

The Australian National University
2600 ACT | Canberra | Australia



Australian
National
University

School of Computing

College of Engineering, Computing
and Cybernetics (CECC)

Java File I/O Performance Comparison on Random Access to NVMe SSD

— 12 pt research project (S1/S2 2024)

A report submitted for the course
COMP8755, Individual Computing Project

By:
Yuki Misumi

Supervisor:
Dr. Shoaib Akram

November 2026

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the [Academic Integrity Rule](#);
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the Epigeum Academic Integrity modules prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

November (2026), Yuki Misumi

Abstract

Modern block-addressable NVMe SSDs provide a similar performance both in random and sequential access, which gives an optimisation opportunity for a Java-based key-value at its architecture level. In combination with the recent Java features such as Foreign Function & Memory API and Linux's new asynchronous IO interface `io_uring`, we compare the random access performance of various IO methods in Java.

The results show that `io_uring` with direct IO via Java Native Interface outperforms `FileChannel#read/write`. We conclude that `io_uring`, in particular with direct IO, has a great potential to compete with the existing Java IO methods for the next Java-based key-value stores with a few areas to further investigate in future research.

Table of Contents

1	Introduction	1
2	Background	3
2.1	Random Access Performance on NVMe SSDs	3
2.2	HaloDB	3
2.3	Recent Java Developments	3
2.3.1	Foreign Function & Memory API	4
2.3.2	Virtual Threads	5
2.4	io_uring	5
2.4.1	io_uring Overview	5
2.4.2	System Call Minimisation	6
3	Methodology	7
3.1	Hardware Setup	7
3.2	Benchmark Set	7
3.2.1	T01 — FileChannel#read/write	7
3.2.2	T02 — FileChannel#map	8
3.2.3	T03 — io_uring	8
3.2.4	T04 — AsynchronousFileChannel	9
3.3	Tools Used	9
3.3.1	Java Development Kit	9
3.3.2	Java Microbenchmark Harness	9
3.3.3	Vmtouch	10
3.3.4	Async-profiler	10
3.4	Benchmark Program Overview	10
3.5	Benchmark Execution	11
4	Evaluation	13
4.1	T01 — FileChannel#read/write	13
4.1.1	Direct Byte Buffer Performance	13
4.1.2	Direct IO Performance	14
4.2	T02 — Memory Mapped IO via FileCannel#map	15

Table of Contents

4.3	T03 — Asynchronous IO via io_uring	16
4.3.1	Direct IO Performance	16
4.3.2	JNI vs. FFM	18
4.4	Multi-thread Performance	19
4.4.1	T01 — FileChannel#read/write	20
4.4.2	T02 — FileChannel#map	20
4.4.3	T03 — io_uring	21
4.4.4	T04 — AsynchronousFileChannel	21
4.5	Overall Performance	22
5	Concluding Remarks	25
5.1	Conclusion	25
5.2	Future Works	25
	Bibliography	27

Introduction

Modern block-addressable NVMe SSDs provide a similar performance both in random and sequential access ([Lepers et al., 2019](#)). With the context in mind, Java-based key-value stores should take this into account by design. Therefore, in this project, we compare various file IO methods in Java and identify the most effective one in random access.

In the recent Java development, Foreign Function & Memory API is introduced to access off-heap memory with better control over the memory life-cycle than direct byte buffers and invoke native functions as an alternative Java Native Interface. In the Linux world, `io_uring` was introduced at the kernel version 5.1 in 2019 as a new asynchronous IO interface ([Corbet, 2020](#)), minimising the number of system calls.

Our benchmark program runs an approximated random access to an NVMe SSD with different implementation variants, including not only the said recent Java features and `io_uring` interface but also traditional approaches, such as `FileChannel#read/write` and `FileChannel#map`. We see that `io_uring` with direct IO via Java Native Interface outperforms `FileChannel#read/write` and that Foreign Function & Memory API gives a bigger overhead in our particular workload.

We conclude that `io_uring`, in particular with direct IO, has a great potential to compete with the existing Java IO methods. Therefore, it is worth considering to choose `io_uring` in the future Java-based key-value store.

The contributions of this project are:

1. Showing the effectiveness of direct byte buffer.
2. Showing different `io_uring` configurations and their effect in random access to NVMe SSDs.

1 Introduction

3. Demonstrating that Java Native Interface can still be a faster choice than Foreign Function & Memory API in random access to NVMe SSDs.

Background

2.1 Random Access Performance on NVMe SSDs

[Lepers et al. \(2019\)](#) shows that a 480GB Intel Optane 905P, released in 2018, has little difference in the IOPS and bandwidth between random and sequential access, whereas, in the older SSDs, random writes are much slower than sequential counterparts. This indicates that a new Java-based key-value store based on this finding may emerge. With the context in mind, this project explores efficient Java IO methods for random access.

2.2 HaloDB

HaloDB is a Java-based log structured key-value store, introduced by Yahoo in 2018 ([Mannaly, 2018](#)). The architecture supposes sequential access performs better than random access, and uses `FileChannel#read/write` for file access ([Mannaly](#)). This report looks at different IO methods to determine whether the recent Java features outperform it, which are discussed in the next section.

2.3 Recent Java Developments

Java is a language of choice for many programmers today as it is ranked 4th in the top languages in Octoverse 2023 ([Daigle and Staff, 2023](#)) and continues evolving — the latest release by Oracle as of writing is Java 23 ([Oracle, d](#)). In this project, we use Java 21 throughout all the benchmarks. In this section, we present an overview of Foreign Function & Memory API and Virtual Threads which are used in our benchmark.

2 Background

2.3.1 Foreign Function & Memory API

Foreign Function & Memory API (hereafter, FFM) is a feature to access the data and code outside of the Java runtime and an alternative to Java Native Interface (hereafter, JNI), addressing its error-prone nature. The FFM is a preview API in Java 21 ([OpenJDK, 2023a](#)) and finalised in Java 22 ([OpenJDK, 2023c](#)).

Accessing Foreign Memory

When objects are created using *new* keyword, they are allocated in the JVM's heap, which is subject to garbage collection. Instead, performance critical applications tend to use off-heap memory to bypass the garbage collection. Historically, Java has supported the access to the off-heap memory by ByteBuffer API in the form of direct byte buffers. Although they are intended for interacting with native I/O functions ([Hitchens, 2009](#)), the maximum size of the buffer is limited to 2GB because of its indexing scheme using a 32-bit value. Moreover, Java programmers do not have control over deallocating the backing memory — the deallocation relies on garbage-collecting the corresponding direct byte buffer object ([OpenJDK, 2023c](#)).

