Improving File Transfer Speed on different devices

Deep Patel
Computer Science Department
University of North Georgia
Dahlonega, USA
dpate2771@ung.edu

Abstract*— Most modern computers come equipped with high-speed processing CPUs, but they are never used to their full potential when transferring files/ folders. The goal is to improve the file transfer speed onto flash drives, preferably USB 2.0, 3.0, and 3.1. There are numerous approaches that allow for faster file transfer. However, just using one solution is not optimal and underutilizes the CPUs' capabilities. Instead, the proposal is to use a combination of these strategies or determine the best combination for a given hardware to optimize the file transfer. (Abstract***)**

Keywords—Speed copy, transfer, Robocopy, Ultracopier, Teracopy.

I. INTRODUCTION

Technology has evolved exponentially in the past 100 years. What seemed to be a work of sci-fi movies is now part of our daily lives. Phones today have more computing power than the computers that landed mankind on the moon. However, these computers are not used to their full potential. The resources available are underused, thus resulting in mediocre performance. Transferring files and folders onto a flash drive is a common occurrence in our daily life. It can take either two seconds or two hours to copy based on the size of the file and the hardware limitations. The purpose of this project is to exploit both to speed up this process. The reason behind this project is that while people upgrade their computers, they normally do not upgrade their flash drives. Transferring large files can be cumbersome as these flash drives suffer from low performance. The idea to is come up with a unique design, to expedite the transfer process to its limits.

II. RELATED WORK

A. ***Robocopy***

Robocopy is the most widely used copying software on Windows. Robocopy was the pioneer for multi-threaded copy ideas and set a standard for newly developing softwares. This software requires no installation as it is shipped with most Windows versions. However, on the other hand, Robocopy is only supported on Windows. There is no native graphical user interface (GUI) integrated with the software, which can be overwhelming for non-tech-savvy people.

B. ***Teracopy***

Teracopy builds upon Robocopy's idea and adds more features. Fast file transfer speeds, error recovery, pause, and resume transfer processes are a few additions. Like Robocopy, Windows is the only supported platform. The biggest downside is that it remains to be for-profit software. Many users have experienced interference with the software from anti-virus. Furthermore, there is no checksum verification built-in resulting in potentially corrupted files.

C. ***Ultrocopier***

Ultracopier is a powerful utility software and the closest work to this project. There are several noticeable features. The biggest highlight of Ultracopier is its multi-platform and architecture support. Most of the other softwares suffer from limited platform support, which can be an issue for other users. Other features include transfer suspension, error correction, and checksum verification to verify file integrity. However, some of these features are only supported via subscription. Also, symbolic links are only handled on Linux.

These are some of the most popular and widely used transfer utility software. However, they are not truly multithreaded when transferring one single file. Suppose the user wants to copy a file that is 100GB in size, all of them will do it in a single-threaded fashion. The problem is it is difficult to duplicate a file when bits must be placed in a specific order. For example, the header of the file cannot be written at a different offset than its original location (the top portion) otherwise, that would corrupt the file. Therefore, a new and improved version of this is being proposed as a part of this project.

III. Design

As mentioned in the previous section, it is hard to achieve multi-threading when transferring a single file. The idea stems from replicating the file rather than modifying it in different ways that would improve the overall performance. There are numerous ways to accomplish this. All these approaches are listed below but note that all of them will not be implemented in the project's beginning phase due to time constraints.

The most basic implementation on Windows and most other OS is single-threaded [4]. When transferring a folder with multiple files it would be ideal to use multiple threads to copy different files at once. This is the path taken by the software mentioned in the above section. All these softwares have seen some kind of improvement in speed, thus it would be naive to not implement a well-tested and approved method. This leads to another problem, blocking or synchronous code which inherited ally suffers from poor performance. Therefore, everything is built on top of async I/O which will keep compute thread separate from the copying task [3]. Another benefit of having an async runtime is having "batteries included". More specifically, green threads. Green threads are like regular OS threads but do not have the overheads that come with OS threads. Different programming languages have different names for them like, Go's goroutines, Kotlin's coroutines, and Erlang's processes. The most important thing to note about green threads is they are non-blocking. Typically, when an OS thread performs I/O or must synchronize with another thread, it blocks, allowing the OS to schedule another thread. When a task cannot continue executing, it must yield instead, allowing the runtime to schedule another task. Tasks should generally not perform system calls or other operations that could block a thread, as this would prevent other tasks running on the same thread from executing as well. Instead, the runtime provides a way for running blocking operations in an asynchronous context.

