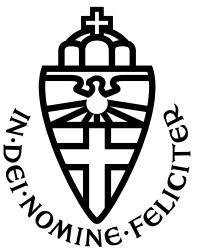
Bachelor's Thesis
Computing Science



Radboud University Nijmegen

Bifrost

Scheduler-Driven Multithreaded File Copying

Author:
Jakub Sykulski
s1071754

Supervisor:
Prof. Arjen P. de Vries

Second reader:
Prof. Sven-Bodo Scholz

March 01, 2026

Abstract

Sequential file copying tools such as `cp` and `rsync` do not take a full advantage of modern hardware. They perform transfers one step at a time, leaving significant performance on the table even on high throughput SSDs. This thesis investigates how to develop a tool for copying files considering current hardware properties, using the Bifrost proof-of-concept tool to answer this question. Bifrost is a multithreaded file copying tool built on the Tokio asynchronous runtime that intelligently groups and distributes file transfers across worker threads before copying begins. Benchmarking under cache-controlled conditions demonstrates that this approach gives a notable speed up over existing tools. This suggests that the bottleneck in sequential tools lies not only in CPU or I/O capabilities, but in how work is ordered and scheduled.

Contents

1 Introduction **5**

2 Preliminary knowledge **7**
2.1 Copying files in the Linux kernel . . . 7
2.2 Context switching . . . 7
2.3 Operating System (OS) threads versus lightweight threads . . . 8

3 Literature Review **8**

4 Methods **10**
4.1 Core assumption for Bifrost operations . . . 11
4.2 Directory analysis . . . 11
4.3 Directory reconstruction . . . 12
4.4 Schedulers — the inner workings and resulting workloads . . . 13
4.4.1 Completely Fair Scheduler (CFS) . . . 13
4.4.1.1 High-Level Description . . . 13
4.4.1.2 Problem solved by the scheduler . . . 13
4.4.1.3 Inner working of the algorithm . . . 13
4.4.1.4 Time and Space complexity . . . 14
4.4.2 Ordering Scheduler (OS) . . . 14
4.4.2.1 High-Level Description . . . 14
4.4.2.2 Problem solved by the scheduler . . . 14
4.4.2.3 Inner working of the algorithm . . . 15
4.4.2.4 Time and Space complexity . . . 15
4.4.3 Temporal Scheduler (TS) . . . 16
4.4.3.1 High-Level Description . . . 16
4.4.3.2 Problem solved by the scheduler . . . 16
4.4.3.3 Inner working of the algorithm . . . 16
4.4.3.4 Time and Space complexity . . . 17

5 Results **18**
5.1 Benchmarking . . . 18
5.1.1 Benchmarking methodology . . . 18
5.1.2 Hardware and test environments . . . 18
5.1.3 Dataset . . . 19
5.1.4 Procedure . . . 19
5.2 Limitations . . . 25
5.2.1 Minerva — sustained load behavior on a real world library . . . 25
5.2.2 ISAC — the possible role of the storage controller . . . 30

6 Discussion **33**
6.1 The hardware ceiling and storage controllers (Sarunya Pumma and Balaji 2019) . . . 33
6.2 Limitations: The straggler problem (Hongzi Mao and Alizadeh 2019) . . . 33
6.3 Future improvements: File chunking . . . 33
6.4 Evolving the Temporal Scheduler (Engin Ipek and Schulz 2006; Ibrahim Umit Akgun and Zadok 2021; Prabhakar and Mishra 2025) . . . 33

7 Conclusions **34**

Bibliography **35**

Chapter 1

Introduction

Modern systems increasingly require efficient, high-throughput mechanisms for moving data, whether for incremental backups, large-scale deployment pipelines, or bulk data processing workflows. The two most widely used tools for file copying on Unix based systems, `cp` and `rsync`, were designed at a time when single-core CPU performance was the main bottleneck. Their largely sequential execution model reflects that era. Files are read and written one at a time, with little usage of the parallelism available in modern hardware. So, on modern systems equipped with multicore processors and high-bandwidth solid-state storage, these tools leave performance on the table.

Bifrost is a multithreaded file copying utility built on top of the Tokio asynchronous runtime, designed to move to concurrent algorithms. Rather than relying on a single thread of execution, Bifrost distributes copying work across multiple asynchronous tasks and worker threads. This allows I/O operations to overlap and proceed concurrently. However, Bifrost does not treat parallelism as a matter of simply spawning as many threads as possible. Instead, it introduces a dedicated scheduler layer that sits between the user-facing interface and the underlying I/O engine. This scheduler is responsible for analyzing the workload by examining sizes, counts, and distribution of files to be copied. This allows for making decisions about how those files should be grouped and assigned to worker threads before copying begins. The aim is to avoid the imbalanced work queues and contention that naive parallelism tends to produce, especially when file sizes vary widely across a dataset.

Different scheduling methods have been proposed to optimize throughput across workloads. To evaluate real world impact on file transfers, this thesis analyzes three strategies implemented in Bifrost: Completely Fair Scheduler (CFS), Ordering Scheduler (OS) and Temporal Scheduler (TS). These algorithms are selected to cover different directory structural profiles. CFS handles standard directories with unordered files of varied sizes. For extremely large directories, TS builds a linear regression model to predict transfer runtimes and schedules tasks using the Longest Processing Time (LPT) algorithm. Finally, for complex or uniform file distributions where conventional algorithms experience performance degradation, Ordering Scheduler applies unsupervised k-means clustering operating on a logarithmic scale of file sizes to group tasks by order of magnitude.

Beyond the scheduler, the choice of an asynchronous runtime as the execution engine is driven by architectural advantage. Traditional approaches to concurrency rely on spawning one operating system thread per unit of work. Operating system's threads carry substantial overhead. Each thread is allocated a fixed size stack, typically several megabytes. Switching between them requires a full context switch managed by the kernel. This involved saving and restoring register state, flushing CPU caches and transitioning between user and kernel space. When hundreds, thousands or millions of files must be copied concurrently this overhead accumulates quickly and begins to bottleneck throughput. Tokio's asynchronous model addresses this directly. Rather than mapping each concurrent task to its own operating system thread, the runtime multiplexes many lightweight asynchronous tasks onto a pool of operating system's threads. These tasks maintain their own state on the heap with growable, on-demand allocated stacks. This allows them to consume only the memory they actually need unlike fixed size operating system's threads. The main advantage is that when a task is waiting on I/O, it does

not block its underlying thread. Instead, it yields execution back to the runtime, without operating system's intervention, allowing another ready task to run immediately on the same thread. This yielding is cheap, it amounts to a function return rather than a kernel context switch. The effect is that Bifrost can maintain many in-flight copy operations simultaneously, with minimal memory pressure and without the most of the context switching costs.

The central research question this thesis seeks to answer is: Does a scheduler-driven file copying architecture, running on top of an asynchronous runtime, allow for faster file transfer compared to both naive parallel execution and established sequential tools?

Chapter 2

Preliminary knowledge

Before discussing Bifrost, it is useful to show how and what building blocks it relies on. Main things worth considering are how the Linux kernel actually moves bytes from one file to another, what a context switch costs and what distinguishes an operating system thread from the lightweight, runtime managed tasks that Tokio schedules.

2.1 Copying files in the Linux kernel

A naive file copy in user space is a loop of `read()` and `write()` calls. A buffer is filled from the source file descriptor and drained into the destination file descriptor, with the kernel involved in both directions. Under the hood, neither call touches the storage device directly. All regular reads, writes, and memory mapped accesses to a file are routed through the page cache. On a read, the kernel first checks whether the requested page already exists. If that is the case the data is copied straight out of RAM and no device I/O occurs at all. If not, the kernel schedules a read, puts the read bytes in the cache and only then satisfies the request. Writes are done very similarly. The actual transfer to disk is up to the kernel. It can flush cached pages later, either because too much data has accumulated or because the pages have aged past some threshold or because the application explicitly requested a write with a command like `fsync()`.

