Minimizing read bottlenecks in I/O bound systems

Effective and adaptive I/O using C++

Master's thesis in Computer Science and Engineering

Tobias Bäckemo

Konrad Olsson

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Minimizing read bottlenecks in I/O bound systems

Effective and adaptive I/O using C++

Tobias Bäckemo

Konrad Olsson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Minimizing read bottlenecks in I/O bound systems
Effective and adaptive I/O using C++
Tobias Bäckemo, Konrad Olsson

Supervisor Company: Patrik Ellren, Carmenta
Supervisor Department: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering
Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Minimizing read bottlenecks in I/O bound systems
Effective and adaptive I/O using C++
Tobias Bäckemo & Konrad Olsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The overall performance of a computer system that reads a lot of data is directly linked to its ability to extract performance from the storage hardware. This thesis, conducted together with the company Carmenta, examines how to optimize a system's data reads. The targeted storage device is an NVMe SSD. Several experiments simulating different workloads were conducted and laid the foundation for a new set of guidelines. Lastly, the guidelines were evaluated in a custom-built benchmark. Based on the evaluation, it was concluded that by using the guidelines, a system's performance could be increased.

Keywords: I/O, read, C++, SSD, NVMe, Windows, GIS, mmap.

Acknowledgements

Firstly we would like to thank all the people at Carmenta for creating such a fun and friendly work environment. Especially we would like to thank Patrik Ellrén for all his guidance and assistance during this thesis. We would also like to thank Magnus Åkerstedt Bergsten for providing us with technical assistance.

Furthermore, we would like to thank our supervisor from Chalmers, Ahmed Ali-Eldin Hassan, for his guidance during this thesis. We would also like to thank our examiner Carl-Johan Seger for valuable feedback.

We had to reach out to some experts during the thesis regarding memory-mapped files. So thank you, Dr. Alexandra (Sasha) Fedorova, University of British Columbia, and Dr. Andrew Crotty, Carnegie Mellon University, for answering our emails.

Last but not least, a great thank you to all of our friends at Chalmers for making our time here so amazing. If you come across this thesis, know that you are not forgotten.

Tobias Bäckemo, Konrad Olsson, Gothenburg, June 2022

Contents

1 Introduction

Many modern computer systems read and process large amounts of data while running. Although storage devices have become faster, the ability to scale processing surpasses the ability to scale Input/Output (I/O). Because of this reason, it is of utmost importance that storage I/O is optimized [15].

At the application level, a developer has multiple options when deciding how to read data from secondary storage [37]. Which option is most suitable depends on several factors. Two factors are what kind of secondary memory the host system is using and the system's workload.

With the introduction of non-volatile memory express solid state drives (NVMe SSDs), latency has drastically decreased. Compared to a hard disk drive (HDD), the NVMe SSD is orders of magnitude faster [34]. Due to the significant performance improvement, latency from CPU tasks is no longer negligible. For example, unnecessary memory copies can now noticeably affect a system's performance [9]. Moreover, the internal architecture of SSDs differs from an HDD's architecture. The best method for reading from one type of storage device is not necessarily the best method for reading from another [13].

Another factor that impacts which option is most suitable for reading data are a system's workload [32]. A system can access data in many different ways. Two examples are data being accessed in small chunks at several offsets in a file and accessing data by large sequential reads.

If the workload and the targeted storage device are not considered, reading data can become a bottleneck of a system's overall performance. Based on this consideration, this thesis introduces a set of guidelines that a developer can use to optimize reading in their system.

1.1 Background

An example of a computer system that reads a lot of data is a Geographic Information System (GIS). A GIS's purpose is to display and analyze geospatial data. Displaying the data is done by combining multiple layers of information. These layers can be, for example, elevation information, roads, rivers, buildings, and geographic names [11].

Carmenta, the company at which this thesis was produced, provides the tools to build Geographic Information Systems. The main tool Carmenta provides for this is called *Carmenta Engine*, which is a software development kit for creating geospatial applications. These geospatial applications are often run on personal computers where all the data needed resides locally [4]. An improvement in reading performance will benefit the performance of the geospatial application as a whole, providing a better user experience with a shorter startup time and smoother interaction with the application.

1.1.1 Previous work

Regarding previous work, three types of papers were of extra interest. Firstly, papers containing discussions on how the advantages of using an SSD can be utilized. There exist several papers of this type [40, 38, 36]. The papers highlight how suboptimal decisions are made due to a lack of awareness of the internal architecture of the SSD and provide possible solutions.

However, what is in common in the previously mentioned papers and differs from the scope of this thesis, is that all of the presented solutions are based on modifying or building a new file system. Furthermore, in contrast to the previous papers, this thesis only presents guidelines for optimization that can be implemented at the application level. Since one aim for this thesis was to utilize NVMe SSDs, understanding how they operate is crucial. Luckily, such papers exist in abundance, providing information about access latency, physical design, possibilities for parallelism, internal page sizes, and more [33, 39].

Secondly, another interesting type of paper for this thesis was those that discussed how one could optimize against different workloads. Saif, Nussbaum, and Song (2020) [33] have written a paper in which they examine how I/O access patterns impact the performance of an SSD. Inspiration was taken from experiments and analyses included in their report. However, what differs between their project and this project is that they did not use different reading techniques. For example, they are not comparing the buffered reading with memory-mapped reading. Moreover, all of their experiments were conducted on a Linux machine.

Lastly, papers that evaluate and compare different read methods were of interest. For this thesis, experiments that compared memory-mapped file reading and traditional buffered reading were of extra interest. Andrew Crotty, Viktor Leis, and Andrew Pavlo (2022) [7] are very critical to using memory-mapped file reading in the context of Database Management Systems. Their paper points out some of the flaws of using memory-mapped file reading. Even though this thesis is not focused on database management systems, the insights they share were of great value for analyzing results. Moreover, what is different from their work compared to this thesis is that their experiments were conducted on a Linux machine.

However, memory-mapped file reading has also been pointed out as an excellent alternative to traditional read calls. Alexandra (Sasha) Fedorova, professor at the University of British Columbia, gives an overview of how memory-mapped files work and some of the gained advantages [10]. Although all of her experiments are conducted on Linux, she shares knowledge that was used to analyze results when comparing memory-mapped files in Windows with other I/O techniques.

This thesis is unique because it researches how a developer can optimize reading at the application layer by choosing the preferable I/O technique given a system's workload.

1.2 Purpose

The purpose of this project was to examine how a read-heavy system can be optimized on an application level. The increase in performance has been gained by utilizing the NVMe SSD's advantages and optimizing against different workloads. The main idea was to select the best suitable read strategy given a workload and provided that the host system is using an NVMe SSD. By being smart regarding the choice of reading technique, one can extract more performance from storage hardware and hence remove bottlenecks.

1.3 Formulation of the problem

As mentioned above, the goal of this project was to examine how a read-heavy system can be optimized by considering its workload and utilizing the advantages that come with NVMe SSDs. To achieve the goal of this thesis, the following three questions had to be answered.

- How can reading be optimized at the application level by utilizing knowledge about a system's workload?
- How can the advantages of using NVMe SSD be utilized in the context of reading data?
- How should the developed guidelines be evaluated?

1.4 Limitations

The following four limitations were set to keep the project's scope reasonable.

1. It was decided to formulate and evaluate the new guidelines on a computer that runs Windows 10 64-bit version. Specifications can be seen in appendix A. Developing guidelines for other operating systems such as macOS or Linux would also be interesting. However, due to lack of time, it was decided to limit the project's scope by only focusing on Windows 10.

2. Data can be stored in many different ways, in databases, on the network, on USB drives, etc. However, this thesis only considers data stored locally in secondary memory.
3. All code is written in C++. Due to this choice, implementations in other languages have not been evaluated.
4. The different I/O techniques were tested in experiments with synthetic data. However, when it comes to benchmarking, it was decided only to evaluate the guidelines for file formats that handle geospatial data.

1.5 Disposition

The remainder of this thesis is organized as follows. The following chapter is a theory chapter, giving all the background knowledge necessary for understanding the rest of this thesis. Subsequently, a chapter about the research methodology presents the crucial parameters for testing. Moreover, the chapter about research methodology also includes a description of the implemented testing functions. After that comes the chapter in which the experiments on synthetic data are presented and discussed. Ensuing the experiments comes a chapter where the gained knowledge is used to improve the performance of a real-life system. Finally, conclusions and suggestions for future work are presented.

2
Theory

This chapter presents the theory about hardware and software relevant to understanding this thesis, starting with the targeted storage device for this project, the NVMe SSD.

2.1 NVMe SSD

A solid-state drive (SSD) is a flash memory-based storage device. An SSD is made out of an array of flash chips. Each chip is made up of two or more planes. The planes contain multiple blocks, where each block, in turn, is made up of 64 or 128 pages. A page is the smallest unit of data that can be read or written. The data area of the page contains the stored data and can be 2KB, 4KB, 8KB, or 16KB in size. In Figure 2.1, one can see the architecture of an SSD. The list that follows briefly describes the components that an SSD consists of and their tasks [18, 20, 13].

- **Host interface:** The SSD and host computer communicate over the host interface.
- **SSD Controller:** The main task of the SSD controller is to translate the I/O operations into flash memory operations.
- **RAM:** The SSD disk has an internal RAM in which it can cache relevant data such as read commands or recently accessed data. The internal RAM is also used to store the map table, which contains information about where data is stored on the device.
- **Processor:** The processor gives commands and tasks to the flash controller.
- **Flash Controller:** Manages flash components. There can be multiple flash controllers in a single SSD.
- **DMA Controller:** Responsible for transferring data when receiving commands from the processor.
- **Channel:** Flash memory packages are connected to different channels. Multiple channels can be accessed in parallel.
- **Flash Memory Packages:** Stores data.

In contrast to a hard disk drive (HDD), an SSD contains no moving parts. Since the memory in an SSD is not accessed by a physical arm, memory access latency does not depend on, in theory, if memory is sequentially or randomly accessed. SSDs are also highly parallel because of their many flash chips, meaning that one SSD can service multiple I/O requests simultaneously. Another vital distinction between

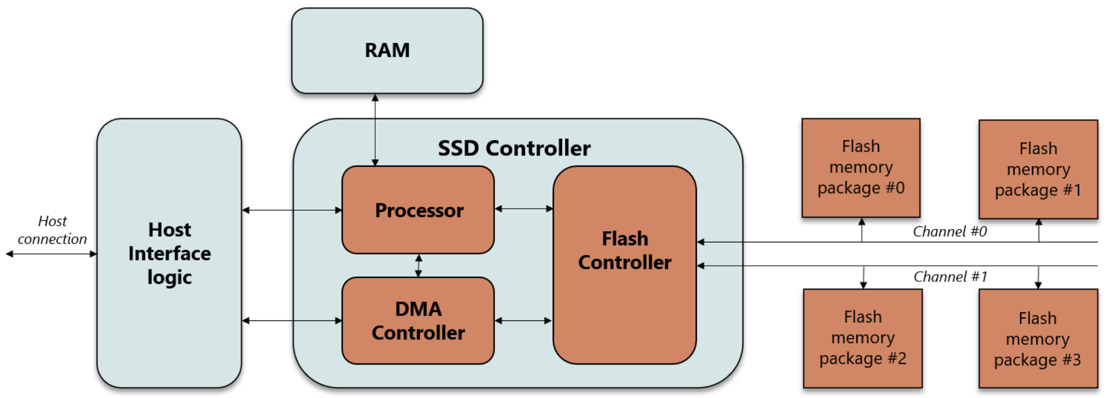

Figure 2.1: The internal architecture of a solid state drive

HDDs and SSDs is that while HDDs use the same block size as the page size that the OS typically uses, namely 4KB, an SSD uses what is called a clustered page. A clustered page is an internal unit of reading or write operations between 4KB and 4MB [33, 18]. The idea is that when the SSD receives a write request for some number of pages, it will write them to different chips. When the same data is to be accessed, the SSD can fetch it in parallel, increasing throughput. See Figure 2.2 for a visual representation.

When a host wants to send an I/O request to the SSD, it does it via the host interface. The type of interface has a significant impact on the overall I/O performance [39]. In personal computers, there are two common ways to connect SSDs. Through Serial Advanced Technology Attachment (SATA) or Non-volatile memory express (NVMe). The NVMe interface is used to access flash-based storage over the peripheral component interconnect express (PCIe) bus. NVMe is a newer interface that has become common in personal computers during the last couple of years. There are two main benefits of the NVMe interface compared to the SATA interface.

The first benefit is the increased throughput enabled partly by the bandwidth of the PCIe but also by features of the NVMe controller. The NVMe SSD connects to 4 PCIe lanes, which gives a theoretical bandwidth of about 3.94 GB/s and 7.88 GB/s on PCIe 3.0 and PCIe 4.0, respectively, while SATA 3.0 speed is about 550 MB/s. When it comes to the interface itself, SATA supports a single I/O queue with a depth of 32, while NVMe offers as many queues as CPU cores, with a maximum of 64K and a depth of 64K per queue [35].

The second advantage is access latency. The I/O path in both hardware and software is shorter for NVMe devices compared to SATA [39].

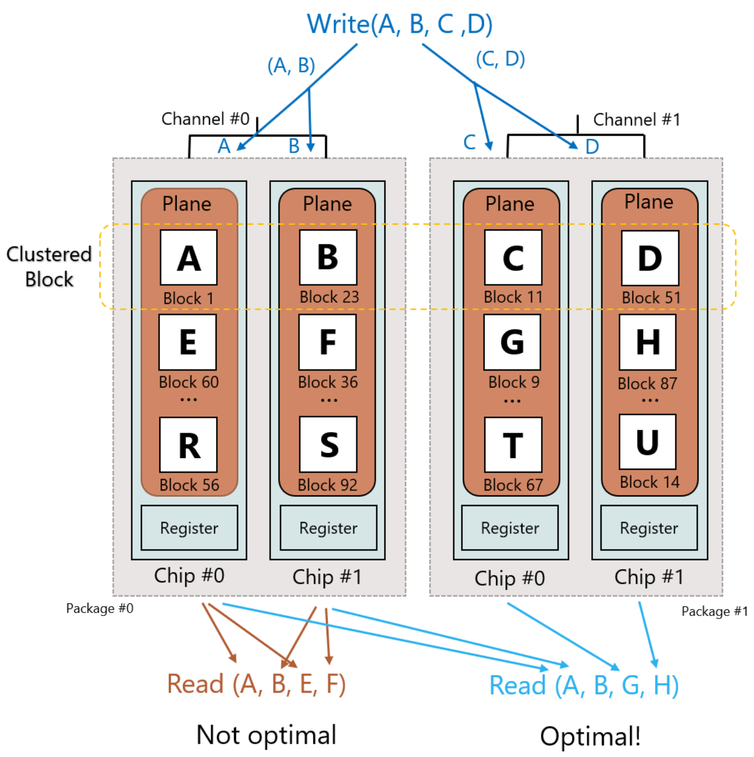

Figure 2.2: Parallelism of an SSD

2.2 The iostream

The *iostream* is a C++ class included in the C++ standard library [6]. The class provides functions that can be used to read from and write to standard output but also other tasks such as file I/O. The functions are built on top of system calls as an abstraction to make I/O easier to use and faster to implement for programmers. A downside with *iostream* is that the implementation includes unnecessary memory copies, which affects the overall performance if the storage device is fast [9].