Compression is one of the most common methods to reduce file size. When a file is compressed, the overall data is downsized by the algorithm, which results in fewer bytes. Older flash drives have slower write speed therefore reducing the number of bytes would increase the transfer speed. Most modern websites use some kind of compression on the server, and compressed data is transmitted over the internet to reduce the size of the data [1]. Furthermore, all files excluding regular text files, are compressed somehow by the respective formatting tool. For instance, JPEG, JPG, PNG, etc. are all compressed versions of RAW file format, which can range in size from 20MB to 100MB. So, utilizing the appropriate compression based on the file type can further improve efficiency [5]. However, no compression algorithm will work with all file types effectively. So, the idea is to use a compression algorithm that yields the best results with fewer compression iterations.

When transferring files, the OS must deal with opening and closing the file. This can become cumbersome when the program is constantly opening and closing multiple files. Hence, using a larger buffer size can reduce the number of times the OS has to open a file. Additionally, copying a file is an I/O-bound task. More sequential calls could result in slower speed. To overcome this issue, some OSes come equipped with NVRAM (non-volatile memory), which stores frequently accessed files for faster reading [2]. Although, it is not a dependable source as not all file systems implement such a feature. The idea is to find the correct buffer size to efficiently transfer a file [3]. This is known as Buffered Writer. The buffered writer copies the data to memory and makes infrequent system calls (which are slow compared to the CPU) when the buffer is full. Better yet, the proposal is to implement Adaptive Buffer Writer. Adaptive Buffer Writer, contrary to Buffer Writer, changes the size of the underline buffer on demand to improve efficiency. For example, if it takes 0.5ms to write 8KiB as well as 16KiB, it is better to use 16KiB of buffer size as that has a faster write speed compared to smaller 8KiB chunks. It is important to know that this is just a proposal and is currently not implemented due to time constraints.

The last approach to truly achieve multi-threading on a single file is to split the file into multiple parts and write those parts on individual files. This is because a logical cursor can only be in one place at any given time. The main reason anyone transfers a file onto a flash drive is to either back up the file or move it to a different computer. Neither of the cases requires the file in its entirety. Later, whenever the user needs to use the file, they can simply assemble the parts using this program. The motivation for this strategy comes from the fact that writing speeds are generally slower than reading. Suppose a file is to be split and written to a flash drive, the program can read from offset 0 to 1024 and begin writing the stored buffer, while reading the offset 1025 to 2048 and subsequently write that into a different file from the previous offset. To better understand this, look at the image below which illustrates this process.