This design is what makes the naive read/write loop expensive for a pure file-to-file copy. Every chunk of data crosses the user/kernel rings twice. Linux provides copy primitives specifically to avoid this. `sendfile()` or `send_file_range()` copies data between two file descriptors entirely inside the kernel, which is more efficient than the combination of `read()` and `write()`. This method removes the need to transfer the data into user space and back again. This is the primary mechanism behind tools such as `cp` and `rsync`.

2.2 Context switching

Whenever the kernel suspends one thread of execution and resumes another, it performs a context switch. The registers, program counter, and stack pointer of the current thread are saved, those of the incoming thread are restored. If the switch crosses process boundaries, the page tables and address space mapping are changed as well. This direct cost is small, but it is not the main expense. The more significant cost comes from the disruption to the CPU's caches and workflow itself. The current thread's data is removed as the incoming thread's instructions and data are loaded. Once the original ("current") thread resumes it must effectively "restart". For a file-copying tool that wants to keep hundreds or thousands of transfers in flight, this really matters. Naively mapping every concurrent transfer onto its own thread may cause the operating systems scheduler to switch between them constantly, and with each switch the overhead accumulates.

2.3 Operating System (OS) threads versus lightweight threads

The traditional unit of concurrency that the kernel manages is the OS thread. Creating one is relatively expensive. The kernel reserves a large, mostly unused stack for it and registers it with the scheduler. From that moment that thread costs a full context switch, including the cache and workflow disruption described above. This overhead is tolerable when the number of threads is relatively small. The overhead becomes a bottleneck when an application wants to spawn hundreds or thousands of cheap, mostly idle threads. Which is exactly what might happen with a directory with many small files.

The alternative is to manage concurrency at user level instead of making the kernel do it. Pure user-level, also known as “green” threads or tasks in the Tokio framework (Behren et al. 2003; Shintaro Iwasaki and Shiina 2021), can be created and switched far more cheaply. This is because no kernel crossing is required.

Tokio, the runtime Bifrost is built on rather than allocating an OS thread per task, it multiplexes many lightweight, heap-allocated tasks onto a fixed pool of worker threads. Tokio distributes work between them with a work-stealing scheduler. Each worker primarily executes from its own local queue and only reaches into another worker’s queue when its own is empty. This approach keeps cross thread synchronization balanced. Because a task yields control easily, where it would otherwise block on I/O, switching between tasks on the same worker thread amounts to a more of a function return rather than a kernel context switch.

Chapter 3

Literature Review

Anvari and Kaeli (2026) study when dynamic task scheduling pays off in parallel systems, and when its overhead outweighs the benefit. They observed that every dynamic scheduler imposes a non-zero cost for queue management, work-stealing decisions, etc. and that this cost is only worthwhile when each task carries enough work to justify it. When tasks become too small, the scheduling overhead actually makes execution time longer. This directly relates to Bifrost's grouping files into workgroups before copying rather than spawning one task per file. By putting many files into a single unit of work, the schedulers described in this thesis raise the effective task granularity to a point where parallel execution can be considered beneficial.

Bramas and Ketterlin (2020) showed that in task based systems, the cost of managing dependencies between tasks starts to matter when the tasks themselves are cheap to run. Their solution is to merge small tasks into fewer, larger ones. This cuts down on that management overhead and shortens the total runtime. Using this kind of clustering, they were able to achieve significant speed up. Bifrost applies the same idea to file copying. Instead of treating every file as its own separate job, Completely Fair Scheduler (CFS), Ordering Scheduler (OS) or Temporal Scheduler (TS) each bundle files together into workgroups, paying a scheduling cost up front in exchange for much less per-task overhead while copying. The main difference is what drives the grouping. Their method groups tasks based on how they depend on one another, where Bifrost groups files by size and predicted copy time.

Blumofe and Leiserson (1999) formalized the work-stealing strategy for scheduling multithreaded computations. Each worker thread executes tasks from its own local queue and only when it runs out of work it steals a task from another worker. They proved that this strategy achieves efficient execution time and memory use. This strategy keeps synchronization costs proportional to the actual load imbalance rather than to the total number of tasks. This logic is behind both runtimes Bifrost is built on: Rayon's thread pool, which drives the parallel directory analysis and scheduler logic, and Tokio which drives the copying phase. Both are work-stealing schedulers as described earlier. The difference in Bifrost's case is that work stealing balances load dynamically *within* a given phase, while the schedulers described in this thesis balance load statically *before* the copying phase begins.

Vazhkudai and Schopf (2003) address the problem of predicting how long a large data transfer will take, using a regression model built from cheaply observable features rather than requiring knowledge of the entire transfer in advance. Their results show that even a lightweight regression fit meaningfully outperforms naive estimators. This is the closest basis for Temporal Scheduler's core assumption. The assumption is that a small, quickly-fit model can stand in for expensive per-file measurement. The main difference lies in what the model is fit on. Their regression uses data from a live network and disk telemetry gathered during the transfer itself. TS regresses purely on file size, sampled once before scheduling begins. Bifrost has no live telemetry channel available at scheduling time as of now.

Behren et al. (2003) argue that user-level threads can provide the good scalability of asynchronous, event-driven state machines while retaining the programming model of traditional OS threads. They showed that by separating thread management from the OS kernel and using dynamic, on-demand stack allocation, applications can efficiently support hundreds or even thousands of threads with

minimal memory overhead. Instead of spawning a memory heavy kernel thread for every file being copied, Bifrost multiplexes thousands of lightweight tasks onto a pool of workers. Just like the scalable threads described in their research, Tokio's tasks maintain their state on the heap and yield control entirely in user space. The main difference is the application domain. Their work focuses on managing concurrent internet server connections, Bifrost applies this lightweight concurrency model to parallelize I/O storage operations without overwhelming the kernel's context-switching capabilities.

This section outlines the main research that supports Bifrost's core execution model. The selected literature explores the trade-offs in task granularity, the work-stealing schedulers, and the benefits of lightweight threading. Further literature, which directly supports the benchmarking results and future improvements, is addressed within the respective chapters later in this thesis.

Chapter 4

Methods

4.1 Core assumption for Bifrost operations

The design of Bifrost assumes that file systems remain active and unlocked during copying operations. So, Bifrost limits its execution scope strictly to the dataset identified during its initial analysis phase ("directory analysis"). Any files added after this snapshot is taken are disregarded for the duration of the transfer. If any modifications occur, such as file deletions or renaming after analysis phase is finished Bifrost will handle these gracefully by logging an error and continuing the pipeline. This snapshot based execution model ensures deterministic scheduling without imposing explicit file locks on the underlying file system.

4.2 Directory analysis

Before any copying can take place, Bifrost must first discover the complete set of files contained within the source directory. This stage, referred to as directory analysis, traverses the source tree, gathers metadata for every file, and produces a vector of records `Vec<FileData>` (see Listing 1) that is handed to the scheduler. Bifrost supports two modes of directory analysis: sequential and parallel, both of which are built on top of the same Rayon thread pool. The only difference between them is the number of worker threads the pool is configured with. In sequential mode the pool is initialized with a single thread. In parallel mode it is initialized with k workers. The value of k is user-specifiable and defaults to the number of CPUs available on the host machine.

The traversal follows a recursive, divide-and-conquer strategy that descends the directory tree depth-first. Analysis begins at the root of the source directory. Its immediate entries are read, files are isolated and analyzed, and the resulting records are pushed into a `discovered: Vec<FileData>` buffer. For every subdirectory encountered at the root, a separate walker is spawned through the function `gather_directory_files()`, which returns a `Result<Vec<FileData>>`.

Within `gather_directory_files()`, the contents of a directory are read into a vector and split into two halves: a left and a right chunk. The two halves are then processed through `rayon::join()`. The first closure (Operation A) is executed immediately on the thread that invoked the join, while the second closure (Operation B) is made available for parallel execution. If an idle worker exists in the pool, it steals Operation B and runs it concurrently. Otherwise, once Operation A completes, the originating thread executes Operation B itself. This work-stealing behavior means the traversal never blocks waiting for a free thread. In the worst case, when no workers are available, execution simply reduces to a sequential pass on the current thread. Thanks to this mechanism, Rayon's scheduler guarantees that every spawned task is eventually executed, and the design is inherently free of starvation and deadlocks.