2.3 Memory-mapped file

Memory-mapping is a technique that can, among other things, be used for reading and writing to files. The technique is available in many operating systems. Two examples are Linux and Windows [37]. The main idea is to use the operating system's paging to perform reading and writing.

When a calling process asks the kernel to memory-map a file, the kernel will allocate a part, equal to the size of the file, of that process's virtual address space and return a pointer pointing at the beginning of this allocation to the calling process. At this point, no data is loaded. When the process uses the given pointer to access a page of mapped file data, i.e., read the file, a page fault will be triggered, and the operating system will swap in the demanded data [7].

It has been hinted that there is an internal parameter in Windows called memory-mapped file chunk size. The internal parameter defines how many pages are paged in upon a page fault. This chunk size can be different for different versions of Windows [5].

Three advantages of using memory-mapped files in Windows for accessing files are listed below [14, 17].

- Because there is no explicit system read calls to be issued, a system can gain performance by avoiding unnecessary memory copies and context switches.
- Using memory-mapped files is very convenient. Once the memory-mapped file is open, reading from a file becomes a simple dereferencing of the pointer.
- In contrast with buffered reading, there is no need to care for buffers when using memory-mapped files. Instead, the operating system will do this task.

One disadvantage with using memory-mapped files is that when it is used on a 32-bit machine, the virtual address space is limited, and hence it is not possible to map files that are larger than 2-3 GB [14]. Another disadvantage is that asynchronous I/O is not possible with memory-mapped files. If a page fault occurs, the operating system will block the calling process until the data is fetched from secondary memory [30].

2.4 I/O metrics

How well a specific reading strategy performs is described by an I/O metric. Two commonly used I/O metrics are latency and throughput [13]. Latency is the time that it takes to perform a specific I/O task. For example, how long it takes to read a 1GB file. Throughput is defined as the number of bytes transferred divided by the latency, B/s. This metric can be interpreted as the speed at which the system can read or write data. Using different I/O metrics will describe the distinct characteristics of a reading strategy [13].

2.5 File systems

The file system defines how files are named, stored, and read from the storage device. In this report, file systems refer to the storage partition's format. Some examples of such formats are the File Allocation Table (FAT) [27], New Technology File System (NTFS) [25] and Apple File System (APFS) [2].

2.5.1 Files system block size

Most of a system's files are usually stored in the secondary memory [37]. A way to manage the memory space could be to allocate memory in the disk for the given file size. However, this strategy has a major defect. If the file grows, for example, a user might append new text to a text file, then the file may have to be copied to another place in storage since the space is no longer enough. Copying large amounts of data from one place in the disk to another takes time and is not something you want to do unnecessarily.

Another strategy, that copes with this drawback, is to instead divide the file into some amount of blocks with a given block size. The blocks do not have to be bordering. A file system's selected fixed block size impacts read performance and space efficiency. A very large block size will result in unused memory since small files will only take up a small portion of the total block size. On the other hand, a small block size will instead make it such that many seeks will be needed to gather all of the data since it is spread out over several blocks. Microsofts file system NTFS default block size depends on the volume of the disk. Disks with a volume size lesser than 16 terabytes have a block size of 4096 bytes [23].

2.5.2 Optimization techniques used by file-systems

To best utilize the available hardware and hence get better overall performance, several file systems have been implemented with different optimization techniques [37]. Bellow follows two commonly used optimization techniques.

2.5.2.1 Buffer cache

The buffer cache is a portion of data that resides in secondary storage but is temporarily also located in the main memory, even though the data is not currently used [37]. The idea is to minimize the number of reads from secondary storage. The file system will first look in the buffer cache when a read operation is executed. If the required data is located in the memory, the data can be accessed a lot faster in comparison to if the data would have to be accessed from secondary storage [37].

2.5.2.2 Prefetching

Another optimization technique is called prefetching. Prefetching, like buffer-cache, relies upon the idea of storing data in the memory since accessing it is faster than accessing the disk. However, what differentiates prefetching from buffer-caching, is that the data is loaded into memory before being demanded. For sequential reads, prefetching can increase a system's performance. For example, if a block is read, there is a high probability that a neighboring block will soon be demanded. However, for random reading, prefetching can be more complex, and if the prefetched data is never used, it can harm the system's performance [37].

2.6 Reading block size

When reading data from the secondary memory using buffered reading, it is done in chunks of a predetermined size. The block size that is used for buffered read has an impact on both latency, and throughput [34]. Reading in larger chunks will naturally take up more main memory, but it can also lead to greater throughput. Previous research shows that for the NVMe SSD, a larger block size is always preferable [34]. However, it also shows that one cannot fully utilize its capabilities even with a block size of 64MB. To do that, one must use multiple threads or asynchronous reading, which are discussed in the following sections.

2.7 Parallel I/O

Parallel I/O, in this thesis, refers to when multiple threads are used to make I/O-requests. Not to be confused with the parallel architecture and mechanisms used by an SSD mentioned before. The idea is that using multiple threads to make read requests will increase performance. Furthermore, simultaneously sending multiple I/O-requests allows the kernel to reorder I/O-requests, improving performance even on an HDD. Prior studies also show that NVMe SSDs are better at utilizing parallel I/O than HDDs [34].

2.8 Synchronous and Asynchronous read

In Windows, it is possible to do either synchronous or asynchronous reads [30]. When using buffered read synchronously, the kernel will block the rest of the system until all demanded bytes are read. However, in some cases, this way of doing I/O can harm a system's performance since it could, for example, block available resources from performing other computations.

An alternative way of doing I/O is asynchronously. The kernel will not block the system when reading asynchronously, even though the data has not been read into memory. Using asynchronous read has some benefits compared to synchronous read. Firstly, the calling process can continue executing other instructions before checking if the result has been returned instead of waiting for the reading to complete. Secondly, creating numerous I/O requests enables the kernel drivers to reorder, split up, or overlap multiple I/O requests for increased throughput [29].

Similar to the parallel I/O approach, asynchronously I/O will allow several read requests to be handled simultaneously. A difference between asynchronously and parallel reading is that in the case of asynchronous reading, a single thread can start up many requests but will have to, in some later part of the program, gather the results by listening to multiple events. The amount of simultaneous events is called the queue depth [34].

3
Research methodology

The research was conducted in four phases. The phases were the prestudy, the development, the experiment, and the evaluation phase.

3.1 Prestudy phase

In this thesis, it was decided to define the workload as a set of parameters. A decision had to be made on which parameters should be included in the the workload. Moreover, which tools available at the application layer for reading data had to be investigated.

3.1.1 Parameters

Several parameters impact the reading performance. However, a decision had to be made regarding which parameters should be examined. The decision of which parameters should be examined was based on two sources of information. The first source was available literature such as books and scientific papers regarding reading performance. Examining the structure of the SSD and how reading is handled in Windows gives hints regarding which parameters might impact reading performance. Secondly, discussions were held with supervisors at Carmenta, in which they described their system's workload. The following parameters were examined.

File size: The file size is the measure of how much data a file holds [37]. A system can read files of many different sizes. The spectrum is vast, and files can differ in size by factors of hundreds of millions.

Read request size: The read request size is the amount of data read in a single read operation. The size typically differs from a few bytes up to megabytes large reads [19].

Total bytes read: Total bytes read in this context means the amount of data read in one file. This size is not limited to the file size since a system can read a part of the file more than once.

State of cache: A cache hit is when a read operation is executed, but the system can find the data in memory and does not have to access the secondary memory. Due to the significant difference in latency between disk and memory access, read operations performed in a cache hit situation will have a greater throughput. However, when conducting a benchmark for a particular reading technique, it is essential to consider the state of the cache. Otherwise, the results will be misleading. Therefore, it has been decided to run all experiments on a cold cache. In other words, all cache is cleared before the execution of the experiment starts [9].

Available threads: Generally, PCs today have a multi-core processor and, by extension, multiple available threads for performing actions. Some applications require heavy computations, leaving little CPU resources available for I/O, while other programs do only little computation and therefore have more resources to put on I/O. If one wants to read a lot of data, it might be worth using more CPU resources, i.e., multiple threads on the reading of data to increase throughput.

Access pattern: When reading data, the access pattern impacts reading performance [34]. Therefore, it was decided to categorize a access pattern as one of two general types. The first type of workload that was considered was *sequential read*. As the name implies, sequential read means that the data is accessed in a sequenced order. An example of a use case where this workload shows up is when reading an image as a whole. The second type of access pattern that was considered was *random read*. In contrast to sequential read, random read is when data is read from several different locations in a file, not necessary in order. An example of where this workload shows up is if the system wants to draw a map and only needs some parts of the geographic data available in a file [32].

3.1.2 Reading techniques

There are several alternatives for how one can read data from a disk in application space. The goal of the developed guidelines is to select the best suitable reading technique given the parameters previously described. By doing literature studies and discussions with the supervisors at Carmenta, the decision was made to examine four different techniques. In the next section, the selected reading techniques will be described.

3.2 Development phase

The developed guidelines presented in the next chapter were based on results from several experiments. However, in order to conduct the experiments, a reliable test environment had to be developed. The designed test environment consists of test functions that test how a specific reading technique performs for a particular workload. The components that build up the test environment and design decisions made to make the tests reliable are described below.

3.2.1 Reader overview

Each synchronous reading technique is implemented as a class with a common interface called *IReader*. Using a common interface has several advantages when comparing reading techniques. One advantage of this design decision is that time is saved by avoiding redundant code. Common tests could be used instead of developing a test function for each reading technique. Another advantage is that the readers' evaluation is more reliable when common test functions are used. By using the same test code, all readers have the same preconditions.

The interface is inspired by how reading is done in a typical read-heavy system, namely Carmenta Engine. The idea was that by following the interface, the readers would be easy to integrate into a real-life system. The interface forces the class to have two instance variables and implement a set of required functions. Below is a quick overview of the interface.

3.2.2 Instance variables

The *IReader* interface has two instance variables. The first is the *path_* variable, which is a string containing the path to the file to be opened. The second variable is the *maxExtent_* variable, which stores the length of the file.

3.2.3 Functions

`size_t readBytes(size_t offset, size_t amount, char* buf)`
This method is used to read from the currently opened file. The function returns how many bytes were read.
readBytes takes three arguments.

- **offset:** Where the reading should begin.
- **amount:** The amount of bytes to be read.
- **buffer:** A pointer to a buffer. After the read operation is executed, the demanded data will be located in the buffer.

`bool isOpen()`
Returns whether a file is open or not.

`bool openNewFile(std::string path)`
This method opens a new file handle for the provided file path *path*. If the file handle is already opened, it will not be reopened since this would clear the cache related to the file. Returns whether the file could be opened or not.

`bool close()`
This function closes the currently open file handle. Returns whether it successfully closed the file or not.

`std::string getName()`
This utility function returns the name of the reader class as a string. It is mainly used when one wants to print information to the console.

3.2.4 Developed Readers

Based on the insights gained from the earlier mentioned literature studies regarding reading techniques, it was decided to implement the following four readers.

3.2.4.1 StreamReader

The *StreamReader* uses functions provided by *iostream* to execute reading operations. This reader was created to use as a comparison against low-level methods.

3.2.4.2 MappedReader

The *MappedReader* uses the available Windows system calls *CreateFileW(), CreateFileMappingW()* and *MapViewOfFile()* to create a memory mapping to the desired file [12, 22, 24]. As explained in the theory section, once the file is mapped, reading some data from the file becomes a simple dereference of a pointer. Reads are done by simply doing a memory copy from the mapped region to the provided buffer. This is not how one would use a memory-mapped file in production, but the memory copying forces the pages to be fetched by the OS.

3.2.4.3 BufferReader

The *BufferReader* uses the available Windows system calls *CreateFileW()* and *ReadFile()* to read files [12, 28]. To use the *BufferReader* one important internal component has to be set, that is the block size. The block size is the largest amount of data read at once. If the read request size is less than the block size, *BufferReader* will use the Windows system call to read all of the requested data at once. However, there is a max limit on how big the buffer size can be, 4GB per reading operation. The limitation comes from the fact that the parameter in the Windows system call that determines the amount to read is only 32-bit long [28].

3.2.4.4 AsyncReader

The AsyncReader uses the same Windows system calls as *BufferReader*, but with different arguments, for making asynchronous read requests. *AsyncReader* has a limit of 64 simultaneous requests, so if the reader needs to make more, it will first wait for the current ones to finish before creating new requests. AsyncReader does not implement the *IReader* interface since the interface is not compatible with how the asynchronous calls work. Therefore the asynchronous reader is both implemented and tested a bit different from the synchronous ones.

The main difference between the *AsyncReader* and the synchronous implementations is the function responsible for reading data. In synchronous readers, the function returns when the read has been performed, but making the asynchronous reader behave the same way would ruin the whole purpose of reading data asynchronously. Instead when *AsyncReader* reads data it returns an *OVERLAPPED* structure.

An *OVERLAPPED* structure contains the offset from which to start reading the data and an event that gets triggered once the read has been performed [26]. Therefore it is the one calling the read method which is responsible for checking that the read has been successfully performed.

3.2.5 Testing methodology

To get reliable data, each data point was created by taking the average of multiple tests. The number of trials was ten. For example, to see what throughput was achieved by reading an entire 1GB file in 4KB blocks, the file was read ten times with the 4KB block size, and the average latency was then used to calculate the throughput. The reason for running so many tests for a single data point is to minimize the impact of outliers on the final result.

3.2.6 Considering the cache

Issues can arise if one is not careful about handling the cache when conducting reading experiments. One would expect similar results for an experiment that runs several times. However, if one does not consider the state of the cache, performance can increase significantly after the first run of the experiment. The speedup comes from data being cached at the first run of the experiment. When the experiment is rerun, the requested data can be found in the memory, and hence access time drops. Therefore, it was decided to run all experiments on a *cold cache*. A *cold cache* is a state of the system in which no file data is already present in memory at the read time [9].