The FFM API provides access to off-heap memory via the MemorySegment interface, which now allows expressing more than 2GB of memory. The programmers can manage the life-cycle, including deallocation, via the Arena interface using e.g. try-with-resources block ([Oracle, e](#)).

Invoking Foreign Function

JNI provides a way to invoke a native function implemented in languages such as C and C++. However, as development experience, JNI has an error-prone workflow. To invoke a function implemented in C, JNI requires the following:

1. Declare a Java method annotated with *native* keyword
2. Generate a C header file using *javac -h* command
3. Implement the native function in the corresponding C file
4. Generate the shared object with a C compiler e.g. *gcc*
5. Load the shared object in Java code with *System#loadLibrary*
6. When running, add *-Djava.library.path=/path/to/shared-object* JVM argument

Since there is no strict type check between the Java method and the corresponding C function, and no change in Java code is automatically reflected in the native code, it is very easy to overlook small mistakes, leading to a runtime exception because of the unsatisfied links. As a matter of fact, because of these challenges, we have faced many bugs during the implementation of native functions to bridge the liburing library.

Furthermore, passing Java objects through the JNI is slow, so Java programmers are tempted to allocate everything off-heap e.g. via Unsafe API and pass the address as a long value, which is unsafe ([OpenJDK, 2023c](#)).

The FFM waives the need for a glue code to bridge Java method and native function and provides the following simpler workflow ([Oracle, a](#)):

1. create *SymbolLookup* object with a shared object file path
2. obtain an address of a native function
3. create *FunctionDescriptor* object describing the function signature
4. create *MethodHandle* object with the address and signature
5. *MethodHandle#invokeExact* calls the native function

Noticeably, all the steps are closed in Java code thus friendly to Java programmers. This contributes to the safety of the FFM APIs because with the notion of memory segment and arena, most use cases of the JNI are supported in Java code and thus Java programmers no longer need to write native code ([OpenJDK, 2023c](#)).

2.3.2 Virtual Threads

Historically, an instance of *java.lang.Thread* is a platform thread, capturing the underlying OS thread until the corresponding Java code finishes. Therefore, they are only available for the number of OS threads ([OpenJDK, 2023b](#)).

Since Java 21, a virtual thread is finalised. While it is still an instance of *java.lang.Thread* and running Java code on the OS thread, it is not monopolising it, unlike a platform thread. Therefore, the number of virtual threads is not limited to that of OS threads ([OpenJDK, 2023b](#)).

2.4 *io_uring*

2.4.1 *io_uring* Overview

io_uring is an asynchronous IO interface, introduced at the Linux kernel version 5.1 ([Corbet, 2020](#)). Linux historically supports a native asynchronous IO interface, *aio*, but is not widely used because of several limitations, including only supporting direct IO and requiring 104 bytes of memory copy for IO submission and completion ([kernel.dk, 2019](#)). *io_uring* supports both non-direct IO and direct IO and shares the IO data structures between the application and the kernel.

2 Background

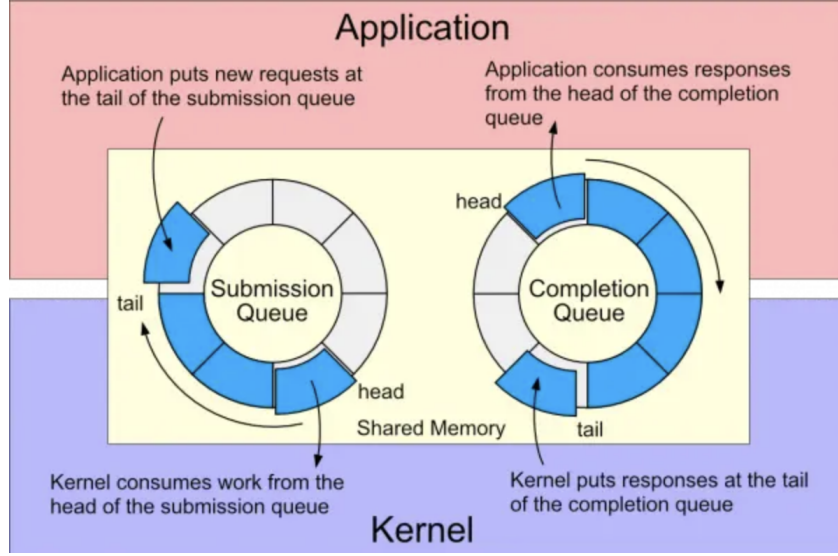


Figure 2.1: An Overview of io_uring Operation (Hunter, 2023)

The core of the io_uring interface is the 2 ring buffers, submission queue (hereafter SQ) and completion queue (hereafter, CQ). The application is expected to submit an IO request to the SQ as the producer, which the kernel gets from it and processes as the consumer and the opposite relationship applies for completing the request with the CQ, as illustrated in Figure 2.1.

The number of both submission queue entries (hereafter, SQEs) and completion queue entries (hereafter, CQEs) is configurable at `io_uring_setup` system call via its `entries` parameter, which sets up an io_uring instance. The number must range from 1 to 4096 (inclusive) for every power of 2.

2.4.2 System Call Minimisation

Another benefit io_uring brings is the number of required system calls. As the overhead of system calls is becoming significant (kernel.dk, 2019), reducing the invocation positively contributes to the application performance.

While aio requires at least 2 system calls to submit (i.e. `io_submit`) and wait for completion (i.e. `aio_waitcomplete`), io_uring only requires 1 system call, `io_uring_enter`, to notify the kernel of the new SQEs to consume. In addition, it can be eliminated further with the kernel side polling feature. With this enabled, a kernel thread is launched to poll SQEs, and the application can submit and complete the IO requests without a single system call (Axboe, 2019).

Methodology

3.1 Hardware Setup

All the benchmarks are conducted on the following same hardware and operating system:

- Operating System: Ubuntu 22.04.2 LTS with the kernel version 5.15.0-119-generic.
- Processor: Intel(R) Xeon(R) Gold 6330N CPU clocked at 2.20GHz. It contains 2 sockets, each socket contains 28 cores, and each core contains 2 threads, thus consisting of 112 CPU cores in total.
- DRAM: 503GB.
- NVMe SSD: SAMSUNG MZPLJ3T2HBJR-00007 with the size of 3.2TB, released in 2019. The sequential read and write rate is 8000 MB/s and 3800 MB/s respectively. For 4 KB random read and write, it gives 1,500,000 IOPS and 250,000 IOPS respectively ([Samsung](#)).