The actual per-chunk work is done by `analyze_dir()`, which returns a `Vec<FileData>`. It collects every file in the chunk it is given, and whenever it encounters a nested directory it recurses by calling `gather_directory_files()` again and expands the directory tree one level deeper. This mutual recursion between `gather_directory_files()` and `analyze_dir()` is what drives the depth-first expansion of the source tree.

Once every directory has been expanded and all files analyzed, the results of the initial walkers are validated. Each walker returns a `Result`, and the analysis succeeds only if all of them returned `Ok`. If any walker fails, the entire analysis is invalidated and an error is returned. A partial file listing could lead to an incomplete or corrupted copy. When all checks pass, the vectors returned by the walkers extend the `discovered` buffer, initialized at the beginning, returning the complete set of files to be scheduled.

Initial manual testing indicates that, for sufficiently complex directory structures, the parallel analysis achieves a speed-up of roughly two to three times faster over the sequential variant.

```
pub struct FileData {
    src_path: Option<PathBuf>,
    relative_path: Option<PathBuf>,
    name: Option<String>,
    size: Option<u64>, // in bytes
    last_modified: Option<DateTime<Utc>>,
}
```

Listing 1: `FileData` struct definition

4.3 Directory reconstruction

A significant source of overhead in tools such as `cp` and `rsync` is that they create destination directories on the fly. A parent directory is always created before any of its children, so a child is never written to a destination that does not yet exist. The cost of this guarantee, is that the copying process is repeatedly interrupted to perform metadata operations, and the ordering constraint between parents and children unfortunately serializes most of the work.

Bifrost avoids this overhead by separating the two concerns. The directory structure of the source is extracted in full during the analysis phase and recreated in the destination before any file is copied. Once the destination tree mirrors the source, the copying phase reduces to writing file contents into directories that are guaranteed to exist. This pre-creation of the directory tree gives three properties that are valuable for parallel copying.

First, because each file's destination is fully determined by the pre-built tree, and filenames within a single directory need to be unique (operating system constraint), no two writes can ever target the same path. The copier therefore never needs to perform conflict or existence checks at write time. Second, since the entire structure already exists, there is no possibility of a race over whether a parent or its child is created first. Third, with the structure in place, workers may write into disjoint regions of the tree simultaneously without coordination, as their operations touch separate paths and cannot interfere with one another.

The disjoint property is what makes scheduling possible in the first place. This is because a file's destination no longer depends on when, or by which worker, its parent directory is created. Files may be grouped without taking their position in the source tree into consideration. Two files located at entirely different depths, or in unrelated branches, can therefore be assigned to the same workgroup. This separates the layout of the source tree from the way work is distributed. This leaves the scheduler free to group files purely by the criteria it cares about.

Extracting the directory structure ahead of time also allows for additional analysis techniques that are difficult to apply when directories are created lazily. One such technique implemented in Bifrost is non-empty extraction, which uses reference counting to determine whether a directory genuinely contains data. Each directory is assigned two counters derived from the number of files it contains and the number of subdirectories it holds. Directory is considered empty only when both counters are 0.

This allows Bifrost to reason about which directories carry content and to handle empty directories during reconstruction in the destination.

4.4 Schedulers — the inner workings and resulting workloads

4.4.1 Completely Fair Scheduler (CFS)

4.4.1.1 High-Level Description

Completely Fair Scheduler is the default scheduler for the program. It distributes files across threads by grouping them into workgroups, then assigning one workgroup per thread. Groups are formed by finding subsets of files whose combined size approaches a target value k, defined as the quadratic mean (RMS) across all file inputs. This ensures each workgroup carries roughly equal total load. Files that cannot be cleanly grouped, either because they are too large or have missing metadata, are treated as outliers and distributed evenly across the existing groups to maintain balance.

4.4.1.2 Problem solved by the scheduler

The scheduler is trying to solve the Bin Packing Problem. To be more precise, a greedy variant of it called First Fit (Dósa and Epstein 2018). This is equivalent to load balancing, where tasks are assigned to workers such that no worker is unnecessarily idle while others are overloaded.

4.4.1.3 Inner working of the algorithm

CFS operates in multiple stages to transform a flat list of input files into a structured set of workgroups.

The first stage is input partitioning. The raw input `Vec<FileData>` is divided into chunks of a user-defined size, with a default minimum of 64 files per chunk. The resulting chunks are processed in parallel using Rayon's `.par_iter()`, assigning one worker per chunk. The number of parallel workers is determined by the number of input chunks, allowing the scheduler to adapt to the data size without requiring manual thread count configuration.

The second stage is outlier isolation. Before bin packing, each chunk is analyzed to identify files that meet either of two criteria. Either size of the file exceeds the RMS of all input files, or file is missing metadata. These outliers are set aside and handled separately because their inclusion would distort the bin-packing process. Specifically, a file larger than the target capacity k cannot be efficiently grouped, resulting in single-file groups, which is undesirable.

The third stage involves greedy bin packing. The remaining files in each chunk are partitioned into subgroups using a first-fit greedy algorithm. Files are sequentially added to the current subgroup until one of two conditions is met. Either the combined size of the subgroup exceeds k, or the subgroup reaches its fixed size limit. When either condition is met, the current subgroup is closed, added to the thread-global accumulator. The target value k is the root-mean-square (RMS) of all input file sizes, calculated before scheduling begins. RMS is chosen over the arithmetic mean because it gives more weight to larger files, resulting in a higher target value that better accommodates datasets where a few large files dominate. Each thread returns a tuple of `(Vec<Vec<FileData>>, Vec<FileData>)`, representing subgroups and outliers.

The fourth stage is outlier redistribution. After all parallel workers complete, their subgroups are combined into a single list and sorted in descending order by subgroup length. Longer subgroups, which contain more and smaller files, have a combined size closer to k and can accommodate additional large files without significant imbalance. Outliers are then distributed by cycling through this sorted list and being assigning starting from the longest group. This approach serves as a lightweight approximation of the Longest Processing Time (LPT) heuristic, using the initial sort to guide distribution and the cyclic pass to balance the remaining load.

The final output of the scheduler is a `Vec<Vec<FileData>>`, a flat list of subgroups that are going to be passed to the copying module.

4.4.1.4 Time and Space complexity

Let n is the number of input files, m the number of isolated outliers, and g the number of resulting groups. Stages 1 to 3 run in parallel across p chunks. Their wall-clock time is smaller by the factor of p. The target calculation is a single parallel reduction performed with `.par_iter()`. Finally, outlier redistribution runs on one thread.

The target calculation, partitioning, outlier isolation and bin packing all perform a constant amount of work per file and are therefore linear in n. The asymptotic cost of the scheduler is caused by the redistribution stage, which sorts the g groups by length and the m outliers by size. Because every file is either packed into a group or set aside as an outlier, and each group holds at least one file. If almost every file forms a singleton group ($g \to n$, $m \to 0$), the group sort costs $O(g \log g) = O(n \log n)$. If almost every file is set aside as an outlier ($m \to n$, $g \to 0$) the outlier sort costs $O(m \log m) = O(n \log n)$. In practice both g and m should far smaller than n, so the redistribution sort should be inexpensive. Pushing outliers into groups is a linear operation.

Stage	Time Complexity	Space Complexity
(0) Target (RMS)	$O(n)$	$O(1)$
(1) Input partitioning	$O(n)$	$O(n \cdot \text{sizeof}(\text{FileData}))$
(2) Outlier isolation	$O(m)$	$O(m \cdot \text{sizeof}(\text{FileData}))$
(3) Greedy bin packing	$O(n)$	$O(n \cdot \text{sizeof}(\text{FileData}))$
(4) Outlier redistribution	$O(g \log g + m \log m)$	$O(g + m)$

Table 1: Per-stage time and space complexity of the Completely Fair Scheduler.

4.4.2 Ordering Scheduler (OS)

4.4.2.1 High-Level Description

The Ordering Scheduler (OS) is a user opt-in scheduler that performs autonomous, unsupervised grouping of files through k-means clustering. Unlike CFS, which groups files against an absolute size target, OS partitions the input into k clusters according to the *magnitude* of each file. Every file size is first converted into a logarithmic scale of base 2. This is done because two files separated by a couple of bytes effectively end up on the same point, while files separated by a large factor are pushed far apart. The number of clusters k is user-definable and defaults to the number of available CPU cores, yielding one cluster per core. This scheduler is particularly well suited to very large datasets and to inputs on which CFS degrades.