Two actions have been taken to ensure the cache is clear before conducting any experiments. Firstly, a Windows tool called *RAMMap* is used to clear the buffer cache [31]. Secondly, the file handle is closed and re-opened between tests since it was observed during testing that some cached data is connected to the file handle. By closing the file handle, this data is cleared.

3.2.7 Simulating random reads

There are a couple of ways one can go about simulating random reads. The first intuitive option is to generate some random offsets within the file and then read a certain amount of bytes at them. This approach is versatile since the number of random reads and the amount to read at each offset can change. However, this approach was not chosen for two reasons. Firstly, the mentioned approach does not simulate a real-world scenario well since it reads from completely random offsets. In reality, a file will contain blocks of data, and it does not make sense to make a read at an offset that is not located at the beginning of a block. Secondly, it can be hard to predict cache hits when the block size and number of reads can are randomly chosen. For example, if one wants to make an experiment determining throughput on a cold cache using random reads, doing many reads with a large block size will result in many cache hits on the last reads.

Instead, a block size is chosen, representing the chunk of data to be read at each random read. The file is then split into some offsets based on the block size. Due to this design decision, the number of random reads in a file will depend on the block size and the file size. Shuffling these offsets and doing block-sized reads from them created a random read scenario more grounded in reality. This method also ensures that the same data is not read multiple times, avoiding cache hits when suitable.

3.2.8 Testing parallel and asynchronous reading

From previous work, it has been shown that more performance can be extracted from an NVMe SSDs when parallel or asynchronous reading is used [34]. For this thesis, a set of test functions was developed to evaluate asynchronous and parallel reading's impact on performance. In the parallel experiments, the work was split between threads, and in the asynchronous experiments, the work was split between asynchronous calls.

When executing a sequential read, the work is divided into block-sized calls. When reading many random offsets, the work is divided by the individual random reads. Of course, one can combine the parallel and asynchronous approaches to optimize performance, although this possibility is not explored in this thesis due to time constraints.

To enable parallel execution, the work was divided into blocks. The tests used the OpenMP API, an API for parallel programming, to separate the responsibility of reading the blocks between threads [3]. The chosen block size decides the total number of blocks to read. For example, if the file size is 24KB and the block size is 4KB, there will be six blocks. The next design decision is how to schedule the reads between the threads. For simplicity, OpenMP's default scheduling is used, which is quite simple: The reading operations are split evenly and in order between the threads. So, for example, if there are six reads to be done and two threads, the first thread will execute the first three reads, and the second thread will execute the last three reads.

3.2.9 Synchronous single-threaded tests

The performance of a reader is measured for different workloads with the help of test functions. The test functions take a pointer to an *IReader* as input and use the interface's visible functions to perform the reading tasks. Below is a description of the implemented synchronous single-threaded test functions.

```
size_t readEntireFile(IReader* reader,std::string path)
```

This function uses a reader to read an entire file into a buffer. First, it starts a timer. Then it opens the file and allocates a buffer of appropriate size. After that, the read is performed. Lastly, it stops the timer, deallocates the buffer, and returns the measured time.

```
size_t readSeq(
    IReader* reader,
    std::string path,
    size_t amount)
```

The *readSeq* function works exactly like *readEntireFile* but with a configurable number of bytes to read.

```
size_t readManyShuffled(
    IReader* reader,
    int nrRandomReads,
    std::string path,
    size_t amount)
```

In the *readManyShuffled* function the file is divided into blocks of size *amount*. Then the blocks are shuffled with a standard library function. Then, the timer is started, and all the blocks of data are read by the provided reader. Lastly, the timer is stopped, and the duration is returned.

```
size_t readOneBlock(
    IReader* reader,
    std::string path,
    size_t blockSize)
```

The function uses a reader to read a single block of *blockSize* at offset zero. Returns the time it took to read the block.

3.2.10 Asynchronous tests

As mentioned earlier, the asynchronous reader is different since the reader returns an *OVERLAPPED* structure instead of a result when issuing a request for reading data. Therefore, waiting for the request to return a result must be done within the testing function. There are two asynchronous test functions: *readManySeq* and *readManyRandom*.

```
size_t readManySeq(
    AsyncReader* reader,
    std::string path,
    size_t amount)
```

readManySeq works as follows. The file is divided into blocks, and the read operations are then divided between asynchronous calls. Since the number of blocks to read might be more than the number of possible simultaneous calls, many reads are started and then waited to finish before starting another batch. This process continues until all blocks have been read. The timer is started at the beginning of this reading process and stopped when it is done. Lastly, the duration is returned.

```
size_t readManyRandom(
    AsyncReader* reader,
    std::string path,
    size_t amount)
```

The function *readManyRandom* works in the same way as *readManySeq*, but the offsets are shuffled so that the reads are done in random order.

3.2.11 Parallel tests

The parallel tests are very similar to single-threaded ones in most regards, but with some differences. There are two functions used to make parallel read test: *readSequentialParallel* and *readManyShuffledParallel*.

```
size_t readSequentialParallel(
    char reader_type,
    std::string path,
    size_t amount)
```

The function *readSequentialParallel* is used to read a file sequentially and works much like *readManySeq* but divides the work between threads instead of asynchronous calls. It also comes with an extra consideration. Even though each thread will read from exactly the same file, they cannot use the same file handle unless it is a memory-mapped file since that would lead to race conditions. Since reading a memory-mapped file is just accessing an address in memory, there is no race condition if multiple threads want to access the same mapped file. Therefore, if the reader to be used is the memory mapping one, just one file handle is opened, but if it is any other reader, one handler is opened for each thread. The timer is started before the threads are created, so that thread creation overhead is measured. The threads read their designated part of the file, then the threads are terminated, the timer stops, and the duration is returned.

```
size_t readManyShuffledParallel(
    char reader_type,
    std::string path,
    size_t amount)
```

readManyShuffledParallel works precisely as the sequential version but with the shuffling included.

3.3 Experiment phase

After the test environment was developed, several experiments were conducted for different workloads. All of the experiments were run on synthetic data. Synthetic data is data that is created artificially rather than data taken from any real-world system [8]. Based on the results from these experiments, guidelines were developed.

The guidelines are based on utilizing knowledge about a system's workload and are tailored for a system that runs an NVMe SSD as the storage device.

3.4 Evaluation phase

As previously mentioned, the span of parameters that impact the final performance is enormous. Hence, it is essential to do in-house benchmarking for the actual workload that the targeted system will have to handle. To strengthen belief in the recommended guidelines, the guidelines were tested on a benchmark that mimics Carmenta engine's workload. The hypothesis was that by using the developed guidelines, one would be able to see a performance gain compared to running the same benchmark on a system that uses a naive way of reading data. The data received from Carmenta was created by recording all executed read calls in some selected use-cases. These reads were recorded into Comma Separated Value (CSV) files. Since all file reads made by the Carmenta I/O system are done using 4KB blocks, all entries in the CSV files were 4KB reads. One could determine which reads were larger sequential ones by comparing subsequent entries. Therefore, all CSV files belonging to a load were parsed and combined into a set of reading instructions. The benchmark iterates through the list of operations and has the examined reader execute each read operation. The performance is measured as the time it takes to run through all operations.

4

Experiments

In the following chapter, results gained from several experiments are revealed. Firstly, the results gained from experiments regarding the internal components of the readers are presented and discussed. Secondly, results from comparisons between readers are presented. Thirdly, it is examined how throughput can be increased by using multiple threads. Lastly, the guidelines are presented.

4.1 Block size impact on performance

To use buffered reading, one has to set a block size. As will be shown, the choice of block size significantly impacts reading performance. In the following section, it is examined how the choice of block size impacts the throughput for both sequential reads as well as for random reads. Moreover, the latency for different block sizes is also examined. *BufferReader* has been used to test the different block sizes. All the data-points displayed in the following three graphs have been computed by taking the average of multiple tests as described in 3.2.5. Notice that the Y-axes have a logarithmic scale.

4.1.1 Sequential reads using different block sizes

In the first experiment regarding the choice of block size, an entire 1GB file is read sequentially using the *readEntireFile* function. The results of the experiment are shown in Figure 4.1.

Notice that using a larger block size gives a higher throughput when reading sequentially except for when using a block size of 256KB. Choosing a block size of 4MB instead of the common preference of 4KB will, for this specific use case, improve the throughput by a factor of approximately two. Moreover, one can see that the increase in throughput starts to stabilize when using block sizes greater than 4MB.

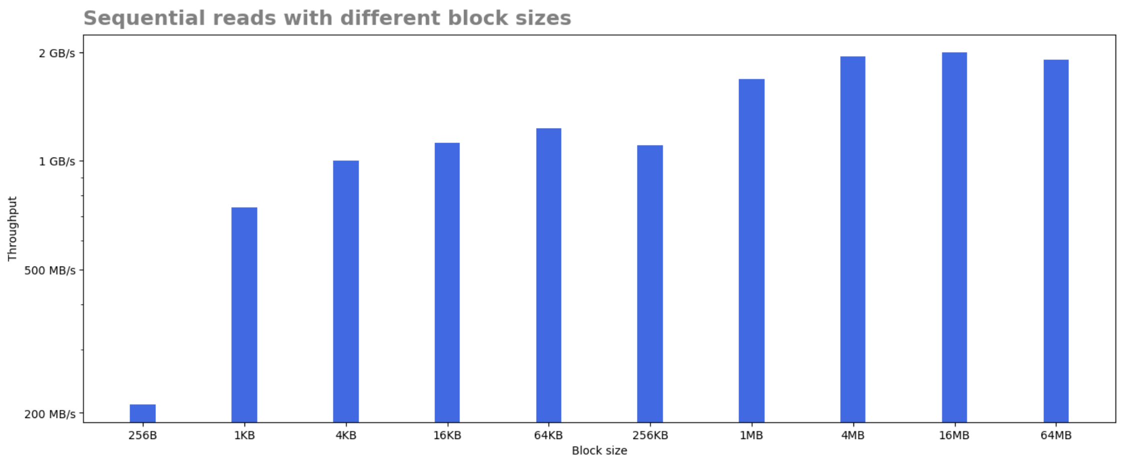

Figure 4.1: Block size impact on sequential read performance. The file size is 1GB. The entire file is read. Y-axis is logarithmic.

4.1.2 Random read using different block sizes

In the second experiment, it was examined how the throughput is affected by block size when doing random reads. This was achieved using the *readManyShuffled* function. The results from the experiment are shown in Figure 4.2.

As the result shows, using a larger block size for random reads will, as for sequential reads, result in higher throughput up to a specific threshold. For random reads, one can see that the gain in throughput stabilizes after choosing a block size that is greater than 4MB.

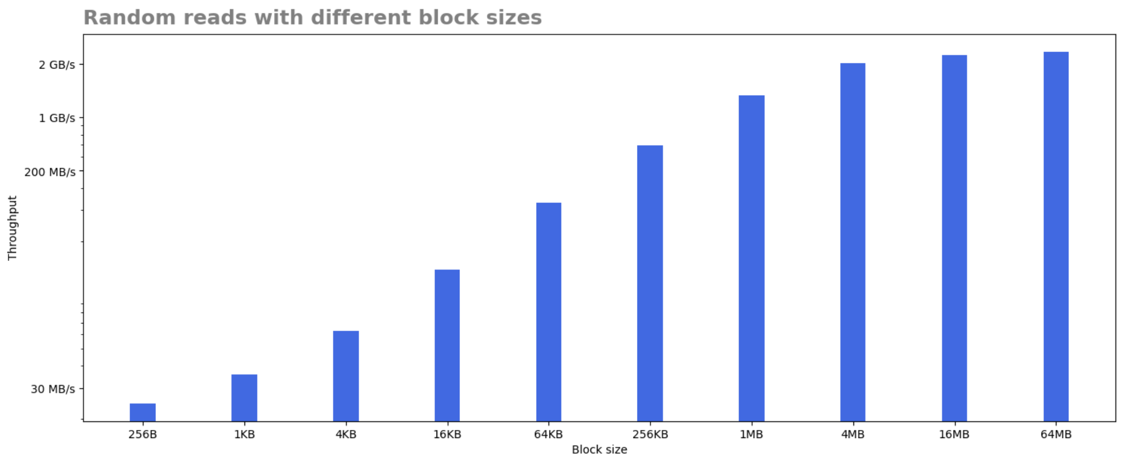

Figure 4.2: Block size impact on random read performance. The file size is 1GB. The entire file is read. Y-axis is logarithmic.

4.1.3 Latency using different block sizes

In the last experiment regarding the choice of block size, the single block latency was examined. The experiment was carried out using the *readOneBlock* function. It was measured how long it takes to read blocks of different sizes with buffered read. The results gained from the experiment can be seen in Figure 4.3.

From the results, one can notice that using larger block sizes results in higher latency for almost all block sizes. However, reading a single block of size 4KB takes less time than reading a block of size 1KB or 256B.

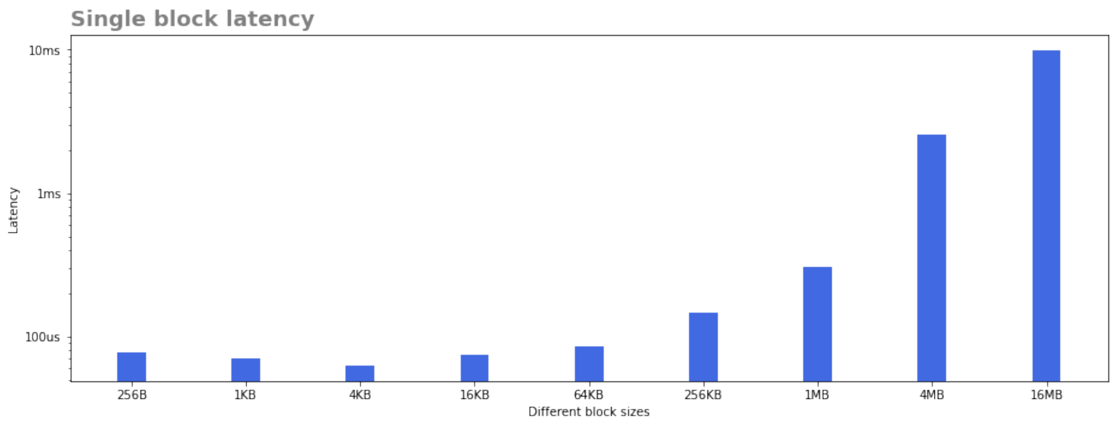

Figure 4.3: Block size impact on single block latency. The file size is 1GB. Y-axis is logarithmic.