3.2 Benchmark Set

To effectively experiment with random access performance with different IO methods in a structured manner, we categorise them into the following types.

3.2.1 T01 — FileChannel#read/write

New I/O (hereafter, NIO) APIs are introduced in Java 1.4. They address the challenge of the existing stream-based IO model — it likes to operate on single bytes, whereas the operating systems efficiently move data in large chunks ([Hitchens, 2009](#)).

In this type, we use one of the APIs, FileChannel#read/write. It supports both a non-direct and direct byte buffer. The FileChannel object can be created with the direct IO

3 Methodology

flag. To identify the effect of these, we enumerate the following benchmark IDs out of the T01 type.

Table 3.1: T01 variants

ID	Byte Buffer	Uses Direct IO
T01A	Non-direct	No
T01B	Direct	No
T01C	Non-direct	Yes
T01D	Direct	Yes

We will look at the results between T01A and T01B to see the byte buffer effect and T01A and T01C for the effect of the direct IO flag.

3.2.2 T02 — FileChannel#map

The NIO package also supports memory-mapped IO. Historically, this is implemented using direct byte buffers. However, since Java 19, it provides another implementation using memory segment from the FFM API as a preview feature ([Oracle, b](#)) (this is finalised as of Java 22 ([Oracle, c](#))). We will test their performance difference as enumerated below.

Table 3.2: T02 variants

ID	ByteBuffer or FFM
T02A	ByteBuffer
T02B	FFM

3.2.3 T03 — io_uring

In this type, we will examine how much additional overhead is incurred by the JVM from the native implementation. To achieve it, we implement a program responsible for the variants T03AN and T03BN, which reads/writes from/to the given file in C using liburing library (version 2.8) developed by [Axboe](#), a wrapper of io_uring interfaces. We will also test how different SQ sizes (i.e. 1 – 4096 (inclusive) for every 2^n) affect the performance.

Table 3.3: T03 variants

ID	JNI, FFM or Native	Uses Direct IO	Kernel Side Polling
T03AN	Native	No	No
T03BN	Native	Yes	Yes
T03A	FFM	No	No
T03B	FFM	Yes	Yes
T03BP	FFM	Yes	No
T03C	JNI	No	No
T03D	JNI	Yes	Yes
T03DP	JNI	Yes	No

T03BP and T03DP are the additional variants for T03B and T03D to segregate the effect of kernel side polling.

Regarding the Java implementation, we use the JNI and FFM to down-call each liburing function. Then, we will see which Java implementation provides the smallest overhead to invoke native functions in the particular setup.

3.2.4 T04 — AsynchronousFileChannel

The NIO package also provides `AsynchronousFileChannel` which handles IO requests with a thread pool. We use this method in the multi-thread benchmark setting for the following variants.

Table 3.4: T04 variants

ID	Byte Buffer	Uses Direct IO
T04A	Non-direct	No
T04B	Direct	No
T04C	Direct	Yes

3.3 Tools Used

3.3.1 Java Development Kit

We use OpenJDK 21.0.2 for all the benchmarks in Java.

3.3.2 Java Microbenchmark Harness

Java Microbenchmark Harness (hereafter, JMH) is a Java framework provided by OpenJDK to build, run, and analyse various benchmarks written in any JVM languages

3 Methodology

([OpenJDK](#)). JMH provides a granular control on what benchmarks to run with user-defined parameters and JVM configurations. It also supports running a benchmark for an arbitrary number of iterations, including so-called warmups — dry runs which are not counted towards the benchmark result. We use this framework as a base for our Java benchmark program.

3.3.3 Vmtouch

Vmtouch is a tool to control the file system page cache in Unix/Unix-like systems ([Hoyte](#)). One of the use cases is to tell the operating system to evict the specified files from the page cache. We use it to evict the entire pages of the target file under the benchmark for each iteration.

3.3.4 Async-profiler

Async-profiler is a Java profiler supporting to trace CPU cycles, hardware and software counters, Java heap allocations and contended lock attempts ([Pangin](#)). We use this to analyse the benchmark results in detail.

3.4 Benchmark Program Overview

In our Java program, we define the following 4 tests for each variant in the benchmark set:

- read: reads randomly from the given file for its size, using platform threads in multi-thread configuration.
- readVirtual: the same as the read benchmark, but using virtual threads.
- write: the same as read benchmark, but for random writes.
- writeVirtual: the same as readVirtual, but for random writes.

For example, if we run a read test for T01A with 1GB file, it will read randomly for 1GB from the file with `FileChannel#read` and non-direct byte buffer.

Since purely random access will result in page cache hit and make the test unreliable for each run, we make it deterministic by approximating the pattern as depicted in [Figure 3.1](#).

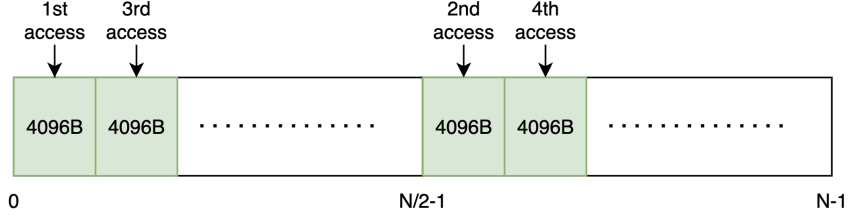


Figure 3.1: Random Access Approximation

Where N is the number of bytes contained in a file, this is designed to access by the following order:

1. access a page at index 0
2. access a page at index $N/2 - 1$
3. access a page at index 4096
4. access a page at index $N/2 - 1 + 4096$
5. ... and so on.

For the native variants i.e. T03AN and T03BN, we make a C program equivalent to the JMH-based program, following the same access pattern approximation.

3.5 Benchmark Execution

We run the benchmarks for each ID in the following 2 settings to see the scaling effect.

1. Single-thread, varying file sizes ($1 - 16\text{GB}$ (inclusive, per 2^nGB)).
2. Multi-thread (4 and 16 threads), 1GB file size.

For the multi-thread setting, we run each benchmark with platform and virtual threads to observe any performance implications. For T04, we set a thread pool of the same size with `Executors.newFixedThreadPool` instead. Each thread is assigned a dedicated 1GB file.