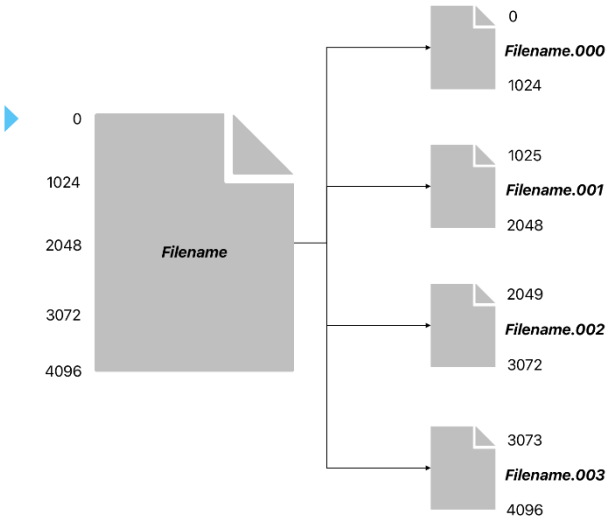

Fig. 1 – File splitter in a nutshell

When the user inputs a given file, the file splitter will allocate resources based on availability. After that, it will spawn four threads, each for a parting file, and begin reading from respective offsets. The logical cursor, denoted using the blue arrow, will constantly move based on the remaining offsets. Whenever the buffer reaches the capacity, it will append the data to the individual file. It may seem that this would increase the number of system calls, but it doesn't largely. Buffer Writer would have made these system calls anyway, so parallelizing does indeed improve performance, at least in theory.

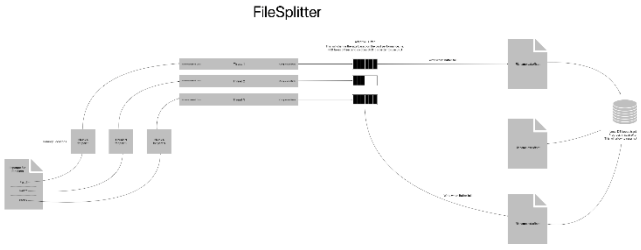

Fig. 2 – Full FileSplitter Architecture Design

On the other end, the assembler merges parted files in reverse order. This is assuming reading happens on a different computer, and computers have either HD or SSD, and it has three times the write speed. File Assembler is a single thread because the logical cursor can only be at one location at a given moment. This can be seen in the figure below.

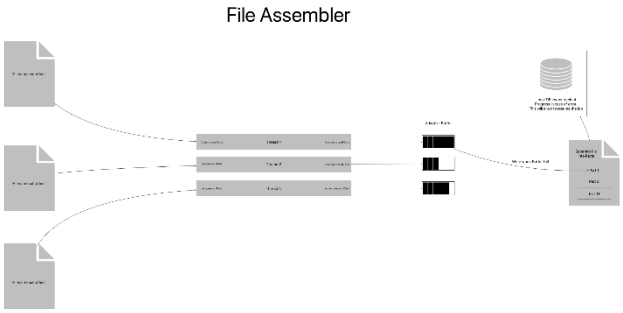

Fig. 3 – File Assembler Architecture Design

The file assembler is logically multi-threaded, but the final write-out is single-threaded because the file must be in its original format for the user to use it. This should be faster nonetheless because the SSDs and hard drives have higher write speed, at least compared to flash drives.

Note: There are problems with these approaches as mentioned in the result section.

IV. Implementation

The sole purpose of this project is to achieve maximum speed when transferring files onto a flash drive. This is a performance-critical task, so using the right programming language is very crucial. All high programming languages are very slow since they run in some kind of virtual machine, sandbox, or just-in-time (JIT) compiler. Additionally, these languages have a garbage collector (GC) which runs quite often. The advantage is that memory is managed by the compiler rather than the programmer. The downside is it runs very often taking up CPU time. Therefore, it was decided to use a low-level language. Traditionally, all performance-critical apps are written either in C or C++ because they provide granular control over CPU and memory management. Based on the table below, C/C++ has ranked top in all categories.

Total

	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Fig. 4 – Programming Language Comparisons [7]

Albeit C has the best performance, it is considered a legacy programming language. There are several drawbacks, which C++ suffers from as well, even though it is a level up from C. Perhaps the most noticeable flaw is it requires manual memory management which can easily cause memory leaks, segmentation faults, and other memory-related issues. Deallocation of memory at the wrong moment could result in dangling pointers. Most modern languages have strict type checking that catches many bugs at compile time, but C/C++ has implicit type checking, meaning any variable could change to other types. Contrary to common beliefs, C/C++ has limited standard library support, so it becomes a problem when writing portable applications. Numerous compilers have been written for these languages, which can become cumbersome when using a third-party library that used a different compiler. These were just a few problems of a longer list. Considering everything mentioned above and in the list, I chose a new and upcoming language that has comparable performance as C/C++ without its drawbacks - Rust.