4.1.4 Choosing the optimal block size

Although the workloads were different, the same pattern emerged. Using a larger block size results in a higher throughput up to some threshold. In these experiments, this threshold was approximately 4MB. But bear in mind that this threshold can be different if the experiments are run on a computer system that has different software and hardware.

However, the throughput does not tell the whole story. One must also consider latency. As the block size increases, so does the latency. Overhead will be introduced if unnecessary large block size is used for a small read request. For example, if the system needs to read a total of 256KB in a file, using a block size of 4MB would result in latency 30 times larger than what one would have gotten if a block size of 256KB were used.

When using buffered read for reading data, the following guidelines are suggested based on the results gained from three previously mentioned experiments.

- If request size $\leq$ 4KB then use block size of 4KB.

- If request size $\geq$ 4MB then use block size of 4MB.
- Else use a block size equal to the request size.

Later in this chapter, different readers are compared. We will then use two readers to represent buffered reading. The first one is *BufferReader4KB* which will use a block size of 4KB. Since Carmenta uses a block size of 4KB when performing reads, *BufferReader4KB* can be seen as a representation of how reading is done in their system today. The second one is *BufferReader4MB* which will use a block size in line with the guidelines above.

4.2 Queue depth impact on performance

When doing asynchronous calls, one can issue multiple read requests simultaneously. As mentioned in the theory chapter, the queue depth is the number of simultaneous calls. This section examines how the choice of queue depth impacts throughput. Experiments were conducted for both sequential and random reads.

4.2.1 Sequential reads using different queue depths

First up are sequential asynchronous reads. The file size is fixed to 4GB and the block size to 4MB. 4MB was chosen since that was one of the best performing block sizes for a buffered read. The experiment was performed using the *readManySeq* function. The result can be seen in Figure 4.4. Notice that the Y-axis is linear.

From the gained results, one can see a positive correlation between queue depth and throughput for sequential asynchronous reads. Therefore, by using a larger queue depth, throughput can be improved. However, as it was for block size, there is a threshold. Using a queue depth greater than 16 will not further increase the throughput for sequential asynchronous reads.

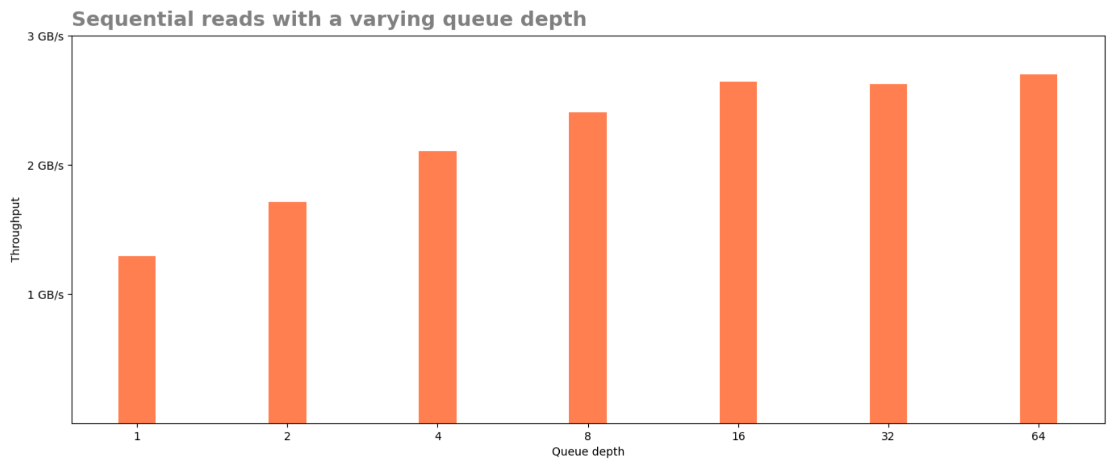

Figure 4.4: Queue depth impact on sequential read performance. The file size is 4GB. The entire file is read. Y-axis is linear.

4.2.2 Random reads using different queue depths

When using asynchronous reading for conducting random reads, one must also consider how the read request size influences the choice of queue depth. Since it might occur that one queue depth is optimal for one request size but not for another, both the read request size and the queue depth are examined in this test. This experiment made use of the *readManyRandom* function and a 1GB file. The result can be seen in Figure 4.5. The Y-axis has a logarithmic scale.

The result shows that the performance of multiple queue depths was very similar for larger random reads, but the difference in performance was larger at minor read requests. Overall, a queue depth of 32 or 64 had similar results throughout the entire test.

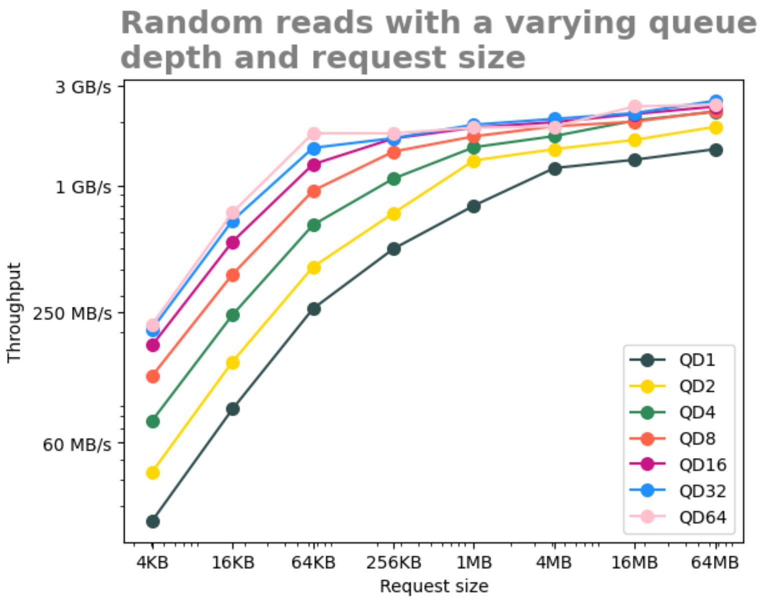

Figure 4.5: Queue depth and read request size impact on random read performance. File size is 1GB. Entire file read. Y-axis is logarithmic.

4.2.3 Choosing the optimal queue depth

Since the result shows that a queue depth of 16, 32 and 64 gave a similar performance for sequential reads and a queue depth of 32 and 64 gave a similar performance for random reads, it is logical to apply a queue depth of 32 when using the asynchronous reader. Therefore in all following experiments, a queue depth of 32 is used. However, an important thing to note is that this result might have been different if other hardware or OS were used.

4.3 MappedReader chunk size

As mentioned in the theory chapter, Windows has an internal parameter called memory-mapped file chunks size. This parameter can differ between different versions of Windows. Even though a developer can not change this internal parameter, knowing what this internal parameter is will help analyze the outcomes of other experiments.

To be able to estimate what the memory-mapped file chunk size was for the Windows version on the test computer, an experiment was executed using the *readOneBlock* function. In this experiment, the single block latency was measured for *MappedReader*. The results can be seen in Figure 4.6. In this graph, the Y-axis scales linearly.

By looking at the result, one can notice that the increase in latency follows a pattern. After the block size has been increased by 32KB, the latency is increased by 250 microseconds. Based on the results, it can be concluded that the memory-mapped file chunk size is 32KB. This means that data will always be swapped in as 32KB chunks, and it is not possible to swap in less since this is an internal parameter in Windows. Later, this knowledge is used to analyze the results from the other experiments.

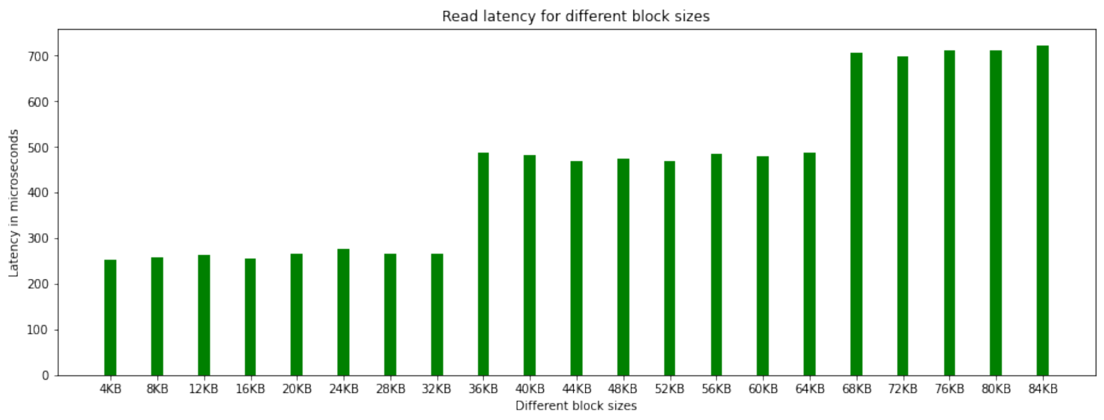

Figure 4.6: Finding the memory-mapped file chunk size. File size is 1GB. Y-axis is linear.

4.4 Comparing readers

As mentioned earlier in this paper, several reading techniques can be used to read data. These techniques have been implemented as readers such that they can be compared. However, which reader is best to use is not a simple question to answer.

In the following part, results gained from several experiments will be presented.

In each experiment, one specific parameter is in focus. The results of the single-threaded reading experiments are divided into two sections. In the first section, readers are compared for a sequential access pattern, and in the second section, the readers are compared for a random access pattern.

4.4.1 Sequential reads

In this section, how the different readers compare against each other for sequential reads is examined. Note that both graphs presented in this section use a logarithmic scale on the Y-axes.

4.4.1.1 File size

The purpose of conducting the *file size experiment* was to examine how file size affects throughput for sequential reads. An experiment was conducted using the *readSeq* function, in which a fixed amount of bytes was sequentially read in files of different sizes. Files of sizes from 1MB to 4GB were used, and the first 1MB was read from each file. The result can be seen in Figure 4.7. In this experiment, the results were inconsistent. Therefore, the graph looks noisy.

By looking at the results in Figure 4.7, one can see that file size impacts *MappedReader*, *BufferReader4KB* and *StramReader*. Furthermore, one can also see that *BufferReader4MB* and *AsyncReader* have the most outstanding throughput for all different file sizes that were tested.

4.4.1.2 Total bytes read

An experiment was conducted to examine how the total bytes read in a file impacts throughput. In the experiment, different amounts of a 1GB file were read. The *readSeq* function was used for this experiment. The results are shown in Figure 4.8.

The results show that in contrast to file size, the total bytes read in a file significantly impacts sequential read performance. Reading more data in a file results in increased throughput for all readers.

Another aspect worth pointing out is that similar to the result gained from the *file size experiment*, *AsyncReader* and *BufferReader4MB* showed the best sequential read performance. Regardless of the total bytes read, the results show that these two readers will consistently outperform the other readers.

4.4.1.3 Discussion on sequential reads experiments

Two parameters were looked at to examine how the readers compare against each other for sequential reads. The results gained from the *file size experiment* show

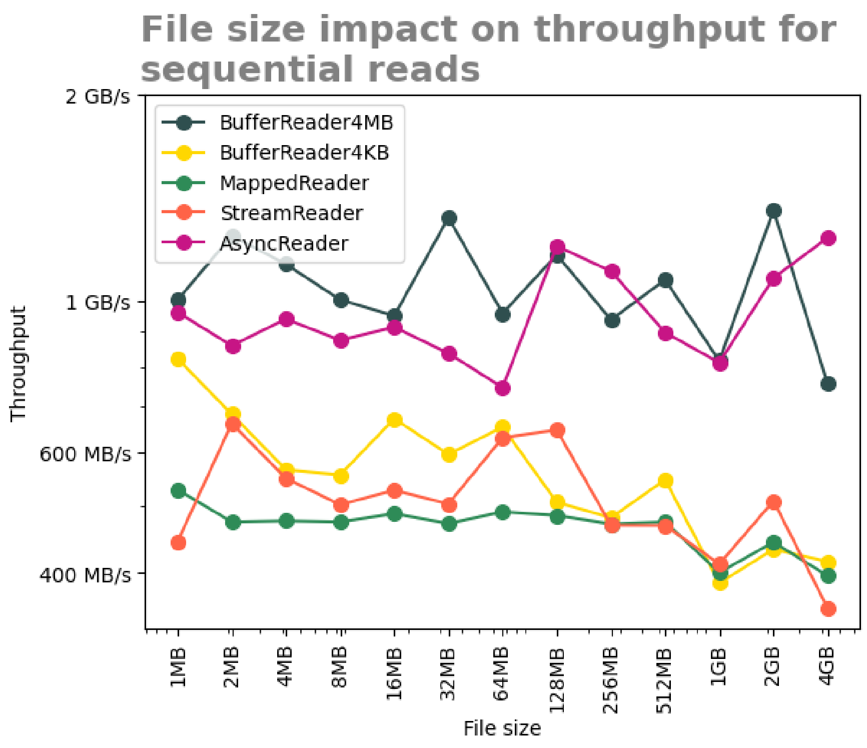

Figure 4.7: File size impact on sequential read performance. The total bytes read are 1MB. The Y-axis is logarithmic.

that file size seems to impact performance for some readers. Though, when it comes to buffered readers, one cannot make a reliable conclusion since the results were so inconsistent. *MappedReader*'s performance was more consistent and loosed nearly 20% throughput when opening a file of size 4GB compared to a file of 1MB. One explanation for this is that, when opening a file with *MappedReader*, it will map the entire file into memory. Hence, one expects mapping a larger file to take more time than mapping a smaller one.

In contrast to the file size, the total bytes read impacts all readers performance. Reading a larger amount of the file results in higher throughput. An explanation for this behavior is that prefetching can be more utilized when more data is read. In addition, a larger read implies the possibility of using a larger block size, which results in higher throughput. *BufferReader4MB* and *AsyncReader* performed similarly up until a total of 4MB was read. After this point *AsyncReader* began issuing multiple simultaneous requests which resulted in *AsyncReader* having a higher throughput compared to *BufferReader4MB*. Although *MappedReader* has advantages over buffered reading, such as no context switches and no unnecessary memory copies, it is never faster than using buffered reads with large block sizes.