Each benchmark is configured to run for 5 warmups to ensure JVM processes their just-in-time optimisation and then 100 iterations to amortise the hardware glitches on a dedicated JVM fork. In doing so, we are expected to obtain more stable results. Because of the execution time and time constraint, some variants are tested differently as an exception as listed below.

- T01C, T01D for single-thread setting: tested with $1 - 8\text{GB}$ (inclusive) files
- T01C, T01D, T04C for multi-thread setting: tested for 5 warmups and 20 iterations

3 Methodology

Before each iteration, we run the following `vmtouch` command and evict the file under benchmark to avoid the caching effect which would otherwise impact the result of the next iteration.

```
vmtouch -eq /path/to/file
```

All the operations are automated by a shell script. Regarding the performance metrics, we compute the mean and P99 throughput (MB/s) and CPU usage (100% indicating a full core usage). The throughput is given by:

$$\text{Throughput} = \frac{\text{File Size (MB)} \times \#\text{Threads}}{\text{Execution Time (sec)}} \quad (3.1)$$

Evaluation

4.1 T01 — FileChannel#read/write

In this section, we evaluate T01 – T01D. In doing so, we aim to segregate the effectiveness of direct byte buffer and direct IO with the Java IO method.

4.1.1 Direct Byte Buffer Performance

While passing non-direct byte buffer to FileChannel#read/write causes a copy of it for the corresponding native function, doing so for direct byte buffer is not the case since it is allocated off-heap, accessible both from JVM and native functions. Therefore, it is expected that T01B provides better throughput than T01A by bypassing the memory copies.

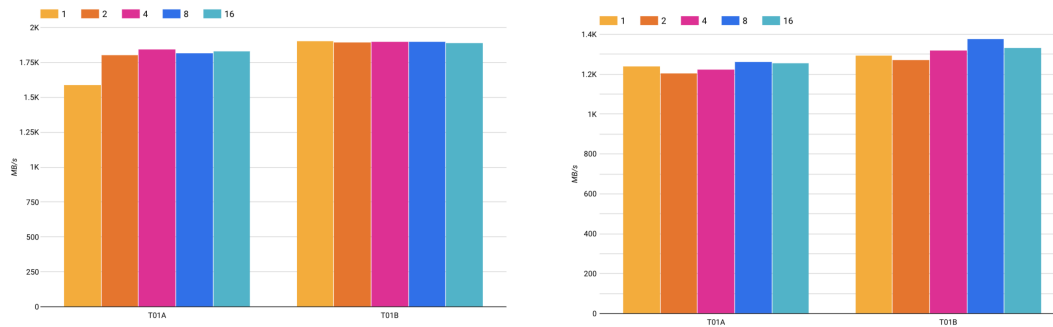


Figure 4.1: Mean Random Read (Left) and Write (Right) Throughput Between T01A and T01B for 1–16 GB File

4 Evaluation

Table 4.1: Mean Throughput Improvement (%) from T01A to T01B

GB	Read %	Write %
1	19.62	4.33
2	4.97	5.80
4	2.97	7.87
8	4.52	9.08
16	3.18	6.17

From Figure 4.1 and Table 4.1, we see an improvement in throughput for both in read and write between T01A and T01B. it is clear that, though slight, we see a positive effect of using direct byte buffer in all file sizes tested.

4.1.2 Direct IO Performance

Since direct IO bypasses page cache, doing it is expected to reduce the overhead, thus providing better throughput. In T01C and T01D, we attempt to obtain the effect of direct IO in FileChannel#read/write.

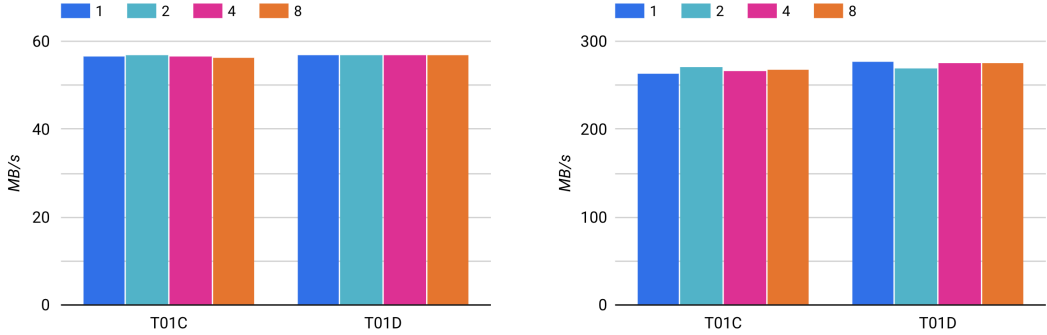


Figure 4.2: Mean Random Read (Left) and Write (Right) Throughput Between T01C and T01D for 1–8 GB File

As apparent in Figure 4.2, the throughput in T01C and T01D is significantly decreased from T01A and T01B respectively in all file sizes. To investigate the behaviour further, we use async-profiler to profile how much time T01C and T01D take for each method invocation on a 1GB file.

4.2 T02 — Memory Mapped IO via `FileChannel#map`

Table 4.2: Wall or CPU time taken for T01C, T01D on 1GB file

ID	Method	Wall time (sec)	CPU time (sec)
T01C	<code>--GI__pread</code>	18.05	0.70
T01C	<code>--GI__pwrite64</code>	3.70	0.71
T01D	<code>--GI__pread</code>	17.95	0.63
T01D	<code>--GI__pwrite64</code>	3.75	0.63

Regarding read access, in T01C, the profiling result shows that `--GI__pread`, invoked ultimately by `FileChannel#read`, takes 0.70 seconds in CPU time but nearly 18 seconds in wall time. T01D shows a similar result as listed in Table 4.2, while neither T01A nor T01B exhibits a disparity of the same level, indicating they suffer from waiting for IO processing. Although the test is random access, it suggests that we possibly see an effect of readahead.

Since direct IO bypasses page cache, the kernel reads exactly the pages requested by a user program. However, in the case of non-direct IO, the kernel reads more than the requested pages and stores them in the page cache to aid future read requests. Therefore, T01C and T01D likely suffer from more frequent IO requests to storage, while T01A and T01B hit page cache stored by readahead.

Regarding write access, in both T01C and T01D, we also see a large disparity in CPU and wall time in `--GI__pwrite64`, invoked ultimately by `FileChannel#write`. Since direct IO bypasses page caches, each write system call causes a write directly into storage. On the other hand, non-direct IO such as T01A and T01B can first write to page caches and then flush them into storage in bulk. This suggests the large throughput difference between direct IO and non-direct IO throughput.