4.4.2.2 Problem solved by the scheduler

CFS uses the root-mean-square (RMS) of all file sizes as its bin-packing target. This works well on heterogeneous inputs, but breaks down when a directory is *simplified*, file sizes are roughly uniform. In that case the RMS converges to the size of a single file, so the greedy approach closes every subgroup after admitting just one file, producing a schedule made almost entirely of groups of size one. Such a schedule offers no meaningful load balancing. The Ordering Scheduler avoids this by clustering on *relative* size instead of packing against an absolute target. A uniform dataset simply reduces to k comparably sized clusters. More generally, OS reduces scheduling to an unsupervised clustering problem.

4.4.2.3 Inner working of the algorithm

OS transforms a list of input files into k clusters in four stages, most of which are parallelized with Rayon.

The first stage is the log-space transformation. Each input file is mapped to a data point whose coordinate is $\log_2(\text{size} + 1)$, where the $+1$ guards against the singularity of $\log 0$ for empty files and base two is chosen so that the scale is expressed in bits. Working in log-space changes the meaning of distance. Since $\log a - \log b = \log\left(\frac{a}{b}\right)$, the distance between two points measures the *ratio* of their sizes, how many times larger one file is than the other. This makes the clustering adjustable to a wide range of real world directories.

The second stage is K-Means++ initialization. The first centroid is chosen uniformly at random from the data. Each of the remaining $k - 1$ centroids is then selected with probability equal to the squared distance from the nearest already chosen centroid. This spreads the initial centroids across the input and gives a faster and more stable convergence than purely random seeding. Initialization requires at least k distinct data points and fails otherwise.

The third stage is the application of Lloyd's algorithm repeated for at most 350 iterations. Each iteration performs an assignment step followed by an update step. In the assignment step every point is bound to its nearest centroid. The work is split into chunks and dispatched across threads, with the centroids extracted into a shared read-only buffer beforehand to avoid locking. In the update step each centroid is recomputed as the arithmetic mean of its members logarithms, which is the logarithm of the geometric mean of the file sizes. Should a cluster become empty, it is reinitialized to the point lying farthest, by summed distance, from all current centroids. This prevents clusters from silently collapsing. The loop terminates early once every centroid shifts by no more than a fixed threshold $\delta = 0.1$ between iterations.

The fourth stage is cluster extraction. Empty clusters are discarded and the surviving clusters are unwrapped from their internal `struct DataPoint` representation back into plain `Vec<FileData>` groups, producing the final `Vec<Vec<FileData>>` schedule that is to be handed to the copying module. Because empty clusters are dropped, the schedule may contain fewer than k groups. At present, very large files are likely grouped together into a single cluster, which can introduce minor load imbalance, addressing this is left as future work.

4.4.2.4 Time and Space complexity

Let n be the number of input files, k the requested number of clusters, I the number of assignment iterations (bounded by 350), and P the number of cores.

The cost is dominated by the iterative refinement, giving an overall sequential bound of $O(I \cdot n \cdot k)$. Because the dominant assignment step is parallelized, the wall-clock cost of each iteration falls by a factor of roughly P. The parameter k also acts as a tuning parameter. Larger k produces finer-grained clusters and more downstream parallelism, at the cost of an initialization phase that grows quadratically in k.

Stage	Time Complexity	Space Complexity
(0) Log-space transform	$O(n)$	$O(n \cdot \text{sizeof}(\text{DataPoint}))$
(1) K-Means++ init	$O(n \cdot k^2)$	$O(n + k)$
(2) Lloyd's iteration	$O(I \cdot n \cdot k)$	$O(n)$
(3) Cluster extraction	$O(n)$	$O(n \cdot \text{sizeof}(\text{FileData}))$

Table 2: Time and space complexity of the Ordering Scheduler per stage. The reported bounds are sequential; the transformation, initialization, and assignment steps are parallelized across P cores.

4.4.3 Temporal Scheduler (TS)

4.4.3.1 High-Level Description

The Temporal Scheduler (TS) is the most elaborate of Bifrost's scheduling strategies and is recommended for very large source directories. Rather than grouping files by their absolute or relative size, as CFS and OS do, TS models the actual time each file is expected to take to copy. It uses that estimate to build workgroups whose predicted makespans (completion times) are as close as possible to one another. It operates in two phases. Firstly, an estimation phase, in which a sample of the source directory is used to fit a linear model relating file size to expected copy duration. Then comes the scheduling phase, in which files are greedily assigned to whichever group is currently projected to finish earliest, a strategy modeled on the Longest Processing Time (LPT) algorithm.

4.4.3.2 Problem solved by the scheduler

Both CFS and OS implicitly assume that a file's contribution to a workgroup's load is well approximated by its size alone. In practice, however, the wall-clock cost of copying a file is a function of the underlying storage device's read and write throughput and the file size. These throughputs are rarely symmetric. A scheduler that balances purely on byte counts can therefore still produce workgroups with very different real completion times. TS addresses this by predicting a per-file operating time, expressed in seconds, and by balancing groups directly against that predicted time. Distribute n independent jobs of predicted duration across k workers so that the completion time, the operating time per group (machine/core), is minimized.

4.4.3.3 Inner working of the algorithm

TS transforms a list of input files into k scheduled groups in five stages.

The first stage is reservoir sampling. Rather than inspecting every file up front, TS chooses a fixed size, uniformly random sample of files from the scanned directory using reservoir sampling. This allows a representative subset of file sizes to be obtained in a single pass over the input.

The second stage is cost-model construction. Each sampled file is mapped onto a two-dimensional point (x, y), where x is the file's size in bytes and y is its predicted operating time, that is, the combined time to read and then write the file. The operating time is computed as

$$\text{time}(x) = \frac{x}{10^6} \cdot \left(\frac{1}{r} + \frac{1}{w}\right),$$

where r and w are the user-provided disk read and write speeds, expressed in MB/s. These sampled points are then fed into a linear regression module, which finds the line of best fit relating file size to operating time. The resulting model is used to predict the operating time cheaply of every remaining, non-sampled file directly from its size.

The third stage is outlier isolation. Before scheduling begins, files that are missing size metadata are removed from the main dataset and set aside as outliers.

The fourth stage is LPT style greedy scheduling. Bifrost first creates k groups, seeded by k randomly chosen files. Using the predicted operating times of these initial files a `BTreeSet` of `TimePoint` is created. `TimePoint` is a struct that holds an `operating_time` and a `group_id`. This data structure is ordered so that the group with the smallest cumulative operating time is always at the front. For every remaining file with a known size, the scheduler pops the minimum `TimePoint` from the set. `BTreeSet` guarantees, thanks to ordering, to identify the least loaded group and appends the file to that group's entry in a `HashMap<group_id, Vec<FileData>>`. A new `TimePoint` is then pushed back into the set, with its `operating_time` set to the popped value plus the current file's predicted operating time.

The fifth and final stage is outlier scheduling. Once every file of known size has been placed, the previously isolated outliers are allocated into the same groups using the same mechanism. Since an outlier's operating time is unknown, its effect on a group's is instead approximated. The popped `TimePoint`'s `operating_time` is updated to old $+ 2 \cdot$ old before being pushed back, treating the outlier as though it were at least as costly as the group's entire accumulated load. Once every outlier has been placed, the algorithm terminates and the resulting `HashMap` is unwrapped into the final `Vec<Vec<FileData>>` schedule handed to the copying module.

4.4.3.4 Time and Space complexity

Let n be the number of input files, s the fixed reservoir sample size, m the number of isolated outliers, and k the requested number of groups.

Reservoir sampling and outlier isolation are both single passes over the input and therefore run in $O(n)$. Fitting the linear regression touches only the sample and runs in $O(s)$. Evaluating the fitted model for a remaining file is $O(1)$, so predicting operating times for all non-sampled files costs $O(n)$ overall. The dominant cost is the LPT scheduling loop itself. Each of the $n - m$ known size files, and each of the m outliers, performs one pop and one push on a `BTreeSet` bounded by k elements, each of which costs $O(\log k)$, giving a total of $O(n \log k)$.