In none of the experiments regarding sequential reads did *BufferReader4KB* perform best. Since *BufferReader4KB*, will always use a block size of 4KB, it will never be able to outperform *BufferReader4MB* for sequential reads. *StreamReader* did not outperform any of the other readers in any sequential experiment. Previous

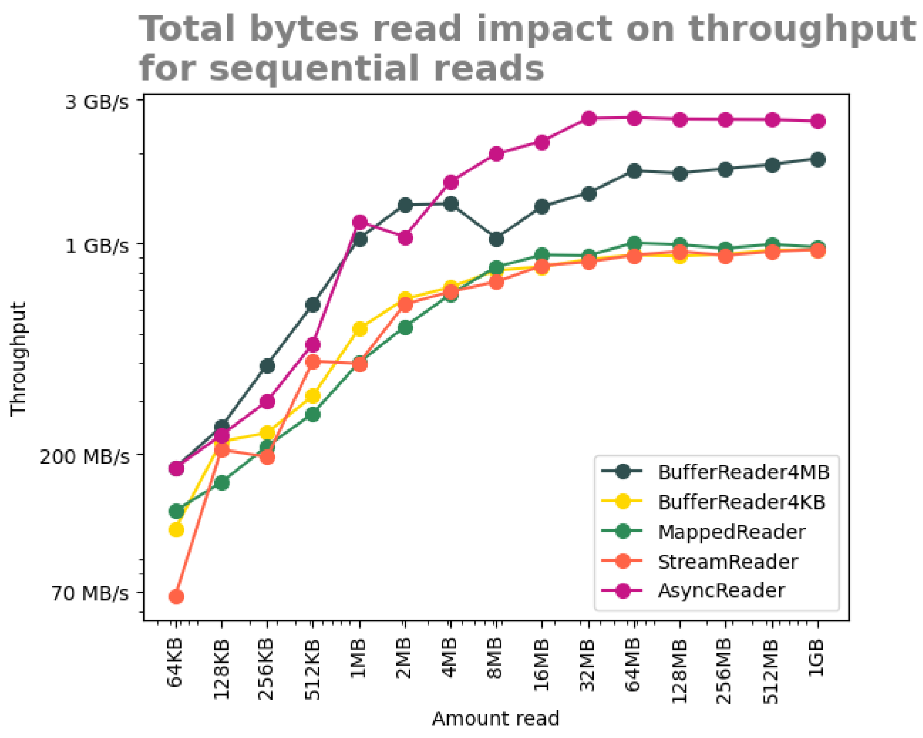

Figure 4.8: Total bytes read impact on sequential read performance. The file size is 1GB. The Y-axis is logarithmic.

studies have shown that *StreamReader* in certain circumstances is slower than *BufferReader4KB* [9]. The reason that *StreamReader* was slower than *BufferReader4KB* was that it has a greater CPU overhead. The CPU cost becomes noticeable if the cache is warm or the disk is very fast. However, since all of our experiments are run on cold cache, the effect of *StreamReader*'s unnecessary CPU cost became less notable.

4.4.2 Random reads

The results gained from the sequential experiments showed that using buffered read with a large block size had the best performance for sequential reads. In none of the experiments did *MappedReader*, *StreamReader*, and *BufferReader4KB* perform better. However, this is not the case for random reads. As will be seen in this section, using *MappedReader* is beneficial in certain situations. Notice the Y-axis has a logarithmic scale in all five graphs belonging to this section.

4.4.2.1 Read request size

When examining how different readers perform for random reads, an additional parameter must be considered. That parameter is the read request size, which is how much data is requested for each random read. The following experiment was conducted to investigate how the different readers perform for different read request sizes. For each read request size, the function *readManyShuffled* was used, and an entire 1GB file was read. The results are presented in Figure 4.9.

By looking at the results from the read request experiment, one can see that in contrast to the experiments regarding sequential reads, which reading technique that has the highest throughput depends on the examined parameter. In this experiment setup, using memory-mapped files for random reading chunks less than or equal to 4KB outperforms synchronous buffered reading. At 1KB or less, memory-mapped file reading also outperforms asynchronous reading. Somewhere between 1KB and 4KB, asynchronous reading begins to outperform all other readers. At 16KB, *BufferReader4KB* and *StreamReader* do a big leap in throughput and outperforms both *MappedReader* and *BufferReader4MB*. For all examined request sizes larger than 64KB, *BufferReader4MB* was the fastest of the synchronous readers.

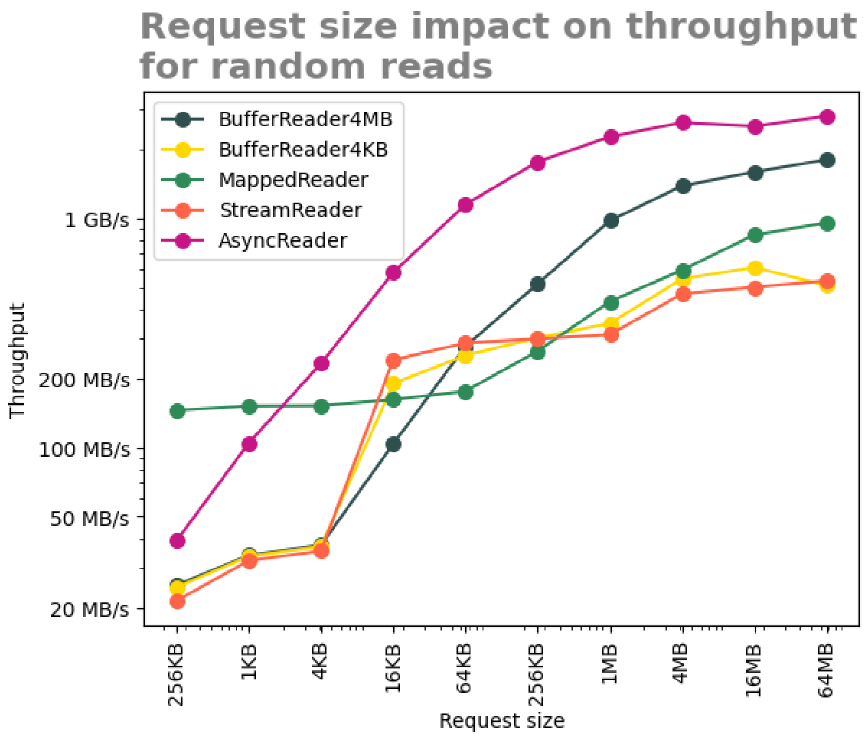

Figure 4.9: Request size impact on random read performance. File size is 1GB. Entire file read. Y-axis is logarithmic.

4.4.2.2 File size

In the previous experiment, it was shown that *MappedReader* outperformed the other readers when the read request was less than or equal to 4KB. However, both the file size and the total bytes read were fixed in that experiment. The file size, as well as the total bytes read, were set to 1GB. Therefore, these two parameters also have to be examined.

An experiment was conducted in which only a total of 1MB were read in files of different sizes. Moreover, the read request size was set to 4KB. The reason that a read request size of 4KB was used was to examine if *MappedReader* would still outperform the other synchronous readers as the file size changed. In this experiment, *readManyShuffled* was once again used. The results can be seen in Figure 4.10.

Although the graphs are noisy, it is easy to notice a trend. *MappedReader* is impacted by file size since its throughput drops when reading larger files. However, this is not true for the other readers since they have approximately the same throughput regardless of the file size.

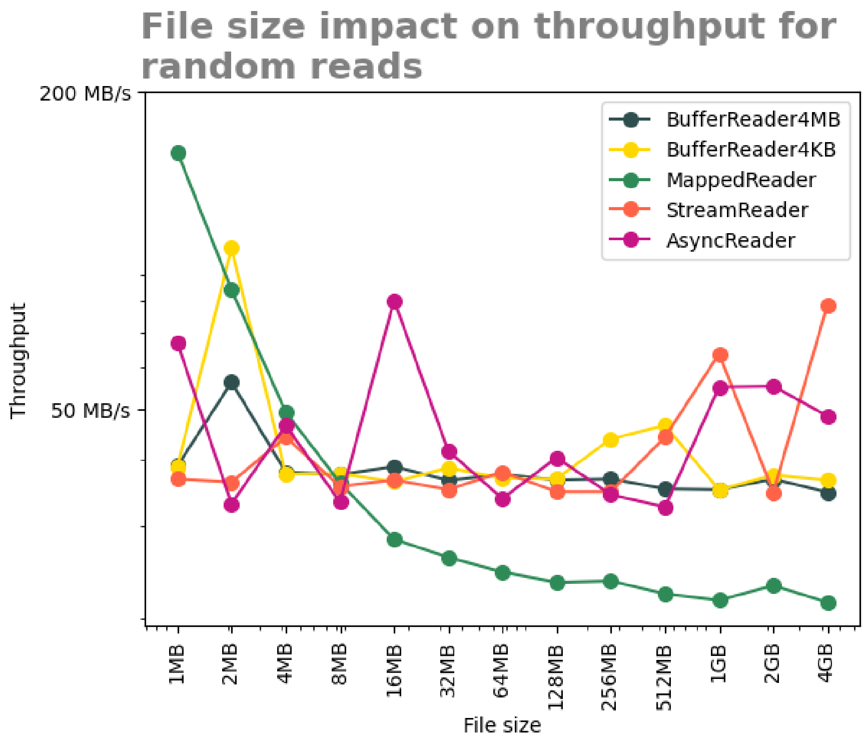

Figure 4.10: File size impact on random read performance. Read request size is 4KB. Total bytes read are 1MB. Y-axis is logarithmic.

4.4.2.3 Total bytes read

To examine how the total bytes read in a file impacts throughput for random reads, two experiments were conducted. To run the experiments, the *readManyShuffled* function was used. In both experiments, a 1GB file was used, and the total amount of data read changed in between each test.

For the first experiment, a read request size of 4KB was used to see how the performance of *MappedReader* would be affected in comparison to the other readers. The results can be seen in Figure 4.11.

In the results gained from the first *total bytes read experiment*, one can see that, similar to the *file size experiment*, the parameter has a big impact on the throughput of *MappedReader*. Reading more bytes in a file will improve *MappedReaders* throughput.

In the second experiment, another read request size was used. As was shown in the *read request size experiment*, for request sizes between 4KB and 64KB, *BufferReader4KB* and *StreamReader* performed best out of the synchronous readers. To examine if this always is the case, the request size was set to 12KB. The results are presented in Figure 4.12.

In the second experiment, a pattern similar to the one in the first *total bytes read experiment* appears. Namely, reading a larger portion of the file improves the throughput of *MappedReader*. But in this case, it also improves the throughput of *StreamReader* and *BufferReader4KB*. However, in comparison to the first experiment, *BufferedReader4MB* is surpassed at a larger amount read, at about a quarter of the file size.

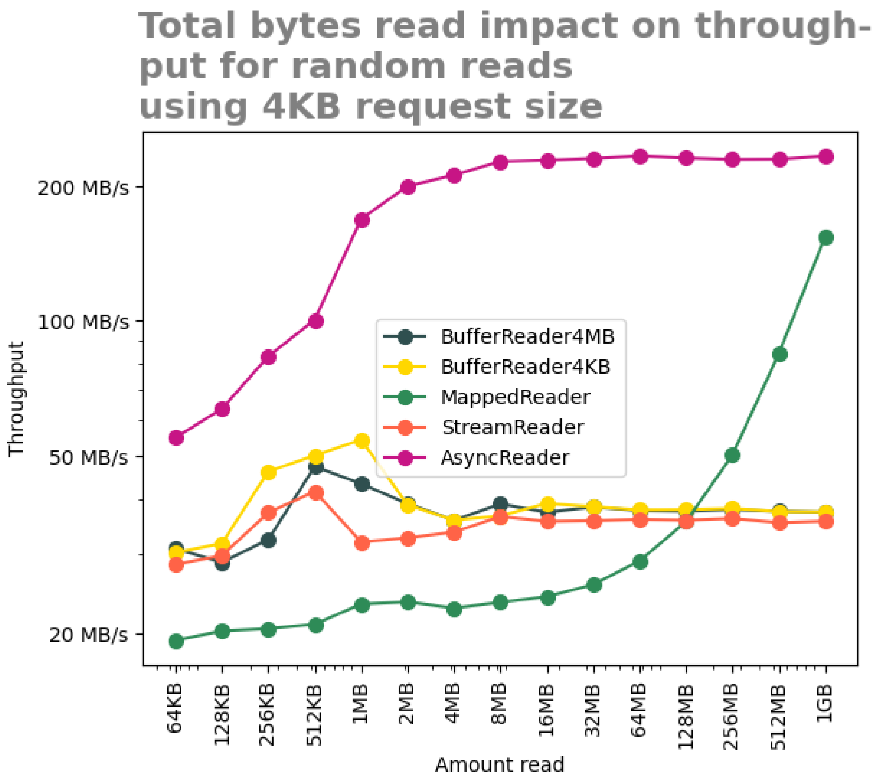

Figure 4.11: Total bytes read impact on performance. Read request size is 4KB. The file size is 1GB. Y-axis is logarithmic.

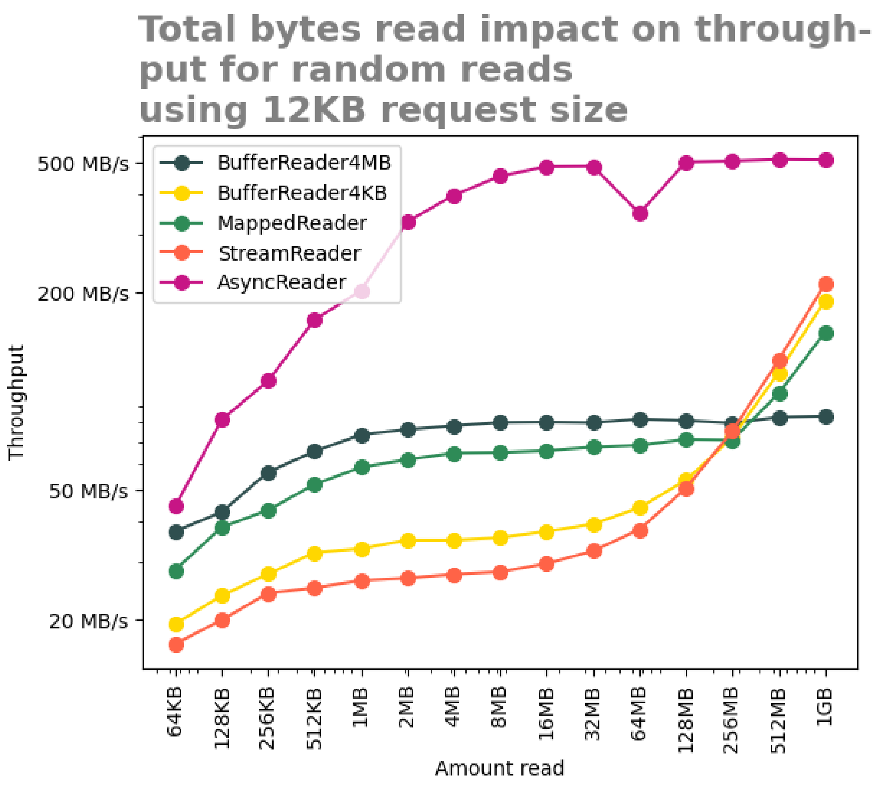

Figure 4.12: Total bytes read impact on performance. Read request size is 12KB. The file size is 1GB. Y-axis is logarithmic.