4.2 T02 — Memory Mapped IO via `FileChannel#map`

Since Java 19, `FileChannel` provides an alternative memory map IO method using the FFM API ([Oracle](#), [b](#)). We compare the throughput of this method (T02B) with the traditional `FileChannel#map` method (T02A) using a direct byte buffer.

4 Evaluation

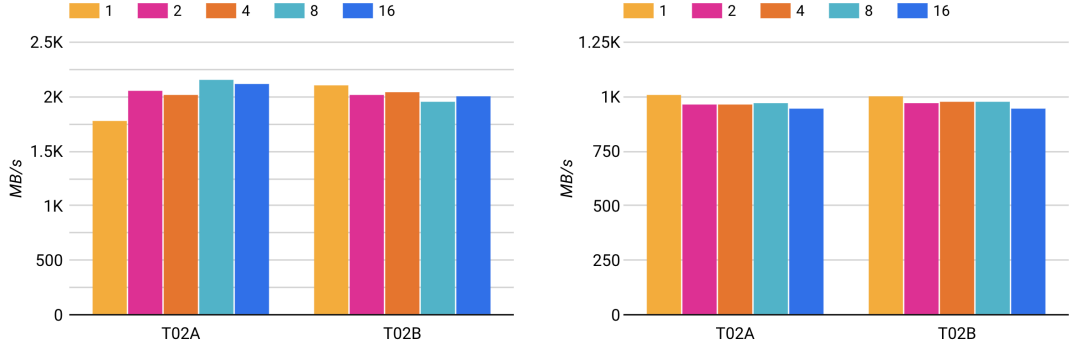


Figure 4.3: Mean Random Read (Left) and Write (Right) Throughput Between T02A and T02B for 1–16 GB File

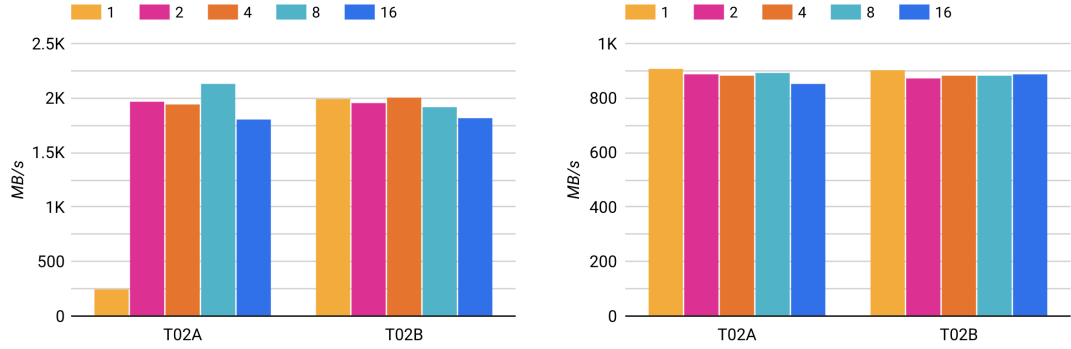


Figure 4.4: P99 Random Read (Left) and Write (Right) Throughput Between T02A and T02B for 1–16 GB File

In Figure 4.3, we observe mostly an identical throughput in write between the 2 variants for all file sizes. Interestingly, when we see the read throughput of P99, T02A presents significantly less throughput (87.30%) than T02B in 1GB, suggesting unstable behaviour although the same trend is not observed with the increased file sizes.

Therefore, the results suggest that the FileChannel#map using the FFM API is more stable than the traditional map method.

4.3 T03 — Asynchronous IO via io_uring

4.3.1 Direct IO Performance

As we see previously in Figure 4.2, direct IO may adversely affect in random access performance. Considering io_uring is asynchronous by its nature, we experiment whether

4.3 T03 — Asynchronous IO via io_uring

it gives any different effect by measuring the throughput of T03A (non-direct IO), T03BP (direct IO) and T03B (direct IO with kernel side polling).

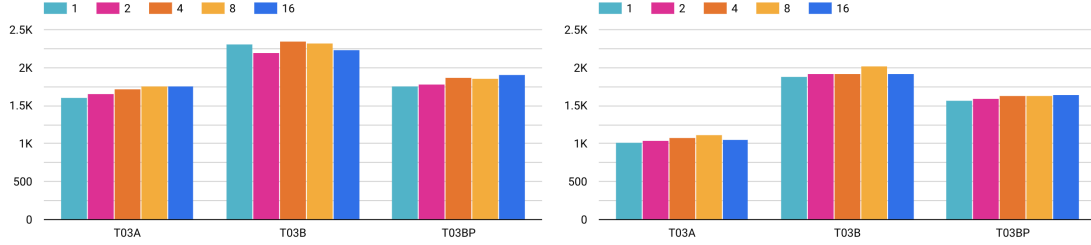


Figure 4.5: Mean Random Read (Left) and Write (Right) Throughput Between T03A, T03BP and T03B for 1 – 16GB File

Table 4.3: Mean Throughput Improvement (%) from T03A to T03BP and T03B on 1GB File

GB	T03BP Read %	T03BP Write %	T03B Read %	T03B Write %
1	9.65	53.19	43.59	82.96
2	7.40	53.48	32.63	84.91
4	9.26	51.00	37.07	77.66
8	6.21	47.40	32.51	81.58
16	8.77	56.34	26.79	82.64

As apparent in Figure 4.5, a clear throughput gain is observed by direct IO (T03BP). In the case of 4GB data, we see an improvement of 9.26% in read and 51.00% in write. This shows that, in io_uring, the overhead incurred by lack of page cache is hidden by simultaneously accessing data and that the benefit of bypassing page cache outweighs it. With scaling file sizes, we see a similar throughput gain, as shown in 4.3. Furthermore, adding kernel side polling (T03B) results in more throughput gain (37.07% and 77.66% respectively), which indicates the benefit of direct IO.

However, it must be noted that the CPU usage fluctuates widely as shown in Table 4.4. Io_uring first tries to execute an SQE in a non-blocking manner, and when it fails, it creates a thread to handle the entry (Axboe, 2019). This explains the higher CPU usage in T03A and T03C write. On the other hand, T03B and T03D indicate the CPU usage of around 2 cores. Since they use kernel side polling, the kernel launches an additional thread dedicated to processing the SQEs, which explains the behaviour (Axboe, 2019). Therefore, the comparison is not entirely fair in terms of CPU utilisation.