As an individual who has been writing code in Rust for nearly two years, it is very different from all traditional programming languages. While version 1.0 was only released 8 years ago (May 2015), fairly new compared to other languages, it is regarded as one of the most loved language. If the code compiles it will run without any runtime bugs. Rust introduced a new compile time tool known as Borrow Checker. The Borrow Checker imposes two rules: Ownership and Borrowing (mutable or immutable). To understand these topics, take a look at the following code.

```
61  fn main() {
62      let mut a: i32 = 0;
63      let b: i32 = 1;
64      Foo(&a);
65      Bar(&mut a);
66      Baz(a);
67  }
68
69  fn Foo(a: &i32) {
70      /* Code */
71  }
72
73  fn Bar(a: &mut i32) {
74      /* Code */
75  }
76
77  fn Baz(a: i32) {
78      /* Code */
79  }
```

Fig. 5 – Rust sample code

The main function on line 61 will be executed when the program runs. It will create the variables `a` and `b`, which implies that the main function owns these variables. On line 64, the `Foo` is called, which takes a reference (borrows) of `i32`. This only allows the `Foo` function to read the variable but not modify it anyways. If the programmer does try to mutate, the compiler will raise an error. The following line of code takes a mutable reference of `i32`. So, the `Bar` function has read as well write access to that variable. On line 66, the `Baz` function takes ownership of the variable `a` because the function definition of `Baz` does not take parameters with a reference (&). So, when `Baz` is done executing (line 79), it will deallocate the memory of the variable `a`. If the programmer tries to access the variable `a` from line 66 onwards the compiler would not allow it since that memory is removed from the stack. Finally on line 67, the variable `b` is deallocated.

Rust also introduces Lifetimes. Lifetimes determine how long each variable lives in a given context. For instance, in the above example, variable `b` lives until the end of the function `main`. In simpler words, Lifetimes fall under the ownership model. All references cannot outlive the owner. Take the following example

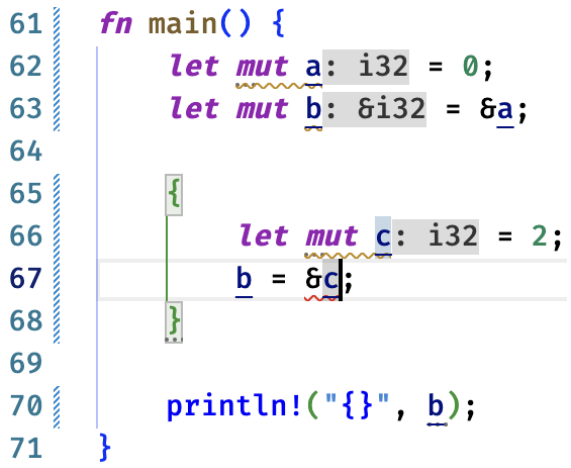

Fig. 6 – Lifetime example

On line 63, variable `b` takes reference to variable `a`. Inside the closure (65 - 68), variable `c` is created and its reference is assigned to `b`. However, it does not work since line 68 will destroy the variable `c` but `b` will still have a pointer to any empty memory cell (dangling pointer) which is an issue. It is possible to shorten a lifetime, but it is not possible to extend one. This is known as variance.

By simply implementing Borrow Checker, the Rust compiler makes it safer to work with pointers/ references. Additionally, Borrow Checker also ensures there is only one mutable reference at a given time. This guarantees no race conditions in the program. Furthermore, the Rust compiler comes with two toolchains, Cargo and Rustup. Cargo is the package manager, which allows for easy integration of third-party libraries. It is as easy as running a single command from the root of the project. Rustup is a command-line toolchain that can do cross-compilation and update the compiler. Considering all of this, it was an easy decision to choose Rust.