4.4.2.4 Reading more than memory

The largest file read in the previous experiment was of size 1GB. Another use case for a system can be to read vastly larger files. From prior studies regarding using memory-mapped files for reading data, it has been shown that the throughput given by memory-mapped files drops when reading a file larger than memory [7]. However, their experiment was run on a Linux machine. A similar experiment was run on Windows to examine if memory-mapped reading has the same flaw.

Forty files of size 1GB were randomly read using memory-mapped files. After each file was read, the throughput was calculated. Since the files were not closed after the reading was done, they remained in memory. After the memory-mapped reading was done, the same procedure was performed with a buffered reader. The result from the experiment can be seen in Figure 4.13.

From the graph one can see that before the memory is full, the throughput of *MappedReader* stays at about 150MB/s, but as soon as the memory gets full, the throughput drops to around 30MB/s. This is just 1/5 of *MappedReader*'s initial throughput.

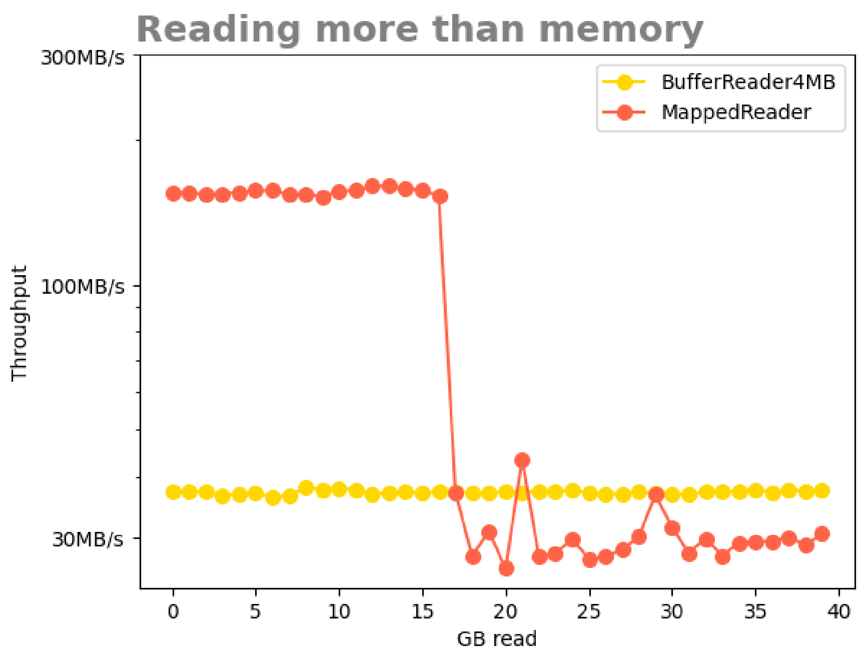

Figure 4.13: Reading more than memory impact on performance. 40 files of size 1GB are read. Read request size is 4KB. Y-axis is logarithmic

4.4.2.5 Discussion on random reads experiments

The results gained from the sequential experiments implied that using asynchronous and synchronous buffered reading is the best option for sequential reads. In contrast, which reading technique is best to use for random reads depends on several parameters. Nevertheless, the results have shown that *MappedReader* is a clear competitor under certain circumstances.

An advantage that *BufferedRead4MB* and *AsyncReader* have over the other readers is that they can read data in large chunks. However, it is not always possible to read in large chunks when doing random reads since the read request size limits the block size. For example, if the system has to read small records of 4KB in a large file, *BufferReader4MB* and *AsyncReader* will not be able to utilize the advantages gained from using a larger block size. This can be seen in the *request size experiment* 4.9. When the read requests gets larger, *BufferReader4MB* and *AsyncReader* can use a larger block size, and hence the throughput increases.

MappedReader was able to perform better in the read *request size experiment* for smaller request sizes. However, as the *file size* and *total bytes read experiments* show, a small read request is not the only parameter affecting the throughput of *MappedReader*. One also needs to read a large portion of the file. In the experiments using a 4KB request size, approximately 13% of the file had to be read for *MappedReader* to surpass *BufferReader4MB* in throughput.

Why does reading a larger portion of the file positively influence the throughput for *MappedReader*? This is most likely because data is already in memory. At the beginning of this chapter, it was shown that *MappedReader* will always swap in at least 32KB of data. In both the first *total bytes read experiment* and *file size experiment*, a read request size of 4KB was used. When reading the entire file, that is, if the total bytes read equals the file size, *MappedReader* peaks in throughput. In that scenario, for each 32KB block in the file, there will be only one actual read from secondary memory. The remaining seven 4KB blocks are already in memory. When reading a smaller portion of the file, the chance of finding the requested 4KB in memory becomes smaller. In the worst-case scenario, for each 4KB read request, *MappedReader* will have to swap in 32KB and will never find the requested data in memory. If data can be found in memory, the read operation goes very fast. Moreover, *MappedReader* does not have to do any unnecessary memory copy from kernel space to user space, nor any context switches. Hence, the *MappedReader* is faster when the read request is small, and a larger portion of the file is read.

Moreover, a similar behavior was found in the results gained from the second *total bytes read experiment*, with a 12KB request size. In that scenario, reading a more significant portion of the file positively influenced *BufferReader4KB* and *StreamReader*. The likely reason that *BufferReader4KB* and *StreamReader* performed better when reading a larger portion of the file has to do with prefetching. When reading many subsequent 4KB blocks, the OS prefetches a more significant portion of the file. This can be seen in the second *total bytes read experiment* 4.12. When a larger portion of the file is read, *BufferReader4KB* and *StreamReader* is more likely to find the prefetched data in memory.

Lastly, how *MappedReader* and *BufferReader4KB* performed when there was more data to be read than available memory was examined. The results gained from the *reading more than memory experiment* were similar to the results of previous studies. When reading more than memory, *MappedReader* becomes significantly slower [7].

4.5 Parallel reads

The sequential and random reads experiments were conducted using only a single thread. In this section, it is examined if using multiple threads can increase the performance of the readers. Similar to previous sections, the experiments are run for both sequential and random reads. Note that all Y-axes in the following figures use a logarithmic scale.

4.5.1 Parallel sequential reads

Firstly, it is interesting to see if sequential reads can be sped up using multiple threads. The function *readSequentialParallel* is used to read an entire 1GB file with a block size of 4MB. The results can be seen in Figure 4.14.

From the graph, one can see that the *BufferReader4MB* always performs best and that the throughput peaks when using two to four threads. Using more threads than four leads to a decrease in throughput. The other readers are not even close in terms of throughput to the *BufferReader4MB*.

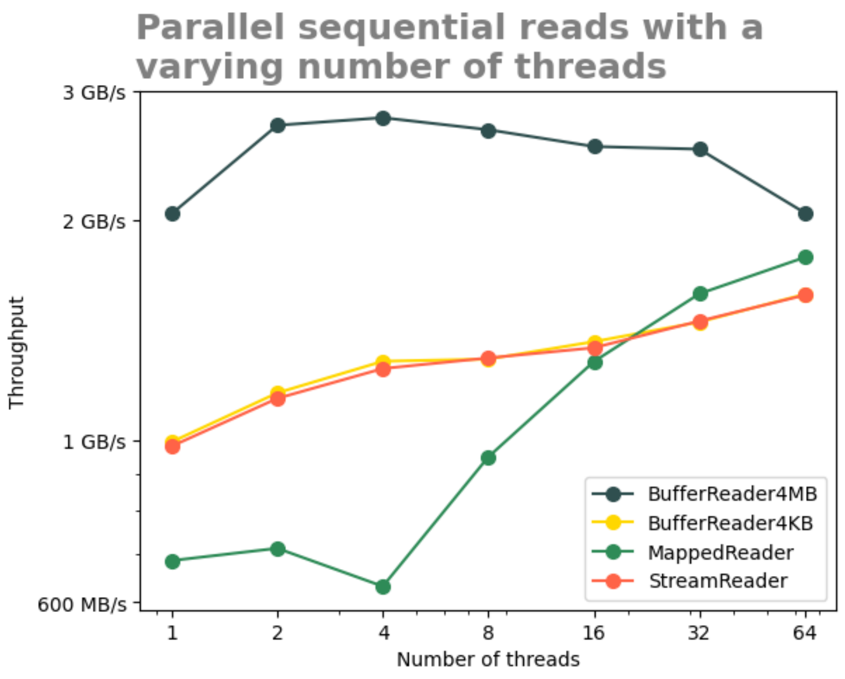

Figure 4.14: The number of threads impact on parallel sequential read performance. File size is 1GB. Entire file is read. Y-axis is logarithmic.

4.5.2 Parallel random reads

Next up is the two *parallel random reads experiments*. Like in the sequential test, multiple threads are used to see if throughput can be increased. This is done using the test function *readManyShuffledParallel*. The results when using a read request size of 4KB can be seen in Figure 4.15 and the result when using a read request size of 4MB can be found in Figure 4.16.

The results show that when the read request size is small, using the memory-mapped technique is always better, and that throughput is almost increasing linearly, for all readers, with the number of threads. As for a large read request size of 4MB, the graph looks very similar to the *parallel sequential reads experiment*. Buffered reading is preferable, and throughput peaks when using four threads.

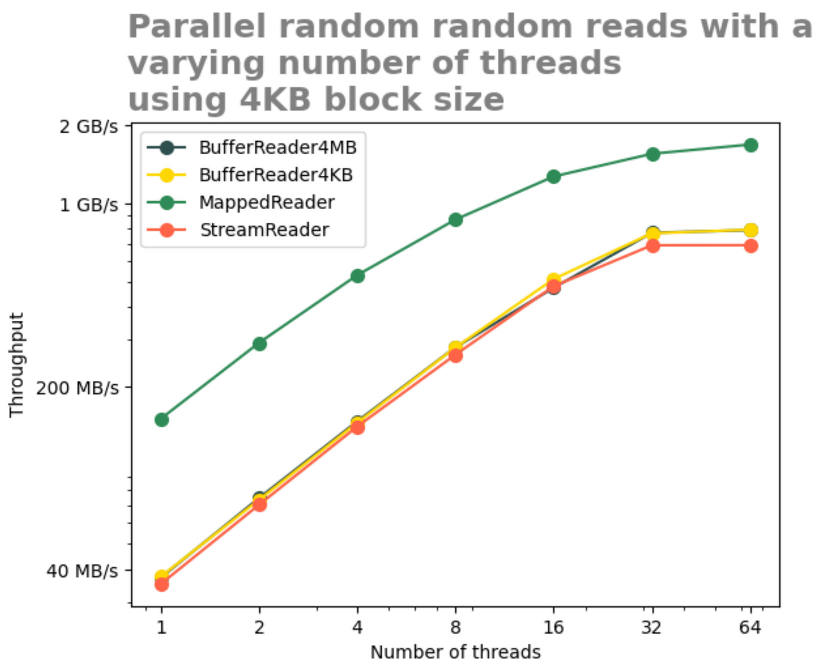

Figure 4.15: Number of threads impact on parallel random read performance. File size is 1GB. Entire file is read. Request size is 4KB. Y-axis is logarithmic.

4.5.3 Discussion on parallel reads experiments

As noted in the experiment, buffered reading performs best and will benefit from using two to four threads in the case of sequential reads. This is because simultaneous request enables the SSD to utilize its internal parallelism better. The throughput drops when using more than four threads. The reason for this is that a higher amount of threads leads to higher overhead handling the threads, together with the fact that the maximum possible throughput of the SSD has already been reached when using four threads. The conclusion drawn from this is that one can amplify the throughput of sequential reads significantly by just using one extra thread.

When looking at the random reads with a small read request size, one can see that memory-mapped reading always results in the highest throughput. However, the peak throughput is never reached, and performance increases with the number of threads used. A higher-performing processor might get the highest possible throughput, but in this case, one must consider if all that processing power is not put to better use elsewhere. In the case of larger read request sizes, the result is, as noted in the experiment, very similar to the sequential one. Once again, buffered reading is the best performing, although in this case, peak throughput is reached when using four threads.

The takeaway from these experiments is that which reading technique has the highest throughput for a specific workload does not change when using multiple threads. Instead, using multiple threads to read files can be an excellent way to increase throughput further. Furthermore, if one wants to fully utilize the performance of

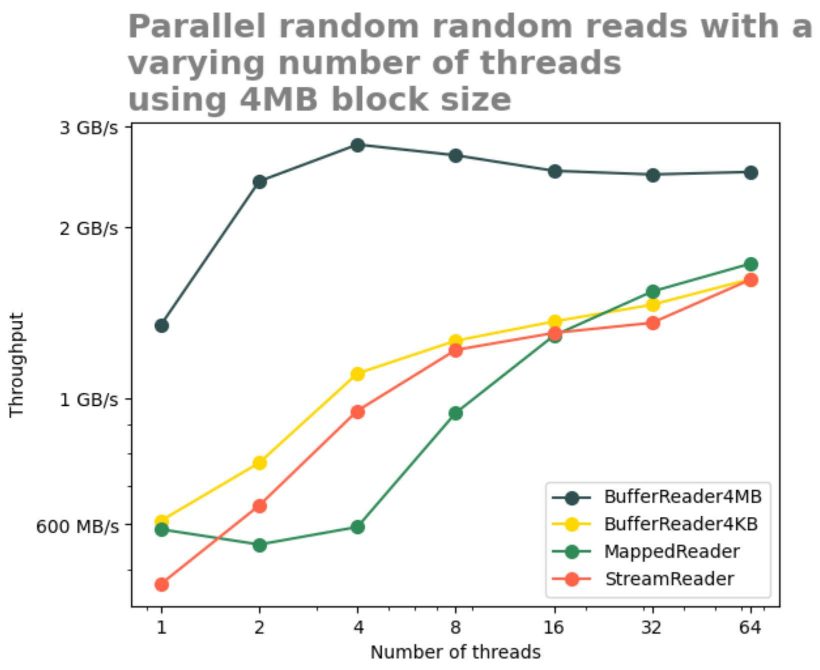

Figure 4.16: Number of threads impact on parallel random read performance. File size is 1GB. Entire file is read. Request size is 4MB. Y-axis is logarithmic.

the NVMe SSD using synchronous reads, taking advantage of multiple threads is required.