Stage	Time Complexity	Space Complexity
(0) Reservoir sampling	$O(n)$	$O(s \cdot \text{sizeof}(\text{FileData}))$
(1) Cost-model fitting	$O(s)$	$O(s)$
(2) Outlier isolation	$O(m)$	$O(m \cdot \text{sizeof}(\text{FileData}))$
(3) LPT scheduling	$O(n \log k)$	$O(k)$
(4) Outlier scheduling	$O(m \log k)$	$O(k)$

Table 3: Per-stage time and space complexity of the Temporal Scheduler.

Chapter 5

Results

5.1 Benchmarking

Note: Due to a mistake in a function that handles conversion of directory size from bytes to GB in the Python script the counts on top of the schemas are incorrect.

5.1.1 Benchmarking methodology

To assess whether Bifrost's scheduler-driven design translates into measurable performance gains, it was benchmarked against the standard GNU `cp` utility across two different hardware environments. Evaluating the tool on more than one machine serves two purposes. First, it separates behavior, which is directly tied to storage device, from behavior that generalizes across devices. Second, it exposes how the schedulers respond to different core counts, memory bandwidth, and I/O characteristics. This is precisely the point along which a scheduler-driven copier is expected to differ from a sequential one.

5.1.2 Hardware and test environments

The environments used are referred to here as CN79, OVH. CN79 is a node of the university SLURM cluster. Due to storage allocation limits for student experiments, the node's administrator provisioned a 200 GB RAM disk on which all copy operations were performed. Because the working set resides entirely in DDR3 memory, CN79 effectively measures the best case in which storage latency is removed and throughput is only bounded by memory bandwidth. OVH is a dedicated server rented from the OVH cloud provider, backed by a solid-state disk exposed through a JBOD. The full specifications of each environment are summarized in Table 4.

	CN79	**OVH**
Role	University cluster node	Rented dedicated server
CPU	Intel Xeon E5-2670 (Sandy Bridge)	AMD Ryzen 7 9700X
Cores / threads	16 / 16 (HT disabled)	8 / 16
Memory	200 DDR3	64 GB DDR5
Storage	200 GB RAM disk	400 GB SSD (JBOD)
OS	Ubuntu 22.04.5	Ubuntu 26.04 LTS
Kernel	6.8.0-124	7.0.0-27

Table 4: Hardware and operating-system configuration of the benchmarking environments.

On CN79 hyper-threading was disabled at the firmware level, so the node exposes 16 physical cores rather than the 32 logical cores that would otherwise be visible. In all three environments Bifrost was compiled with the stable Rust tool-chain, version 1.96.0. The version of Bifrost under test was 1.3, which is the final release of the program and the one on which every measurement in this thesis is based.

5.1.3 Dataset

The workload used for benchmarking is the iNaturalist 2019 competition dataset[1], a large collection of natural-image files with total size of approximately 73 GB. This dataset was chosen because it reflects a realistic file-copying scenario. A deep directory hierarchy containing a very large number of moderately sized files with a naturally skewed size distribution. On CN79 the dataset was copied in its native form. On OVH the available storage permitted a larger workload, so the mini-dataset (smaller subset of the iNaturalist 2019) was added to yield a working set of roughly 120 GB.

```
Source      -->   /home/jakub/test_data
Destination -->   /home/jakub/test_target
-----------
Analysis of source path run in the Multi-Threaded mode.
16 walkers used.
Analysis took: 0.30s
768,243 files discovered.
11,019 unique paths discovered.
```

Listing 2: Directory analysis on a representative source tree in multithreaded mode. This is the directory analysis of iNaturalist 2019 dataset.

5.1.4 Procedure

The entire benchmark was automated with a Python script that invoked each tool repeatedly and recorded its timings. This approach ensures that no manual timing or intervention influenced the results. On CN79 and OVH each tool was executed 50 times.

Cache handling differed between environments, and this distinction is central to interpreting the results. On CN79 the page cache was deliberately not dropped between runs. The dataset lives on a RAM disk, so its contents remain in memory regardless and a fixed pause of roughly two seconds between runs was sufficient for "cool down"[2] period in this case. CN79 therefore reports a purely memory-bound scenario. On OVH the page cache was flushed between every run via the kernel's `drop_caches` interface, forcing each copy to read cold from the SSD. The "cool down" period was roughly twenty seconds between runs. OVH therefore reports a cold cache, storage bound scenario, which is the more demanding and arguably more representative of real deployments.

The primary metric was wall-clock elapsed time, as this is what a user ultimately experiences.

Across all three environments Bifrost copied the dataset faster than `cp`, though the size of the advantage varied strongly with the storage device and the amount of data moved. Table 5 summarizes the main figures.

Environment	Data size	Runs	Bifrost (mean)	`cp` (mean)
CN79 (RAM disk, warm)	∼73 GB	≥ 50	∼5 s	∼55 s
OVH (SSD, cold cache)	∼120 GB	≥ 50	∼160 s	∼220 s

Table 5: Mean wall-clock copy times for Bifrost and `cp` across the benchmarking environments.

On CN79 the difference was the most visible. Bifrost completed the copy in approximately five seconds on average, where `cp` required around fifty-five seconds. This shows a speed-up of one order of magnitude. This is the clearest demonstration of Bifrost's ceiling. Once the storage device is removed as a bottleneck, the sequential tool is limited by its inability to keep more than one transfer at the time, while Bifrost saturates the available memory bandwidth by overlapping many transfers at once.

[1] https://github.com/visipedia/inat_comp/tree/master/2019

[2] "cool down" period is the amount of time in-between the transfers to limit thermal throttling

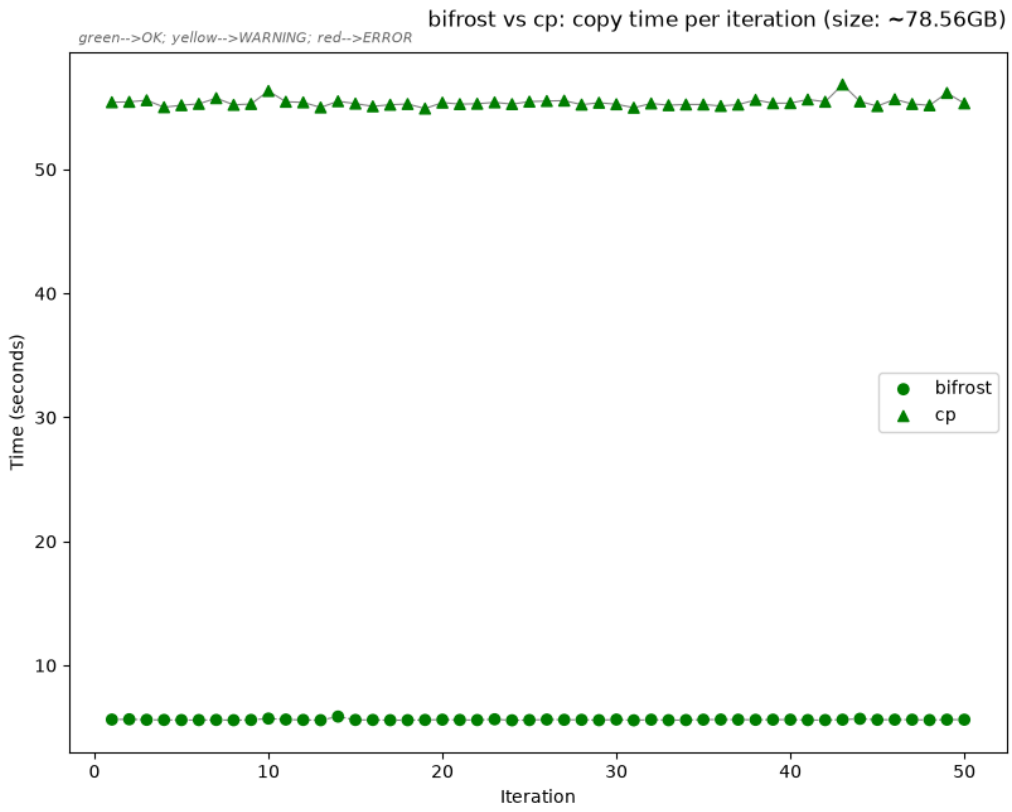


Figure 1: Distribution of copy times on CN79 (RAM, warm cache), when Bifrost CFS scheduler used compared against `cp`.