However, there is a problem with the programming language. Rust does not have a runtime, so there is no way to write async code. The language supports it, but it does not have an executor, which will schedule tasks. To combat this issue, there is a popular third-party library - Tokio. Tokio is divided into multiple modules to avoid dead code. It was mainly built for writing server/ network async code, but as demand has grown many new features have been included since its release, notably async I/O. Tokio uses a non-blocking I/O model, which allows the Rust program to parallelize its execution while separating blocking I/O. Additionally, Rust does not have built-in support for GUI since it requires a significant amount of platform-specific code and integration with the operating system. Another reason why Rust does not have built-in support for GUI development is that the language is still relatively young and growing, with a focus on performance, reliability, and safety. Adding a built-in GUI toolkit would require significant resources and development effort and would likely take away from the language's core strengths. There are several popular Rust libraries for building GUI applications, such as GTK, Qt, and Druid, among others. These libraries provide bindings to platform-specific GUI toolkits, allowing Rust developers to build native GUI applications for various operating systems. However, it is not truly cross-platform because not all the features are supported on every platform. An innovative technology was recently released that uses HTML/ CSS/ JS to write native desktop applications. Tauri has grown in popularity lately, overtaking Electron - a cross-platform app with a few problems. Electron and Tauri work in the same ways, but the implementation of Tauri gives it a significant boost in performance and security.

Almost all modern websites are built using a framework to avoid common pitfalls. This project is no exception to this rule. The front end of this project will be written in Svelte. While Svelte may seem to be a framework, it is a transpiler. All of Svelte's code will be converted to plain Html, CSS, and JavaScript, unlike other frameworks, which add a ton of boilerplate code while giving the benefits of these frameworks.

V. Results

While the project did seem promising, the results are not. There are quite a few drawbacks within the implementation phase that potentially led to this moment. For the basis of the comparison and understanding of the results, the experiment was conducted on M1 Macbook Pro with 16GB memory and 256GB SSD. The built-in transfer utility tool took 65 seconds to transfer a 222.1MB file. Using the file splitter technique, it took 394 seconds (about 6 and a half minutes) to transfer the same file. This is using all of the implementation ideas combined.

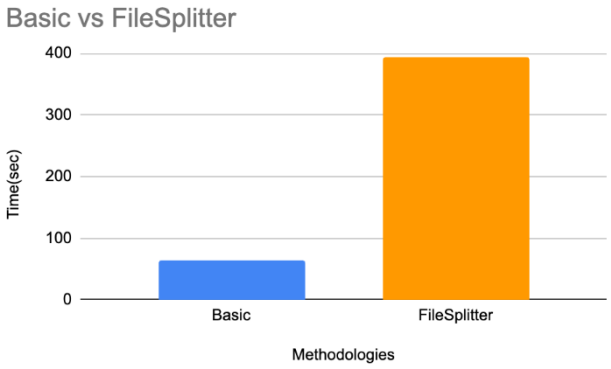

Fig. 7 – Basic vs FileSplitter

FileSplitter yielded worse results than expected. There were several factors contributing to it. Contrary to common belief, asynchronous I/O does not necessarily make a process faster. The reason for this is because asynchronous I/O involves a lot of overhead. Almost all platforms implement synchronous/ blocking I/O. The async is just a wrapper around it that constantly polls and checks if the operation is completed. This is normally done on a separate thread pool so that the main thread can continue executing with other parts of the app. Albeit, it seems an efficient way of working, it causes a lot of context switching - which can get expensive when there are several thread pools.

```rust
fn poll_next(
    self: Pin<&mut Self>,
    cx: &mut Context<'_>
)
    -> Poll<Option<Self::Item>> {
    if SomeTaskDone(cx) {
        return Poll::Ready(TheReturnValue);
    }
    return Poll::Pending;
}
```