4 Evaluation

Table 4.4: T03 Average CPU Usage (%) by Random Access for 4GB File.

ID	Read %	Write %
T03A	105.10	334.69
T03B	223.73	214.06
T03BP	105.50	97.16
T03C	100.13	308.62
T03D	219.24	208.85
T03DP	100.12	89.71

For T03BP and T03DP, the CPU usage ranges around 1 core, indicating SQE processing is done in a non-blocking manner without fail. It is interesting to see that they provide much better throughput than non-direct IO approaches (T03A, T03C), in particular in write, where they surpass over 300% utilisation. This also supports the positive effect of direct IO.

4.3.2 JNI vs. FFM

In Java 21, official support for `io_uring` is not provided, likely because it is a platform-dependent, asynchronous IO interface by Linux kernel. Thus, it is essential to use either the JNI or FFM to manually invoke any of the functions from Java. In this subsection, we run both implementations and identify which foreign function interface gives the closer throughput to the native implementations in C.

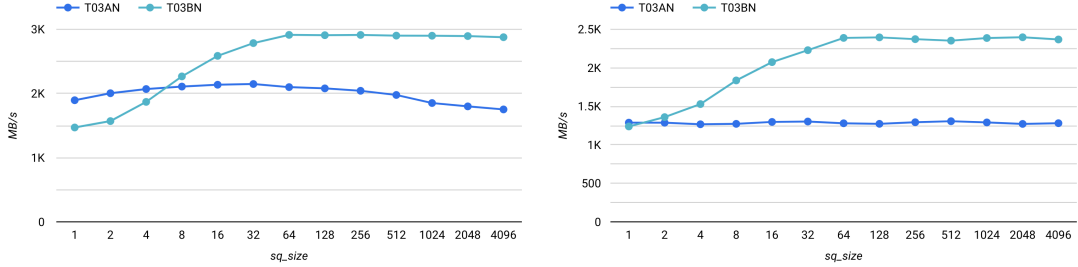


Figure 4.6: Mean Random Read (Left) and Write (Right) Throughput Between T03AN and T03BN for 1 GB File Across 1–4096 SQEs

As shown in Figure 4.6, T03BN (direct IO with kernel side polling) scales the throughput as the SQ size increases up until 64.

On the other hand, T03AN (non-direct IO) in read gradually increases the throughput until the SQ size of 32 and then gradually decreases thereafter. The write does not have a particular scaling trend in the SQ size although the SQ size of 32 and 512 gives the top throughput. Since all the write requests in T03AN lead to storing the data on page

4.4 Multi-thread Performance

cache and they are flushed at the end of the program in bulk by *fsync* system call, it is possible that T03AN does not benefit the asynchronous nature of *io_uring*.

Now that we understand the performance of the native implementations, we compare them with the 2 implementation groups in Java, the FFM (i.e. T03A, T03B) and the JNI implementations (i.e. T03C, T03D) as presented below.

Table 4.5: Throughput Ratio Against Native Implementation on 1GB File and 32 SQEs. Each ratio is given by $\frac{T03X}{T03XN}$ where $X \in \{A, B\}$. For T03C and T03D, we use *A* and *B* for *X* respectively.

ID	Read MB/s	Read Ratio	Write MB/s	Write Ratio
T03AN	2148.02	-	1302.9	-
T03BN	2785.61	-	2283.14	-
T03A	1606.26	74.78%	1022.13	78.45%
T03B	2306.37	82.80%	1888.54	82.72%
T03C	1913.25	89.07%	1222.34	93.82%
T03D	2706.53	97.16%	2236.34	97.95%

Compared with the FFM implementations (i.e. T03A, T03B), it is clear that the JNI implementations (i.e. T03C, T03D) are significantly closer to the native implementations. This suggests that, while the FFM provides a much cleaner interface to native functions, the overhead of invoking them is larger than JNI in this particular workload.

However, it should be noted that the JNI is not necessarily more efficient than the FFM for all cases. In the JNI implementations, we avoid passing Java objects where possible but an address of native memory as a long type to save the overhead. This optimisation may have contributed to their small overhead. For instance, although it is Java 22, [Cimadamore \(2024\)](#) shows that quick-sorting an integer array using the *qsort* function in *stdlib.h* performs much more efficiently (more than 2x faster) with the FFM than JNI in latency.

Considering our results, when we only down-call native functions and avoid passing Java objects, it is likely that the JNI provides less overhead than the FFI. However, a further analysis is necessary to determine the exact implication.

4.4 Multi-thread Performance

In this section, we look at how each benchmark ID behaves both in the platform and virtual threads with the increasing thread size. Due to the time constraint, the evaluation does not go into great depth, requiring further exploration with potentially more massive parallelisation in future research.

4 Evaluation

4.4.1 T01 — FileChannel#read/write

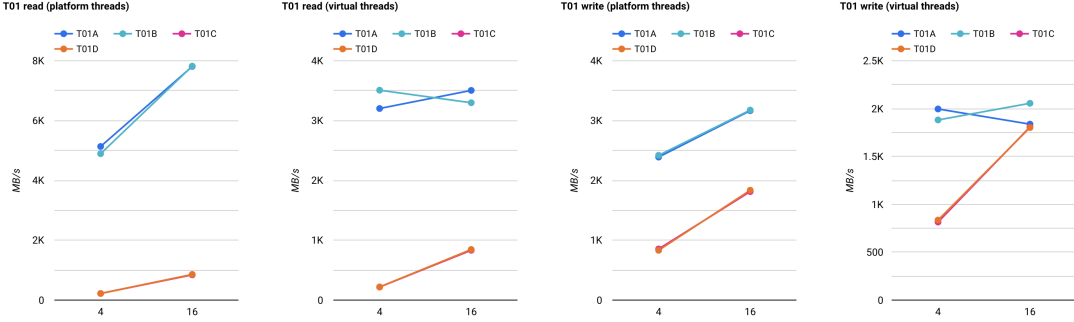


Figure 4.7: Mean Random Access Throughput on T01 for 1GB File with 4, 16 Threads

In Figure 4.7, we see that T01's throughput scales with platform threads. On the other hand, in virtual threads, T01B and T01A slightly degraded in read and write respectively. Furthermore, the overall throughput is much less than the platform threads counterparts, suggesting the effectiveness of platform threads with FileChannel#read/write.