4.6 Integrating asynchronous reading

Asynchronous reading was shown to be the best alternative for multiple workloads. Using asynchronous reading was the only way to fully utilize the NVMe SSD's capabilities on a single thread and often outperforms all other options. When it comes to sequential reads, asynchronous reading can be used much like parallel reading to increase performance. It is, however, essential to note that the random reads experiments offer ideal conditions for asynchronous reading since all the read operations are known beforehand.

In addition to read requests not being known beforehand in a real-world application, using asynchronous reading comes with many other considerations that need to be handled to reach good performance. For example, one might want to interleave asynchronous read requests with computation to make the system more effective. A question related to this endeavor is how to handle the asynchronous read queue. Should you send all requests at the same time they come in or wait for multiple requests to send them simultaneously? When and how should you wait for a request to finish? Providing an optimal answer to these questions is out of scope for this thesis. Additionally, asynchronous reading is more complicated to integrate than synchronous reading and might be difficult to integrate if one wants to use different reading techniques based on the situation.

4.7 Guidelines

The results gained from the experiments show that by using asynchronous reading instead of synchronous reading, one can significantly improve performance. *AsyncReader* showed the best performance in all experiments except the scenario of random reading a larger portion of the file with a read request size of less than 1KB. However, as pointed out in the previous section, it can be difficult and sometimes impossible to utilize asynchronous reading to its fullest. Hence, it is recommended that the demanding developer examine if asynchronous reading can be integrated into their system. However, the developer should have in mind that, based on gained results, using memory-mapped files is faster than asynchronous reading for random reading of a larger portion of the file with a read request size smaller than 1KB. Therefore, the best overall reading strategy is to swap between memory-mapped reading and asynchronous reading depending on the workload.

Even though a developer cannot or does not want to use asynchronous reading, it is still possible to optimize reading. Therefore, a set of general guidelines are now presented for optimizing a read-heavy system by considering its workload and utilizing the advantages that come with NVMe SSDs. The guidelines are not dependent on the application and are therefore easy to implement in real-life system.

The results regarding single-threaded sequential reads have shown that *BufferReader4MB* consistently outperforms the other synchronous readers regardless of the value of the examined parameter. Therefore, based on the results from the experiments, the following is suggested for sequential reads.

- Use buffered read for sequential reads.
- Choose block size according to the conditions given in 4.1.4.

In contrast to the guidelines for sequential reads, for random reads, the choice of reading technique depends on the workload. The discussion regarding single threaded random reads concluded that using memory-mapped files is only preferable over buffered read if the read request size is small. Moreover, it was shown that cache hits were very important for memory-mapped file reading. To predict how often one gets a cache hit is difficult. One way of predicting it is to assume that the offsets are uniformly distributed and that the read request size is 4KB. As mentioned in the previous discussions, if the read request size is 4KB and the offsets are uniformly distributed, when reading more than 13%, memory-mapped file reading is preferable. Otherwise, a buffered read with a large block size is preferable. Although this is a very rough estimate, it gives an idea of how one can reason about the number of cache hits. Because of the hard assumptions, in-house benchmarking is recommended.

Moreover, it was shown that for random reads, *StreamReader* and *BufferReader4KB* performed equally good or better than *MappedReader* for read requests that were between 4KB and 32KB. However, in order for *BuffedReader4KB* and *StreamReader* to be as good as *MappedReader*, a large portion of the file had to be read. Hence, for simplicity's sake, the guidelines will suggest using *MappedReader* in that interval since the reader is most often the best alternative.

There was also one particular case in which *MappedReader* performed worse. That case was when more data was read than available memory. The OS won't evict memory-mapped pages until there is no memory left. Once this happens, performance will drastically decrease.

Based on the abovementioned points, the conditions below have been formulated as guidance. These conditions should be considered every time a file is opened for reading. If all conditions evaluate to true, memory-mapped reading should be used. Otherwise, one should use buffered reading with the same block size rules as sequential reading.

- The amount of data to be read from the file is less than the available memory.
- The reads to be made are at non-sequential offsets, i.e., random.
- It is expected that a larger portion of the file will be read.
- The read request size is less than or equal to 32KB.

Once the choice of reading technique has been made, one can decide if multiple threads can be used to execute the read.

The guidelines have been visualized as a flowchart and can be seen in appendix B. In the flowchart, the conditions have been grouped into levels. There are three different levels, system, file, and request level. To evaluate if a condition at the system level is true, the developer needs to have knowledge about the system as a whole. At the file level, the developer needs to know how the file is accessed; and at the request level, the developer only needs to have knowledge about the particular request.

Parallel reading can, as shown experimentally, increase the throughput in many cases. Multiple threads do not change what read technique to use but rather amplify the throughput. If there is a big sequential read to be done, use two or four threads to maximize the throughput. For random reads, if the reads are small, which often is the case, use all the threads that are available in the application's thread pool. For larger random reads, four threads are enough.

These recommendations work in the scenario that only one file is handled at a time. However, many applications might want to process multiple files at a time. In that case, one must do in-house benchmarks to see how to best divide the work between the threads. Therefore, how one uses multiple threads depends on the system at hand.

5 Benchmarks

To evaluate the guidelines, a new reader has been implemented, called *AdaptiveReader*. The reader was based on the guidelines presented in the previous chapter. Moreover, a custom-built benchmark has been developed. The benchmark mimics the behavior of a read-heavy system, namely the Carmenta Engine. The new *AdaptiveReader* has, together with the other readers, been evaluated for three different Carmenta Engine use-cases. The outline of this chapter is as follows. Firstly, *AdaptiveReader* will be described. After that, the results gained from the benchmarks will be presented.

5.1 AdaptiveReader

The main idea behind *AdaptiveReader* is to swap between memory-mapped file reading and buffered reading by following the earlier presented guidelines. However, the conditions that are not on the request level cannot be evaluated by merely observing a read request. For example, when receiving a request, the reader cannot know if the requested file will be accessed several times in a random read pattern. Therefore, in contrast to the other readers, a flag is passed as an argument when the developer wants to open a file with an *AdaptiveReader*. The flag is called the *swapFlag* and should only be set if and only if the following three statements are all true.

- The amount of data to be read from the file is less than the available memory.
- The reads to be made are at non-sequential offsets, i.e., random.
- It is expected that a larger portion of the file will be read.

In the case that *AdaptiveReader* opens a file and the *swapFlag* is set to false, it will work exactly as *BufferReader4MB*. However, if the *swapFlag* is set to true, then *AdaptiveReader* will decide which reading technique to use by looking at the read request. When the *swapFlag* is true, for a read request, if the request size is less than or equal to 32KB, *AdaptiveReader* will use memory-mapped file reading. If the request size is larger than 32KB, then a buffered read with a large block size will be used.

In a real-world scenario, the developer would set the *swapFlag* based on their knowledge about their system's workload. However, in a benchmark, the flag has to be set programmatically. To set the flag, an analysis of the log files was conducted. Since the total size of all of the files read from in the benchmark is less than memory, it's impossible to read more than memory. Moreover, by analyzing the log files, it was noticed that, for the most part, a larger portion of each file was randomly read. Hence for simplicity's sake, each time a file was opened by *AdaptiveReader*, *swapFlag* was set to true.

5.2 Carmenta test-suite

The Carmenta Test Suite is a set of tests that are used to validate and evaluate the Carmenta Engine. Two benchmarks based on the log files from the Carmenta test suite were used to evaluate the readers. The first benchmark is called the "Carmenta test-suite benchmark" (CT), and the second one is called the "Animation files benchmark" (AM). Note that the Y-axes in the following two graphs are linear.

5.2.1 Carmenta test-suite benchmark

CT contains all of the read operations that are executed in the Carmenta test suite. A total of 652.88MB are read. Of the total bytes read, 9.2% were requested with read request sizes larger than 32KB. The results gained from the benchmark can be seen in Figure 5.1.

The result given from the CT benchmark shows that *MappedReader* and *AdaptiveReader* had the lowest latency. *BufferReader4KB* and *StreamReader* got the highest latency. In this benchmark, *AdaptiveReader* was approximately 2.5 times faster than *BufferReader4KB*.

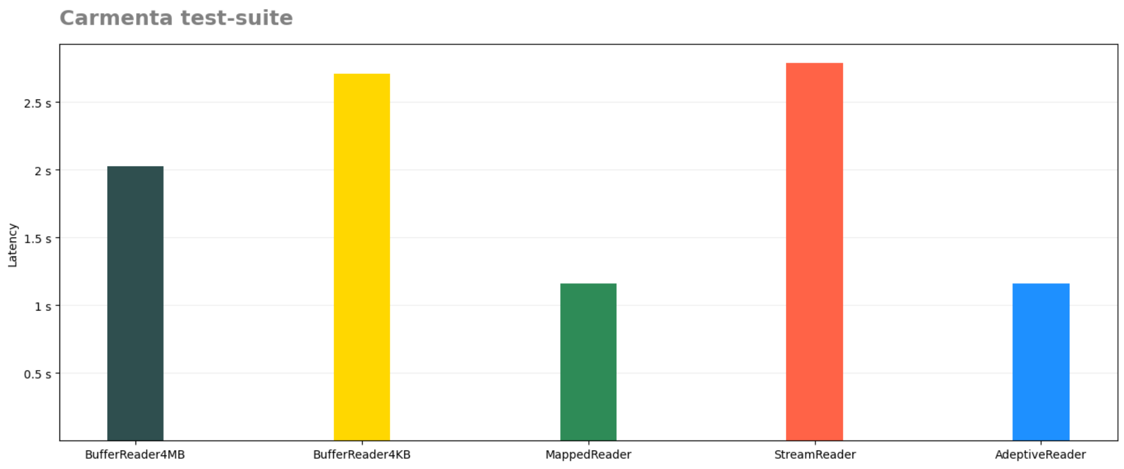

Figure 5.1: Results gained from Carmenta test-suite benchmark. Y-axis is linear.

5.2.2 Animation files Benchmark

As CT, AM is based on a log file from the Carmenta Test suite. However, what differs is that AM only contains a small part of all of the executed read operations in the Carmenta test suite. The process that is examined concerns reading animation files. In AM, a total of 7MB are read. However, in comparison to CT, almost all data read is requested with read request sizes larger than 32KB. Of the total bytes read, 99.8% were requested with read request sizes larger than 32KB.

In contrast to the results from the previous benchmark, *BufferReader4MB* performed best. *AdaptiveReader* performed as good as *BufferReader4MB*. *MappedReader*, *BufferReader4KB* and *StreamReader* performed significantly worse.

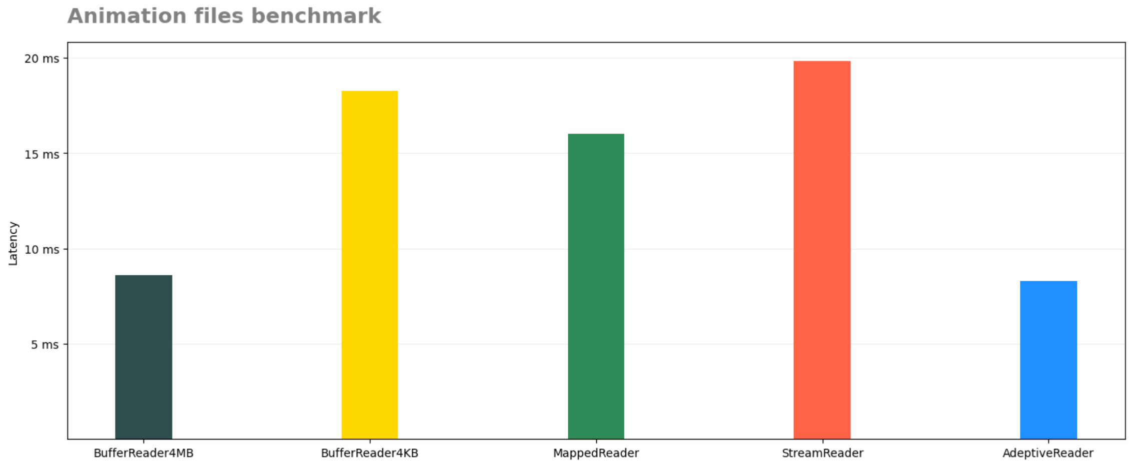

Figure 5.2: Results gained from Animation files. Y-axis is linear.

5.3 Interactive map

Interactive map (IM) mimics the use case in which a user interacts with a geospatial applications. The logs that the benchmark is based on come from an application in which a developer at Carmenta interacted with a map. A total of 2415MB were read during the benchmark. Of that amount, 0.6% were read in chunks larger than 32KB. The results are shown in Figure 5.3. The results gained from the Interactive map benchmark are similar to the ones gained from the Carmenta test suite. *AdaptiveReader* and *MappedReader* performed best.

5.4 Discussion of benchmarks results

The analysis of the log files showed that the three first statements of the guidelines were all true. Hence the choice of reading technique only depended on the read

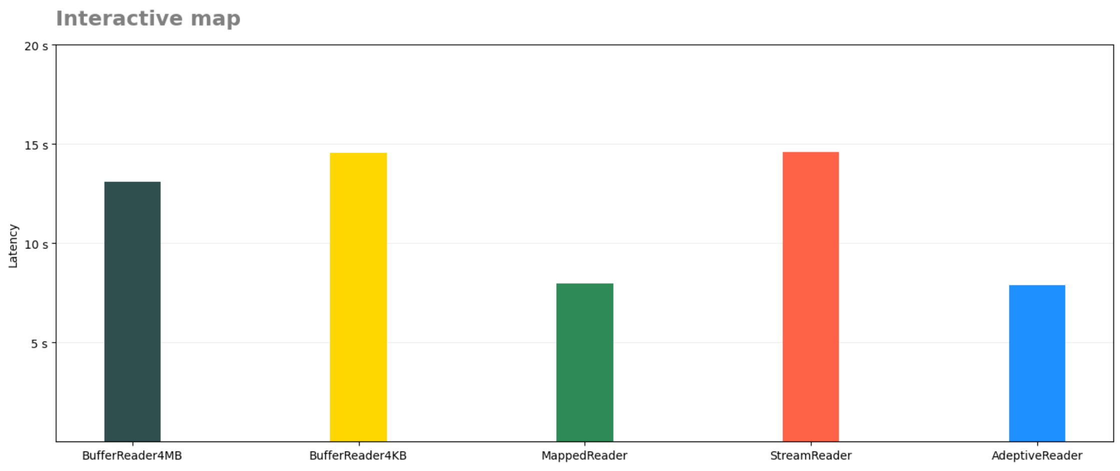

Figure 5.3: Results gained from Interactive map. Y-axis is linear.

request size. Since CT and IM contain many read requests that are less than 32KB and the three other statements were true, the guidelines suggest that *MappedReader* should be used. By looking at the results gained from CT and IM, it is clear that it is better to use memory-mapped reading over buffered reading since it gets a lower latency for this use case. In contrast to CT and IM, AM contains larger read requests. As the guidelines suggest, in that scenario, it is better to use *BufferRead4MB* than to use *MappedReader*.