Fig. 8 – Under the hood implementation of all Async

The code above has two parameters, self of type Pin and cx of type Context. The Pin type ensures that a given struct does not move in memory; Otherwise, the pointer to it would be invalid, causing the program to crash. The Context keeps track of the progress and calls it often to check if it is ready. This is how all the futures are written at a granular level. The executor/ runtime is a loop with a series of state machines that manages all the context. Once all of the futures have finished, the runtime can exit the program.

```rust
enum StateMachine<T> {
    NotReady(Context),
    Ready(T),
    Done,
}

fn main() {
    let mut state_machine = StateMachine::NotReady(cx);
    loop {
        match state_machine {
            StateMachine::NotReady(cx) => {
                if SomeTaskDone(cx) {
                    state_machine = StateMachine::Ready(TheReturnValue);
                } else {
                    state_machine = StateMachine::NotReady(cx);
                }
            },
            StateMachine::Ready(value) => {
                return Poll::Ready(value);
            },
            StateMachine::Done => {
                return Poll::Ready(None);
            },
        }
    }
}
```

Fig. 9 – Desugaring of the executor

For demonstration purposes, I wrote two simple test functions. These tests were performed under the same conditions with the same files. The first function is written in an asynchronous fashion. The second function is built using the standard library. I ran this test 3 times for each function and the results have been plotted in the graph below.

```rust
async fn test_regular_copy_async() {
    let src: PathBuf = PathBuf::from("../testing/bike.blend1");
    let dst: PathBuf = PathBuf::from("/Volumes/PNY 2/test_dst1/bike.blend1");
    let reader: File = tokio::fs::File::open(src).await.unwrap();
    let mut reader: BufReader<File> = tokio::io::BufReader::new(reader);
    let writer: File = tokio::fs::File::create(dst).await.unwrap();
    let mut writer: BufWriter<File> = BufWriter::new(writer);

    tokio::io::copy(&mut reader, &mut writer).await.unwrap();
}
```

Fig. 10 – Async Test function using tokio

```rust
fn test_regular_copy() {
    let src: PathBuf = PathBuf::from("../testing/bike.blend1");
    let dst: PathBuf = PathBuf::from("/Volumes/PNY 2/test_dst2/bike.blend1");
    let reader: File = std::fs::File::open(path: src).unwrap();
    let mut reader: BufReader<File> = std::io::BufReader::new(inner: reader);
    let writer: File = std::fs::File::create(path: dst).unwrap();
    let mut writer: BufWriter<File> = std::io::BufWriter::new(inner: writer);

    std::io::copy(&mut reader, &mut writer).unwrap();
}
```

Fig. 11 – Regular Test function using Std Library

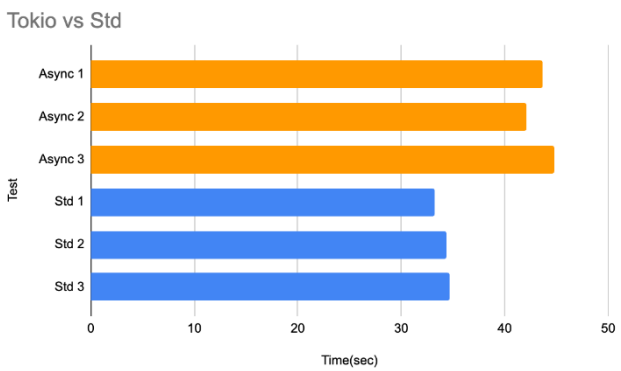

Fig. 12 – Tokio vs Std lib I/O comparison

As noticed the async I/O is slower than the std lib I/O even though the tokio is built on top of the std library.

Another reason why the original design failed was the CPU spent more time fighting for access to shared memory. Rust is inherently safe allowing multiple readers or one writer. Now this applies to all types and variables on one thread. However, this is a multi-threaded application, which added more complexity to the program. Rust only allows for one thread to have ownership of a variable; However, there are ways to work around this limitation. The std library has an 'Arc' (atomic reference count) type, which allocates the variable onto the heap. Although the prior rule still holds, there can be multiple readers or one writer, as long as the type encapsulated by Arc implements the 'sync' trait. This can be achieved by using either Mutex or RwLock. Mutex and RwLock have similar functions, but choosing the right type can significantly improve performance. Mutex is more restrictive than RwLock. Once a mutex lock is acquired the thread can either read or write, while all the threads wait to gain access. RwLock, on the other hand, allows multiple readers as long as there are no writers. Once a writer lock is issued, all the reader locks are revoked. Hence, RwLock is the right option for most tasks. Nevertheless, both of them will block the thread until the lock is acquired, and they can only hold it for up to 0.5ms. As soon as the thread hits this threshold, the lock is released for the other threads, which can cause the thread to stop before or mid-I/O execution. This can lead to a constant battle between threads for resources, essentially slowing down the program drastically.

