata Flushes. We made the observation that metadata such as block checksums, the data block freemap, and "opened" inodes are cached in-memory. As a result, it is unnecessary to flush modified metadata blocks to the underlying device each time a new data block is appended to a file. We modified the file write implementation to instead keep track of the dirty cached metadata blocks and then flush them all together in bulk at the end of a transaction. This approach ensured that a metadata block is written to the underlying device at most once per transaction.

In-Memory Indirect Blocks. We reduced the number of extra reads and writes of an inode's indirect block by keeping a copy of the indirect block in memory. The indirect block is loaded into memory lazily (i.e. when it is first requested). This allows subsequent reads and writes to the indirect block to be made to the copy stored in-memory. The modified indirect block is flushed to the underlying device together with the in-memory copy of the inode.

5 Evaluation

The goal of our evaluation is to quantify the performance improvement associated with using asynchronous I/O requests with multiple threads on the testFS file write path over synchronous I/O requests.

Since testFS is a toy file system, we did not run any end-to-end file system benchmarks such as `fio`, or Filebench [6]. As a result, we do not make claims as to the performance improvement associated with real workloads. Through our experiments we instead aim to quantify the potential performance benefit for adopting multi-threaded asynchronous I/O for NVMe devices in production-ready user space file systems.

5.1 Experimental Setup

We ran our experiments on a machine[1] with a 14-core Intel Xeon 2.40 GHz CPU and 128 GB of memory. The machine was equipped with two Seagate PLC Nytro 5000 NVMe SSDs.

[1] We used `mel-12`.

We used one of these SSDs, with a capcity of 400 GB, in our experiments. Our code was linked with the version of SPDK at git hash a2bf3cded.

In each experiment we measured the amount of time it took to complete a given task up to a granularity of microseconds. We repeated each experiment five times and used the average of all five trials in our results. We used primitive futures (described in Section 4.1.1) as the thread synchronization mechanism in all of our experiments.

6 Conclusion

TODO

Acknowledgments

We would like to thank Shehbaz Jaffer for his guidance and feedback on our project. We also thank the Systems and Networks Lab at the University of Toronto for providing access to hardware for our experiments.

References

[1] SPDK. https://spdk.io.

[2] Specifications NVMe. https://nvmexpress.org/resources/specifications/.

[3] TestFS filesystem. https://github.com/shehbazj/testfs.

[4] Edsger W. Dijkstra. Cooperating sequential processes. Technical report, Technological University, Eindhoven, Netherlands, 1965.

[5] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 7(4):501–538, October 1985.

[6] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

A Reproducing our Results

B Code Walkthrough