AdaptiveReader performed as well as all of the readers in all three examined use-cases. However, it did not perform significantly better than the best non-adaptive reader. It did not perform much better because the benchmarks either contained almost only small reads or almost only large reads.

In CT and IM, only 9.2% and 0.6% of the total bytes read were read in chunks larger than 32KB. Hence *AdaptiveReader* will use memory-mapped file reading for almost all of the read operations and thus get a latency very similar to *MappedReader*. However, for AM, 99.8% of all bytes read are read in chunks larger than 32KB. Thus, *AdaptiveReader* will almost only use buffered read and get a latency very close to the one of *BufferReader4MB*.

In order to see a significantly lower latency for *AdaptiveReader* compared to the best non-adaptive, the workload has to contain a mix of larger and smaller read request sizes. In that scenario, neither *MappedReader* nor BufferReader4MB will be the best choice, and hence switching between techniques will be more crucial in order to get the highest performance.

Whilst the non-adaptive readers had their weaknesses for different use cases, *AdaptiveReader* was always as least as good as the best non-adaptive reader. Therefore, it can be concluded that reading can be optimized by utilizing knowledge about a system's workload and the available hardware.

6

Conclusion

In this paper, guidelines have been presented that can be used to reduce the I/O bottleneck in read-heavy systems. The guidelines are based on results from several experiments in which it was examined how multiple reading techniques perform for different workloads. Furthermore, a new reader was created based on the guidelines and evaluated with a custom-built benchmark that mimics the behavior of a real read-heavy system. The results gained from the benchmark indicate that the new reader, which adapts a reading strategy based on the guidelines, performs significantly better than a reader using a static strategy. Hence, it was concluded that reading could be optimized at the application level by utilizing knowledge about a system's workload and utilizing the advantages of an NVMe SSD.

7
Future Work

Optimizing data reading is a vast area of research, and there is a lot of research to be done that could expand on this work. Below are some suggestions on areas that could be expanded upon.

7.1 Create a better model for cache hits

It was shown in the experiments that the number of cache hits has an impact on which reading technique that performs best. Moreover, the guidelines suggest that it is beneficial to use memory-mapped files if the system should read a larger portion of the file. The suggestion is based on the idea that it is more likely to get cache hits if a larger part of the file is read.

However, this is only true if the read request size is 4KB and the memory-mapped chunk size is 32KB, which is not always the case. A ”larger portion of the file" is also an unspecific amount. It would be beneficial to have a better model for reasoning about cache hits. This is, however, not a simple problem, and there was not enough time to explore it in this thesis.

7.2 Testing other storage hardware

Another suggestion for future work is to perform the tests made in this thesis on other storage mediums than an NVMe SSD. For example, an HDD, SATA SSD, or a USB-connected SSD. One could also perform the experiments on different NVMe SSDs to see if there is a difference between brands, capacities, etc.

7.3 Exploring more options in Windows

When opening a file in Windows, the system's caching of data read from a file can be disabled using the FILE_FLAG_NO_BUFFERING option [1]. Since the behavior of the cache has been a major part of the consideration when designing the guidelines, completely disabling the cache can potentially lead to a different results. However, using the FILE_FLAG_NO_BUFFERING option comes with additional constraints regarding alignment and file access [1]. Therefore, due to the additional constraints, the flag's impact on reading performance was not examined.

Even if the cache is kept activated, hints about how the file is about to be accessed can be given to the OS when creating the file handle [21]. The hint for sequential read is FILE_FLAG_SEQUENTIAL_SCAN, and the hint for random read is FILE_FLAG_RANDOM_ACCESS. By using these hints, the OS can optimize how it caches data. The reason that these flags were not explored in this thesis was mainly due to time constraints.

7.4 Testing a different OS

Another interesting addition to this research would be to execute the experiments on a Linux machine to see if any guidelines would change. Since Linux is free open-source software, it provides more overall control over the system and the low-level software. On Windows, it was shown that when using memory-mapped files, 32KB were swapped in at each page fault which shaped the guidelines substantially. If this parameter can be manipulated in Linux, the guidelines might look different. Linux also provides a different interface for asynchronous I/O [34]. Comparing this to what is available on Windows might change the reasoning around asynchronous reading.

If the hardware were the same, conducting experiments on Linux would simultaneously be a good comparison of reading performance between Linux and Windows.

7.5 Explore DirectStorage

Microsoft DirectStorage for Windows was released in spring 2022 [16]. It is a technique to reduce overhead and CPU workload when loading data from secondary storage into a system's graphic processing unit (GPU). It is mainly intended for games but can perhaps also be used in other kinds of applications. For example, if a system utilizes the GPU to accelerate computations, it might benefit from using DirectStorage to load data into the GPU.

Bibliography

[1] alvinashcraft, v kents, DCtheGeek, drewbatgit, mijacobs, and msatranjr. File buffering. https://docs.microsoft.com/en-us/windows/win32/fileio/file-buffering. Accessed: 2022-05-27.

[2] Apple. File system formats available in disk utility on mac. https://docs.microsoft.com/en-us/troubleshoot/windows-client/backup-and-storage/fat-hpfs-and-ntfs-file-systems. Accessed: 2022-03-18.

[3] OpenMP ARB. The openmp api specification for parallel programming. https://www.openmp.org/. Accessed: 2022-05-24.

[4] Carmenta. Carmenta engine. https://carmenta.com/en/geospatial-technologies/carmenta-engine. Accessed: 2022-05-17.

[5] Raymond Chen. How do i prefetch data into my memory-mapped file? https://devblogs.microsoft.com/oldnewthing/20120601-00/?p=7483. Accessed: 2022-03-16.

[6] cplusplus.com. <iostream>. https://www.cplusplus.com/reference/iostream/. Accessed: 2022-05-18.

[7] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are you sure you want to use mmap in your database management system? 2022.

[8] Cem Dilmegani. Synthetic data in-depth synthetic data guide: What is it? how does it enable ai? https://research.aimultiple.com/synthetic-data/. Accessed: 2022-05-30.

[9] Niall Douglas. P1031r2: Low level file i/o library. Technical report, C++ Standards Committee, 2019.

[10] Alexandra (Sasha) Fedorova. Why mmap is faster than system calls. https://sasha-f.medium.com/why-mmap-is-faster-than-system-calls-24718e75ab37. Accessed: 2022-05-18.

[11] Evangelos C Fradelos, Ioanna V Papathanasiou, Dimitra Mitsi, Konstantinos Tsaras, Christos F Kleisiaris, and Lambrini Kourkouta. Health based geographic information systems (gis) and their applications. *Acta Informatica Medica*, 22(6):402, 2014.

[12] CreateFileW function (fileapi.h). Createfile. `https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew`. Accessed: 2022-03-20.

[13] Emmanuel Goossaert. Coding for ssds. `https://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents`. Accessed: 2022-05-17.

[14] Johnson M. Hart. *Appendix C*, page 575–591. Addison-Wesley, 4 edition, 2010.

[15] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. I/o acceleration with pattern detection. In *Proceedings of the 22nd international symposium on High-Performance Parallel and Distributed Computing*, pages 25–36, 2013.

[16] Cassie Hoef. Directstorage api now available on pc. `https://devblogs.microsoft.com/directx/directstorage-api-available-on-pc/#comments`. Accessed: 2022-05-24.

[17] Randy Kath. Managing memory-mapped files. `https://docs.microsoft.com/en-us/previous-versions/ms810613(v=msdn.10)?redirectedfrom=MSDN`. Accessed: 2022-03-16.

[18] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware i/o management for solid state disks (ssds). *IEEE Transactions on Computers*, 61(5):636–649, 2011.

[19] Stan Lanning. Understanding how i/o workload profiles relate to performance. `https://education.dellemc.com/content/dam/dell-emc/documents/en-us/2014KS_Lanning-Understanding_How_IO_Workload_Profiles_Relate_to_Performance.pdf`. Accessed: 2022-05-23.

[20] Sungjin Lee, Jihoon Park, Kermin Fleming, Jihong Kim, et al. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE Transactions on consumer electronics*, 57(4):1732–1739, 2011.

[21] Microsoft. Createfilea function (fileapi.h). `https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea`. Accessed: 2022-05-27.

[22] Microsoft. Createfilemappingw function (memoryapi.h). `https://docs.micro`

soft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-createfilemappingw. Accessed: 2022-03-20.

[23] Microsoft. Default cluster size for ntfs, fat, and exfat. https://support.microsoft.com/en-us/topic/default-cluster-size-for-ntfs-fat-and-exfat-9772e6f1-e31a-00d7-e18f-73169155af95. Accessed: 2022-05-25.

[24] Microsoft. Mapviewoffile function (memoryapi.h). https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-mapviewoffile. Accessed: 2022-03-20.

[25] Microsoft. Ntfs overview. https://docs.microsoft.com/en-us/windows-server/storage/file-server/ntfs-overview. Accessed: 2022-03-18.

[26] Microsoft. Overlapped structure (minwinbase.h). https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-overlapped. Accessed: 2022-03-20.

[27] Microsoft. Overview of fat, hpfs, and ntfs file systems. https://docs.microsoft.com/en-us/troubleshoot/windows-client/backup-and-storage/fat-hpfs-and-ntfs-file-systems. Accessed: 2022-03-18.

[28] Microsoft. Readfile function (fileapi.h). https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile. Accessed: 2022-03-20.

[29] Microsoft. Supporting asynchronous i/o. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/supporting-asynchronous-i-o. Accessed: 2022-03-17.

[30] Microsoft. Synchronization and overlapped input and output. https://docs.microsoft.com/en-us/windows/win32/sync/synchronization-and-overlapped-input-and-output. Accessed: 2022-03-18.

[31] Microsoft. Rammap v1.61. https://docs.microsoft.com/en-us/sysinternals/downloads/rammap, 2021. Accessed: 2022-05-31.

[32] Chuck Paridon. Storage performance benchmarking guidelines-part i: workload design, 2010.

[33] Abdulqawi Saif, Lucas Nussbaum, and Ye-Qiong Song. *On the Impact of I/O Access Patterns on SSD Storage.* PhD thesis, Inria, 2020.

[34] Ruslan Savchenko. Reading from external memory. *arXiv preprint arXiv:2102.11198*, 2021.

[35] Yongseok Son, Hara Kang, Hyuck Han, and Heon Young Yeom. An empirical evaluation of nvm express ssd. In *2015 International Conference on Cloud and Autonomic Computing*, pages 275–282. IEEE, 2015.

[36] Yongseok Son, Heon Young Yeom, and Hyuck Han. Optimizing i/o operations in file systems for fast storage devices. *IEEE Transactions on Computers*, 66(6):1071–1084, 2016.

[37] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems.* Pearson, 2014.

[38] Yaofeng Tu, Yinjun Han, Zhenghua Chen, Zhengguang Chen, and Bing Chen. Urfs: A user-space raw file system based on nvme ssd. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 494–501. IEEE, 2020.

[39] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[40] Tianming Yang, Ping Huang, Weiying Zhang, Haitao Wu, and Longxin Lin. Cars: A multi-layer conflict-aware request scheduler for nvme ssds. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1293–1296. IEEE, 2019.

A

Test computer

- Processor: Intel(R) Core(TM) i7-10850H CPU @ 2.70GHz 2.71 GHz
- RAM: 32GB
- Storage: NVME PC611 (1TB NVMe SSD)
- OS: Windows 10 64-bit

B

Guidelines

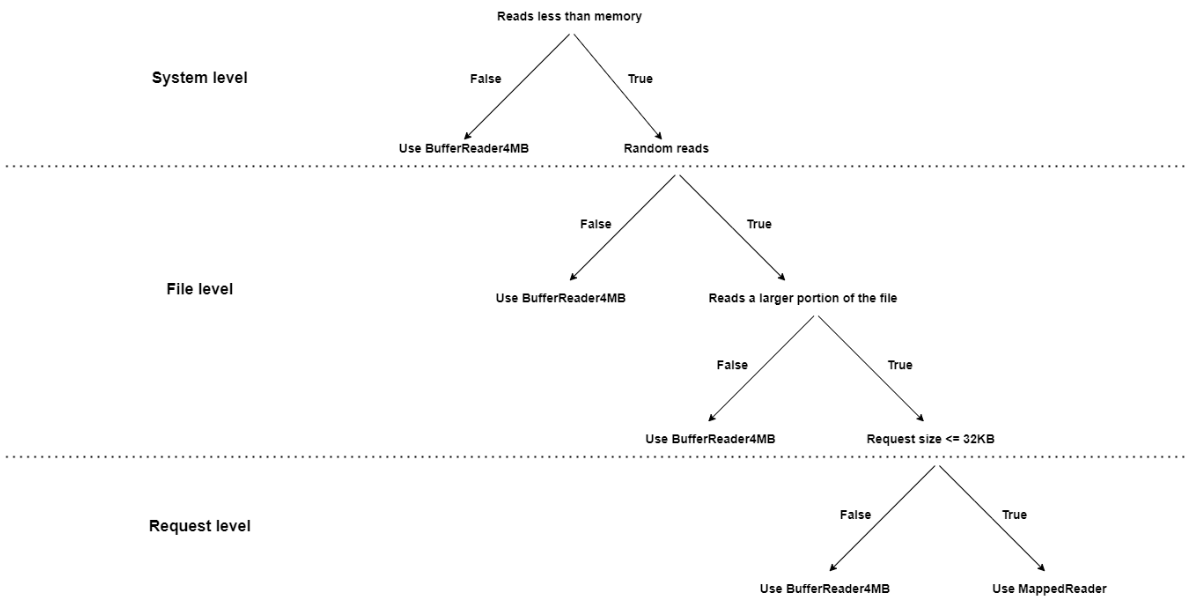

Figure B.1: Guidelines for reading data