Lastly, the compression degraded the app's performance. Compression is supposed to reduce the file size, but it is also possible for it to take a long time, depending upon the algorithm and the input. This project does not implement a proprietary algorithm, but it has a smarter way of determining which might be a good choice. Using the filename, type, and size, it is capable of deciding the correct algorithm. This is because no one algorithm can work for all file types, each algorithm has its strengths and weaknesses. There were four algorithms used for this project - Brotli, Bzip2, Xz, and Zstd. For the experiment, the program chose Bzip2 and completed the transfer in 79 seconds.

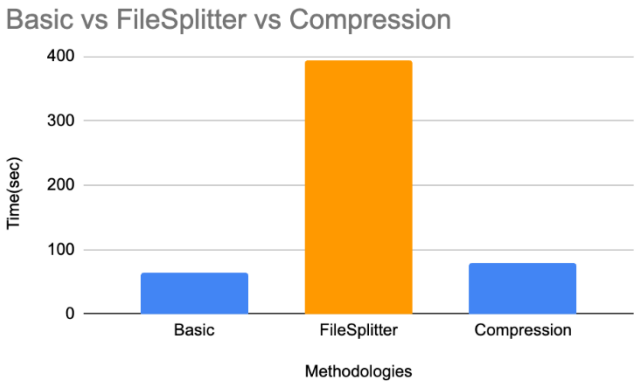

Fig. 13 – Comparison between Methodologies

Desperate for better results, I decided to redesign the architecture of the program.

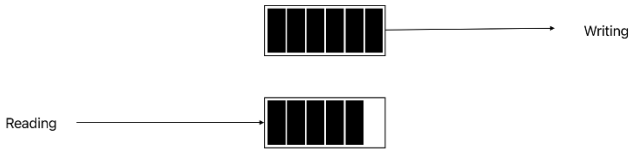

Fig. 14 – New multi-thread design

The new design performs reading and writing in parallel and avoids the pitfall of the previous design. This does not involve any kind of guarding mechanism, which means it won't block the thread, neither will it fight for resources. The program spawns two threads and buffers. The reading thread will read the chunk into buffer 1 until it's full. In the next iteration, the reading thread will read into buffer 2 while the writer writes from the buffer. The reader and writer will keep alternating until the end of the file. And this process can be spawned across multiple threads when transferring several files. This design resulted in a performance boost; However, it wasn't drastic. While using the same file, it took 48 seconds. In comparison to other approaches, this has thus far yielded the best results.