Figure 2: Distribution of copy times on CN79 (RAM, warm cache), when Bifrost OS scheduler used compared against `cp`.

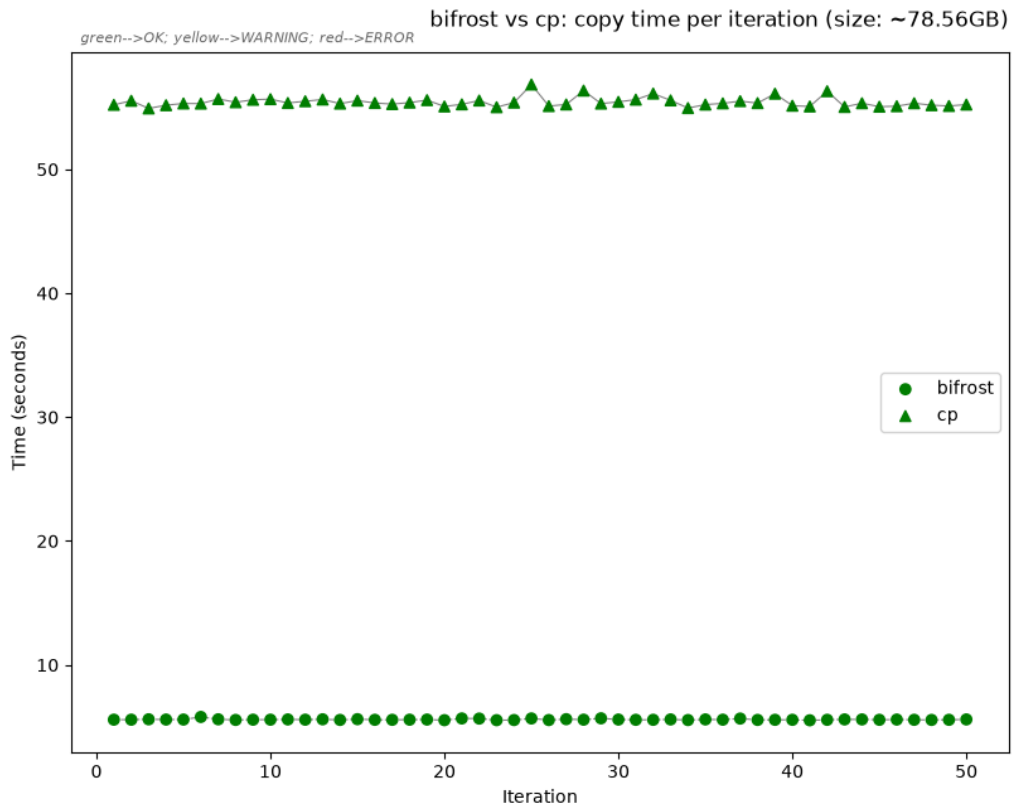


Figure 3: Distribution of copy times on CN79 (RAM, warm cache), when Bifrost TS scheduler used compared against `cp`.

On OVH the overall expected results were preserved, but the margin narrowed when a real storage device was reintroduced. Bifrost averaged around 160 seconds, occasionally rising to 180, against roughly 220 seconds for `cp`, which spiked as high as 280 seconds. The gain here, on the order of 1.4 times, is more modest than on CN79 but still substantial over a 120 GB transfer. The result is also highly sensitive to the scheduler in use. The Temporal Scheduler produced the most stable behavior, holding close to 160 seconds consistently across runs, which is consistent with its design goal of balancing groups by predicted completion time rather than by raw size.

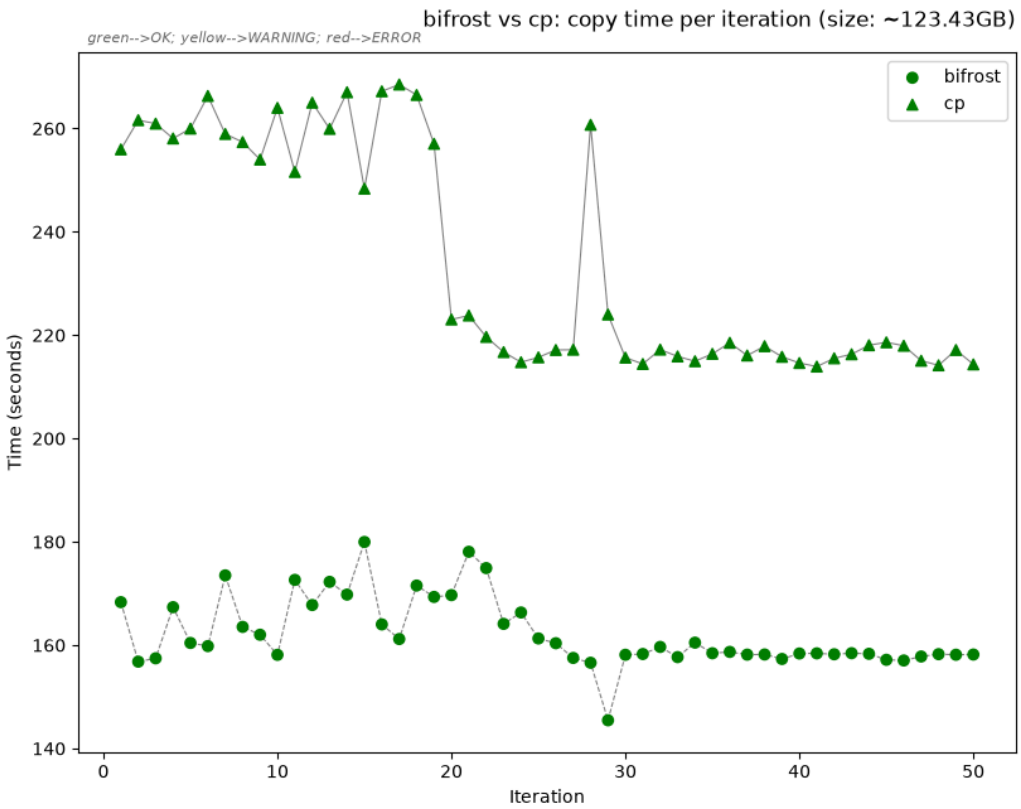


Figure 4: Distribution of copy times on OVH (SSD, cold cache), when Bifrost CFS scheduler used compared against `cp`.