4.4.2 T02 — FileChannel#map

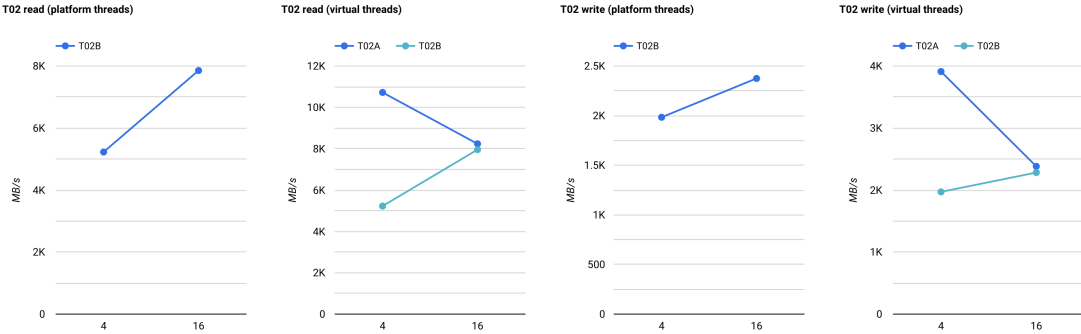


Figure 4.8: Mean Random Access Throughput on T02 for 1GB File with 4, 16 Threads. Unfortunately, the T02A's platform threads implementation gives segmentation fault during the test, which cannot be addressed under the time constraint.

On the contrary, as shown in Figure 4.8, T02 presents better throughput with virtual threads. Interestingly, T02A decreases throughput with increasing virtual threads. However, with the program failure and the fact that T02A read in virtual threads surpasses the specification of the NVMe SSD under benchmark, we need a further investigation for validity.

4.4.3 T03 — io_uring

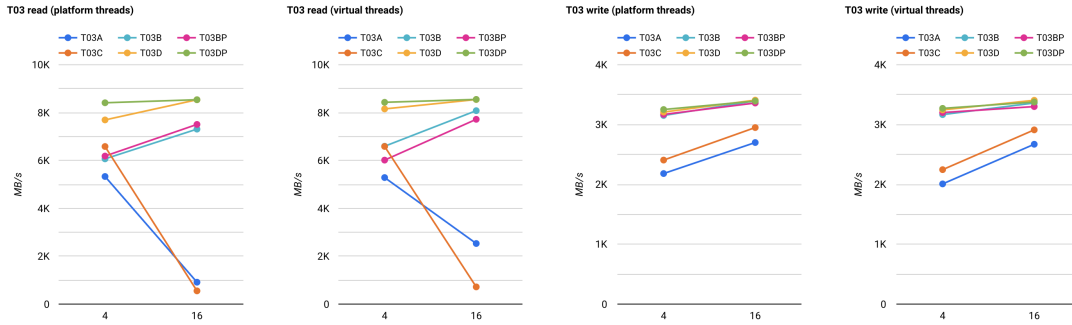


Figure 4.9: Mean Random Access Throughput on T03 for 1GB File with 4, 16 Threads

Interestingly, Figure 4.9 shows a large throughput degradation in read for T03A (non-direct IO with the FFM) and T03C (non-direct IO with the JNI). The figure also shows similar a throughput change for each benchmark ID. This suggests that the choice between the FFM and JNI in the context of a foreign function interface affects little to the throughput.

4.4.4 T04 — AsynchronousFileChannel

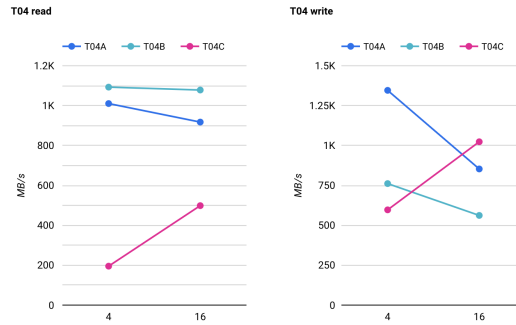


Figure 4.10: Mean Random Access Throughput on T04 for 1GB File with 4, 16 Threads

Unfortunately, the NIO's asynchronous IO interface does not present a throughput scaling with our benchmark setting, except for T04C (direct IO) possibly because of the IO operation overlaps. In this benchmark, we only use a fixed-sized thread pool. More thread pool configurations should be tested to evaluate its potential.

4.5 Overall Performance

In this section, we look at the performance results across different IO methods discussed above to determine which approach provides the best performance. For this purpose, we show a consolidated Figure 4.11 containing single-thread, random access throughput for a 4GB file.

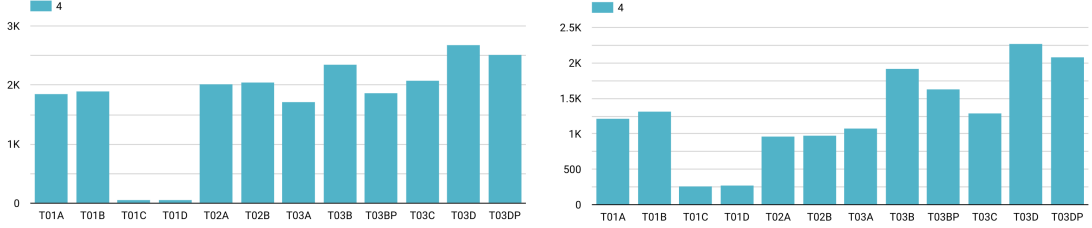


Figure 4.11: Mean Random Read (Left) and Write (Right) Throughput for 4 GB File

Table 4.6: Average CPU Usage (%) by Random Access for 4GB File.

ID	Read %	Write %
T01A	84.28	85.87
T01B	81.87	85.19
T01C	6.02	28.11
T01D	5.48	26.66
T02A	61.66	96.56
T02B	61.19	96.15
T03A	105.10	334.69
T03B	223.73	214.06
T03BP	105.50	97.16
T03C	100.13	308.62
T03D	219.24	208.85
T03DP	100.12	89.71

Figure 4.11 clearly shows that T03D (io_uring with direct IO and kernel side polling) provides the best throughput. In particular, it provides 41.47% and 72.36% more throughput than T01B (direct byte buffer) in read and write respectively. However, as already discussed and shown in Table 4.6, it has much more than 100% CPU usage because of an additional thread by kernel side polling, making the comparison not entirely fair.