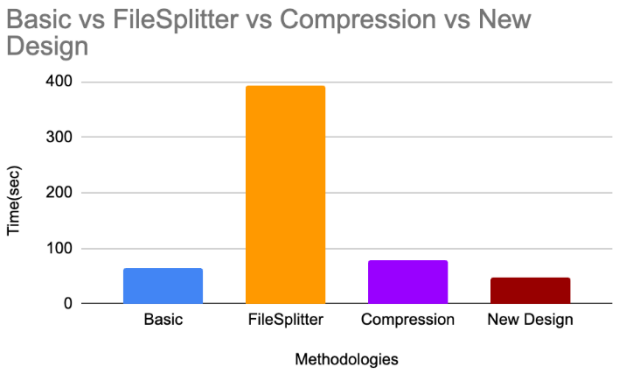

Fig. 15 – Comparison of new design from other methods

VI. Conclusion And Future Work

The project is just the beginning of a potentially larger product. Currently, it only supports multi-threaded copy. This is done uniquely compared to other approaches. There are mainly two reasons someone would copy data onto a flash drive. Firstly, transfer the file(s)/ folder(s) to a different computer. The second reason someone transfers data onto the flash drive is to store it for later use. The only requirement is that the final output needs to be the same as the original file. Considering these prerequisites, it is easy to form ideas around them to improve efficiency. Efficiency improvements tend to result in better performance. Generally, flash drives have low write speeds but high read speeds. This led to forming the filesplitter design. While it did see some success, it quickly struggled due to bottlenecks within the program. This was because each thread was constantly fighting for resources – CPU time, access to shared memory, and files. Additionally, the compression took too long to process. However, on the flip side, the compression algorithm worked wonderfully. The compressed file from the experiment was ~90MB, with approximately 2.1 compression ratio, which is not the best, considering the amount of time it took to compress. The reason to use a file splitter was to allow multiple logical cursors but it is not possible to do so on a single file hence this idea was introduced. The assembler is also multi-thread, but only if there are multiple files. This is because reassembling parts requires the file to be in order; otherwise, it is considered corrupted. As a last attempt to save this project, I decided to redesign the multi-thread model. One that does not involve a lot of overhead on CPU and I/O.

By no means is this project considered complete and can be further optimized to yield better efficiency. Also, it is important to know that these are I/O bound tasks, which are awfully slow. That is why it is important to ask if this project is even worth the time and labor. As mentioned above, there are numerous projects, and all of them seemed to have reached their peak performance due to I/O limitations. There are no official benchmarks because every computer is different. However, there are unofficial and slightly inaccurate benchmarks. It takes anywhere between 5.5 - 9.75 minutes to transfer 63GB based on various softwares and configurations [6]. This was tested on a Samsung SSDs; therefore, the numbers look high and show immense performance. The built-in windows explorer copied the files in 6.75 minutes, which is faster than all of the softwares mentioned above. Therefore, it may seem that the Operating System utilizes the resources very efficiently, making other softwares obsolete.

While this project has seen somewhat improvement under test conditions, calling it a success would be an overstatement. Had it succeeded in achieving its goal, there were plans for creating a modified version to be deployed over servers. Servers usually cross back up across data centers when the traffic is low. There were several drawbacks the project encountered. Noticeably most of the mistakes were made during the initial phases of the project. This includes decision-making and design choices. Furthermore, Tauri is not mature enough and has a number of ongoing issues. Nonetheless, while some may call this project a failure, it has been a great learning experience and advanced my understanding of the Rust programming language. Not every project is meant to be successful, and failure is just a step closer to success.

References

[1] Subathra, S., Sethuraman, M., & Babu, J. V. (2005, December). Performance analysis of dictionary based data compression algorithms for high speed networks. In 2005 Annual IEEE India Conference-Indicon (pp. 361-365). IEEE.

[2] Edel, N. K., Tuteja, D., Miller, E. L., & Brandt, S. A. (2004, October). MRAMFS: A compressing file system for non-volatile RAM. In The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. (pp. 596-603). IEEE.

[3] More, S., Choudhary, A., Foster, I., & Xu, M. Q. (1997, April). MTIO. A multi-threaded parallel I/O system. In Proceedings 11th International Parallel Processing Symposium (pp. 368-373). IEEE.

[4] Feki, R., & Gabriel, E. (2022, March). Design and evaluation of multi-threaded optimizations for individual MPI I/O operations. In 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP) (pp. 122-126). IEEE.

[5] Al-Laham, M., & El Emary, I. M. (2007). Comparative study between various algorithms of data compression techniques. IJCSNS, 7(4), 281.

[6] YouTube. (2021). FastCopy, SuperCopier, TeraCopy, UltraCopier and Copy Handler Speed Comparison (2021). YouTube. Retrieved April 29, 2023, from https://www.youtube.com/watch?v=35UxFaVsG-g.

[7] Lott, C. (2021, November 18). C is the greenest programming language. Hackaday. Retrieved April 29, 2023, from https://hackaday.com/2021/11/18/c-is-the-greenest-programming-language/