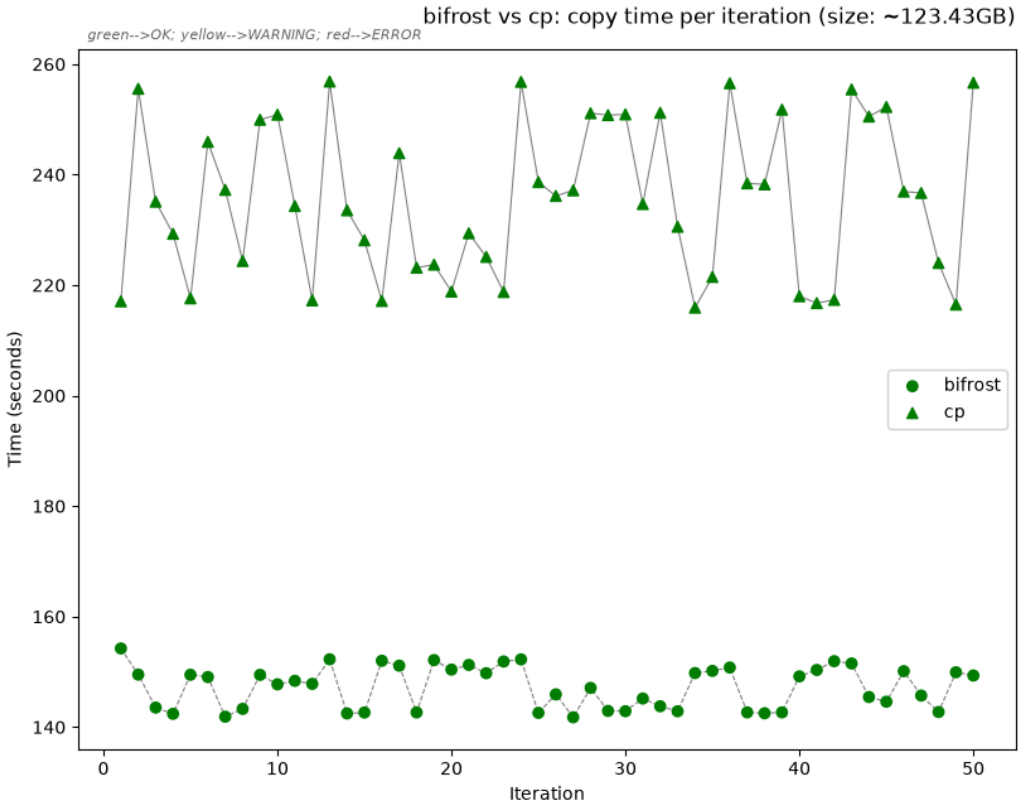


Figure 5: Distribution of copy times on OVH (SSD, cold cache), when Bifrost OS scheduler used compared against `cp`.

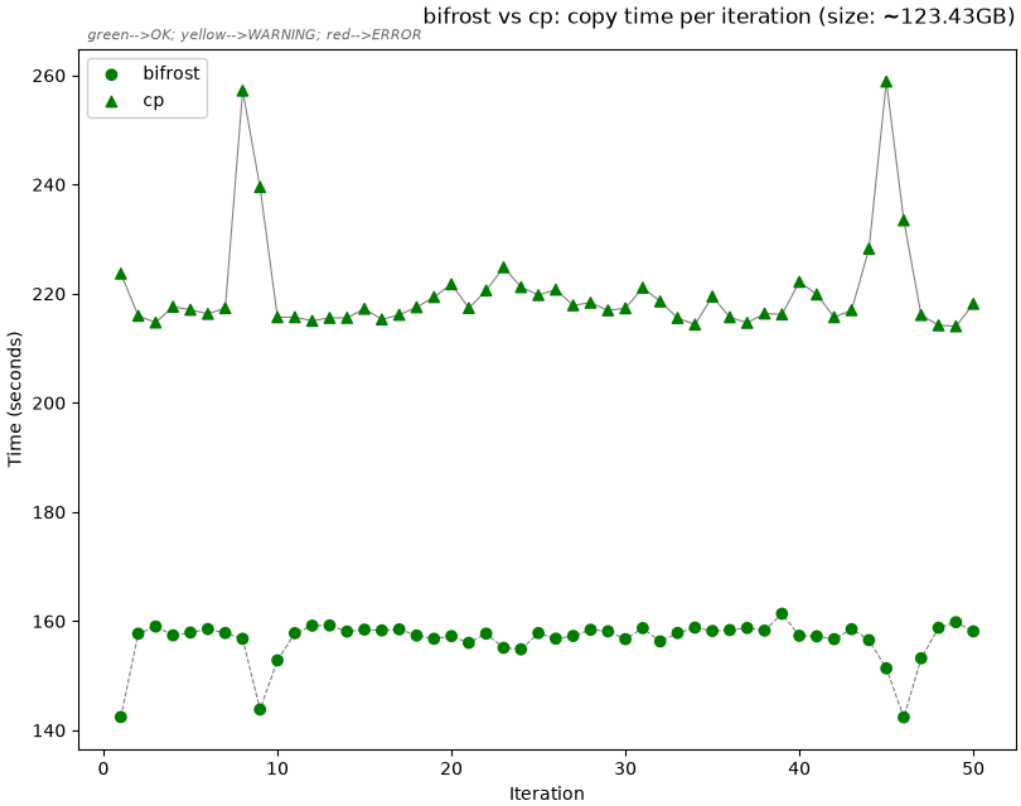


Figure 6: Distribution of copy times on OVH (SSD, cold cache), when Bifrost TS scheduler used compared against `cp`.

5.2 Limitations

	Minerva	**ISAC**
Role	Personal server	Personal server
CPU	AMD Ryzen 9 7940HX	AMD Ryzen 7 7745HX
Cores / threads	16 / 32	8 / 16
Memory	64 GB DDR5	64 GB DDR5
Storage	3 x 2TB Gen 4 SSDs	2 x 1TB Gen 4 SSDs
OS	Ubuntu 26.04 LTS	Ubuntu 26.04 LTS
Kernel	7.0.0-27	7.0.0-27

Table 6: Hardware and operating-system configuration of the benchmarking environments.

5.2.1 Minerva — sustained load behavior on a real world library

In place of a curated dataset, the Minerva workload was a direct export of a personal photo library from a self-hosted Immich instance, totaling approximately 140 GB. Of Minerva's three Gen 4 SSDs, one held the source library, which was copied onto the other drives. Copy integrity was verified with `dua-cli`[3], which computed the total directory size in bytes at both source and destination. A copy was considered valid only when the two figures matched exactly.

[3]https://github.com/byron/dua-cli

On a single, cold-run Bifrost retained a clear advantage over `cp`, completing the copy in roughly half the time. When the benchmark was instead repeated ten times per scheduler, however, only the first run showed this gain. Every subsequent run converged onto the `cp` baseline. Bifrost's copy time flattened to within noise of the sequential tool. The most plausible explanation is that repeated write-heavy copying puts the SSD into thermal throttling, at which point the drive's internal write rate, not the scheduler, becomes the main constraint. Once the device itself is the bottleneck, the scheduling layer can not do anything and Bifrost and `cp` are held to the same physical ceiling. For this reason the main figures for Minerva are taken from cold-runs that isolate the scheduler's contribution before throttling starts.

Experiment One, where a single coping operations was done, using Bifrost's OS scheduler, against `cp`:

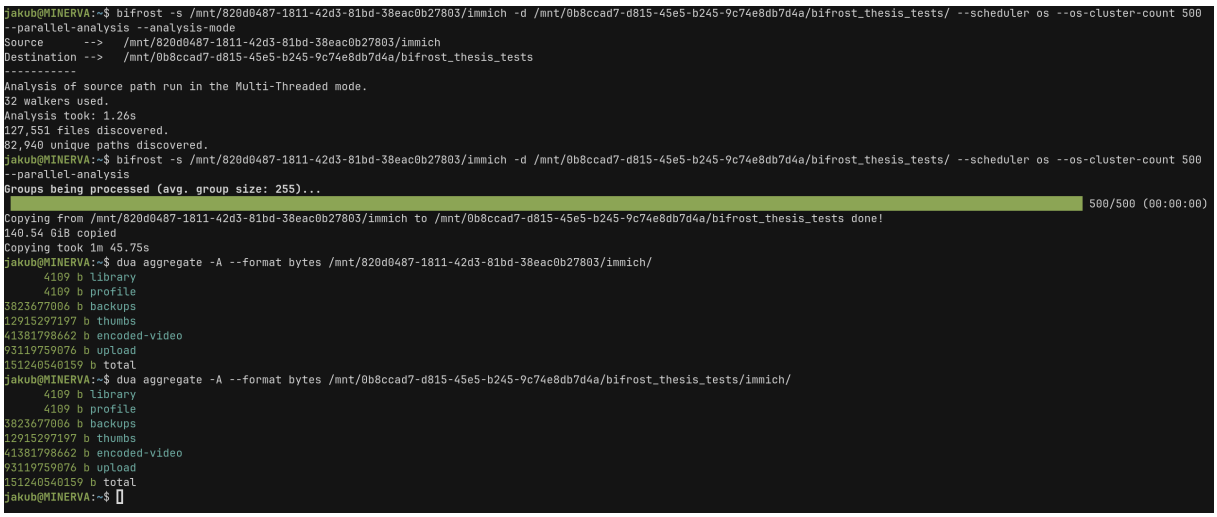


Figure 7: Copy time on Minerva with Bifrost's OS scheduler for the 140GB dataset.