On the other hand, T03DP (io_uring with direct IO) is the second best and stays around 100% CPU usage. It provides 32.25% and 57.79% more throughput than T01B in read and write respectively. Therefore, it clearly shows that io_uring potentially outperforms

FileChannel#read/write, which is used in HaloDB.

Apart from direct IO, in read, memory-mapped IO methods (T02A and T02B) are more efficient than FileChannel#read (T01A and T01B). In particular, T02B (FFM) has 7.46% more throughput than T01B (direct byte buffer). In choosing between the traditional FileChannel#map or FFM one, considering the FFM API provides a predictable behaviour in its memory management by design (whereas direct byte buffer does not), it is clear that the FFM's map method (T02B) should be favoured. In write, T01B (direct byte buffer) performed the best (34.65% more throughput than T02B (FFM)). These methods will likely be a good choice where direct IO is not a suitable option.

Concluding Remarks

5.1 Conclusion

In this project, we experimented with and analysed the different IO methods in Java for random access to an NVMe SSD. With the benchmarks conducted, we saw great potential in `io_uring` to compete with the existing Java IO methods for the future Java-based key-value stores, surpassing the throughput of `FileChannel#read/write` and `FileChannel#map`. We also observed that the JNI still is, under a certain circumstance, a very fast interface for native functions, while the non-performance factors brought by the FFM, including safety and developer experience, are undeniably attractive. Through this project, we have identified various areas for further analysis as discussed below.

5.2 Future Works

We see the following items as an area to further explore in future research.

- Although we warmed up the JVM to let it process just-in-time optimisations before the tests, we did not explore how the changes to the JVM configurations (e.g. the memory size, garbage collection strategy) affect the performance.
- We identified the use of multi-threads at the kernel, making the comparison across different IO methods discussed in the chapter 4 less fair. While `IOSQE_ASYNC` flag of `io_uring_enter` function enables the use of additional threads for processing the SQEs ([Axboe, 2019](#)), we could not identify an effective way to disable the behaviour entirely under the time constraint.
- Due to the time constraint, we did not explore all the features. For example, the polled IO feature is worth testing to optimise the completion operation, which performs IO without relying on hardware interrupts to notify of a completion event

5 Concluding Remarks

([kernel.dk, 2019](#)).

- Unfortunately, multi-thread performance lacks insights in great depth as discussed in the chapter 4. As the benchmark execution time is enormous at this point, even with a 1GB file, a different and effective benchmarking approach should be designed in future research, possibly by fixing the total data size among the files under test and decreasing the size of each file.

Bibliography

- AXBOE, J. liburing. <https://github.com/axboe/liburing>. [Cited on page 8.]
- AXBOE, J., 2019. liburing/man/io_uring_enter.2. https://github.com/axboe/liburing/blob/master/man/io_uring_enter.2. [Cited on pages 6, 17, and 25.]
- CIMADAMORE, M., 2024. Foreign function & memory api. https://archive.fosdem.org/2024/events/attachments/fosdem-2024-1714-foreign-function-memory-api/slides/22193/fosdem_2024_FtLDvIv.pdf. [Cited on page 19.]
- CORBET, J., 2020. The rapid growth of io_uring. <https://lwn.net/Articles/810414/>. [Cited on pages 1 and 5.]
- DAIGLE, K. AND STAFF, G., 2023. Octoverse: The state of open source and rise of ai in 2023. <https://github.blog/news-insights/research/the-state-of-open-source-and-ai/>. [Cited on page 3.]
- HITCHENS, R., 2009. *Java NIO*. O'Reilly Media, Inc. <https://learning.oreilly.com/library/view/java-nio/0596002882/>. [Cited on pages 4 and 7.]
- HOYTE, D. vmtouch - the virtual memory toucher. <https://github.com/hoytech/vmtouch>. [Cited on page 10.]
- HUNTER, D., 2023. Why you should use io_uring for network i/o. <https://developers.redhat.com/articles/2023/04/12/why-you-should-use-iouring-network-io>. [Cited on page 6.]
- KERNEL.DK, 2019. Efficient io with io_uring. https://kernel.dk/io_uring.pdf. [Cited on pages 5, 6, and 26.]
- LEPERS, B.; BALMAU, O.; GUPTA, K.; AND ZWAENEPOEL, W., 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19* (Huntsville, Ontario, Canada, 2019), 447–461. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3341301.3359628. <https://doi.org/10.1145/3341301.3359628>. [Cited on pages 1 and 3.]

Bibliography

- MANNALY, A. <https://github.com/yahoo/HaloDB>. [Cited on page 3.]
- MANNALY, A., 2018. Introducing halodb, a fast, embedded key-value storage engine written in java. <https://yahooddevelopers.tumblr.com/post/178250134648/introducing-halodb-a-fast-embedded-key-value>. [Cited on page 3.]
- OPENJDK. Java microbenchmark harness. <https://github.com/openjdk/jmh>. [Cited on page 10.]
- OPENJDK, 2023a. Jep 442: Foreign function & memory api (third preview). <https://openjdk.org/jeps/442>. [Cited on page 4.]
- OPENJDK, 2023b. Jep 444: Virtual threads. <https://openjdk.org/jeps/444>. [Cited on page 5.]
- OPENJDK, 2023c. Jep 454: Foreign function & memory api. <https://openjdk.org/jeps/454>. [Cited on pages 4 and 5.]
- ORACLE, a. Calling a c library function with the foreign function and memory api. <https://docs.oracle.com/en/java/javase/21/core/calling-c-library-function-foreign-function-and-memory-api.html>. [Cited on page 5.]
- ORACLE, b. Filechannel (java se 21 & jdk 21). <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java.nio/channels/FileChannel.html>. [Cited on pages 8 and 15.]
- ORACLE, c. Filechannel (java se 22 & jdk 22). <https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java.nio/channels/FileChannel.html>. [Cited on page 8.]
- ORACLE, d. Java downloads. <https://www.oracle.com/au/java/technologies/downloads/>. [Cited on page 3.]
- ORACLE, e. Memory segments and arenas. <https://docs.oracle.com/en/java/javase/21/core/memory-segments-and-arenas.html>. [Cited on page 4.]
- PANGIN, A. async-profiler. <https://github.com/async-profiler/async-profiler>. [Cited on page 10.]
- SAMSUNG. Mzplj3t2hbjr-00007. <https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1733-pm1735/mzplj3t2hbjr-00007/>. [Cited on page 7.]