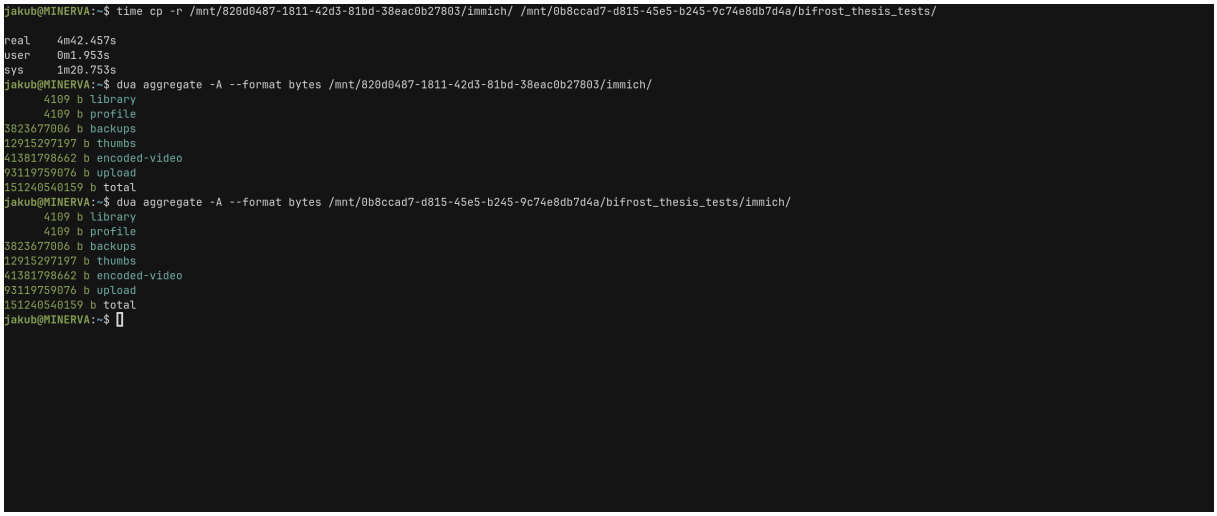


Figure 8: Copy time on Minerva with `cp` for the 140GB dataset.

Experiment Two, automated testing with a Python script:

Note: This is an actively working directory for the Immich server. The warnings appear due to detected directory size mismatch. This is because the directory grew in-between the runs and the source size, calculated once at the beginning of the script, was never updated in that version of the script. The tests were not re-run with the latest version of the script due to large amount of time needed to complete the run.

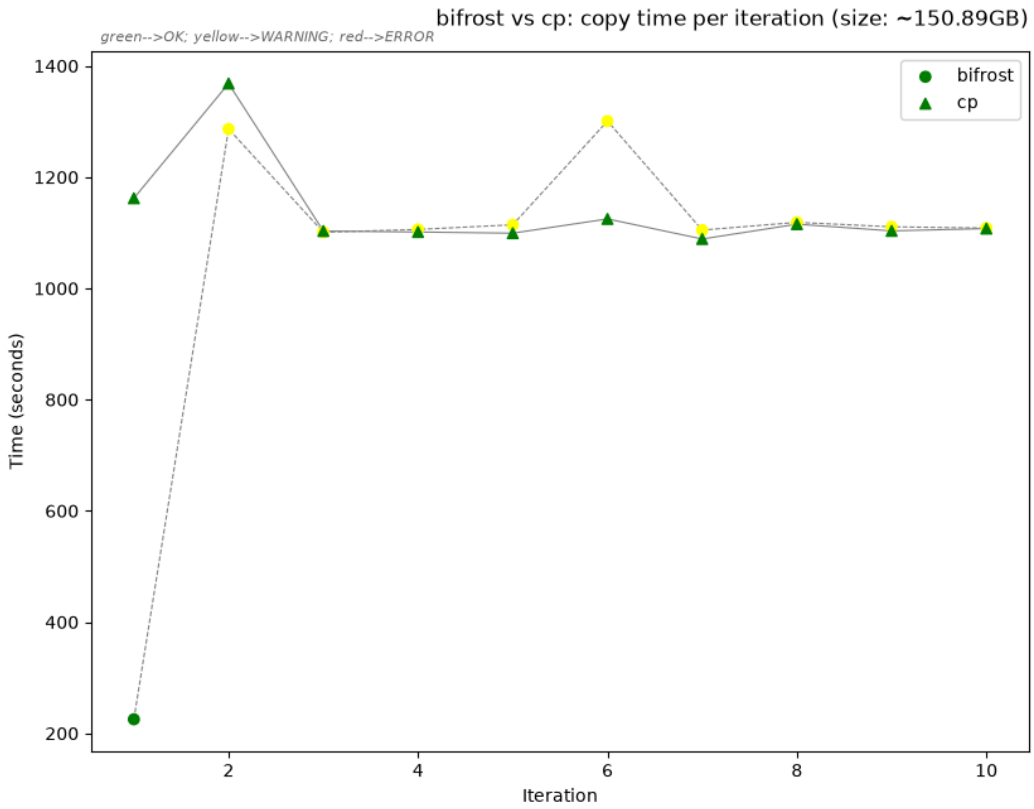


Figure 9: Copy time on Minerva with Bifrost CFS scheduler for the 140GB dataset compared against cp.

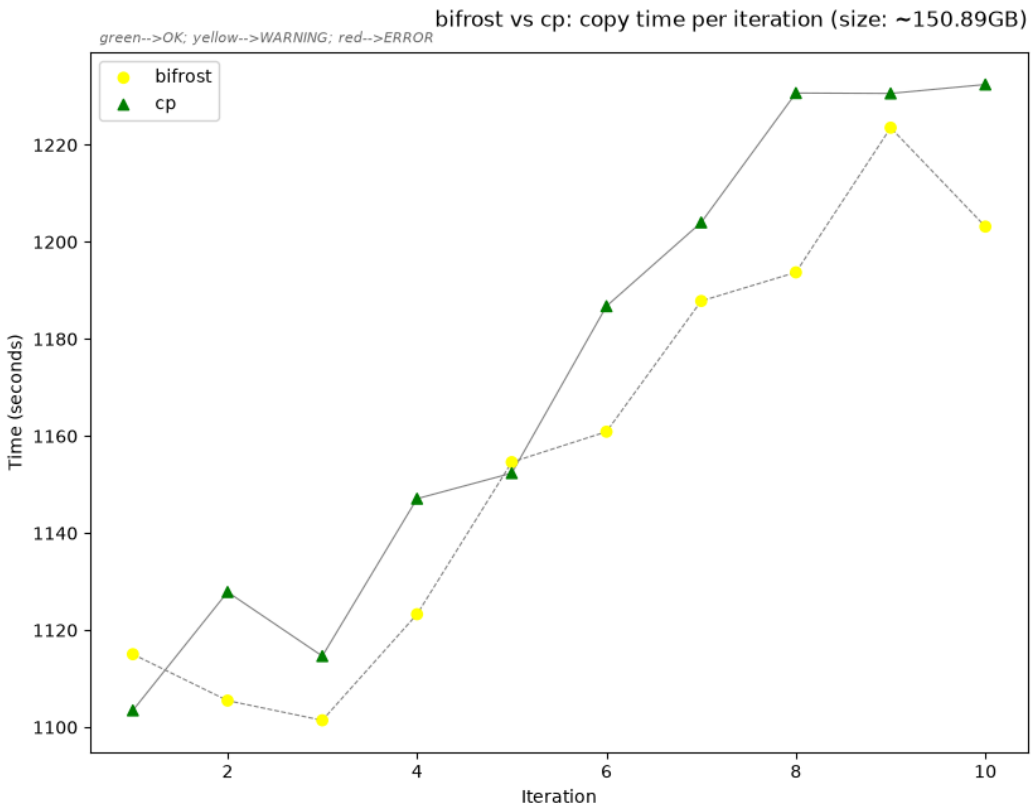


Figure 10: Copy time on Minerva with Bifrost OS scheduler for the 140GB dataset compared against cp.

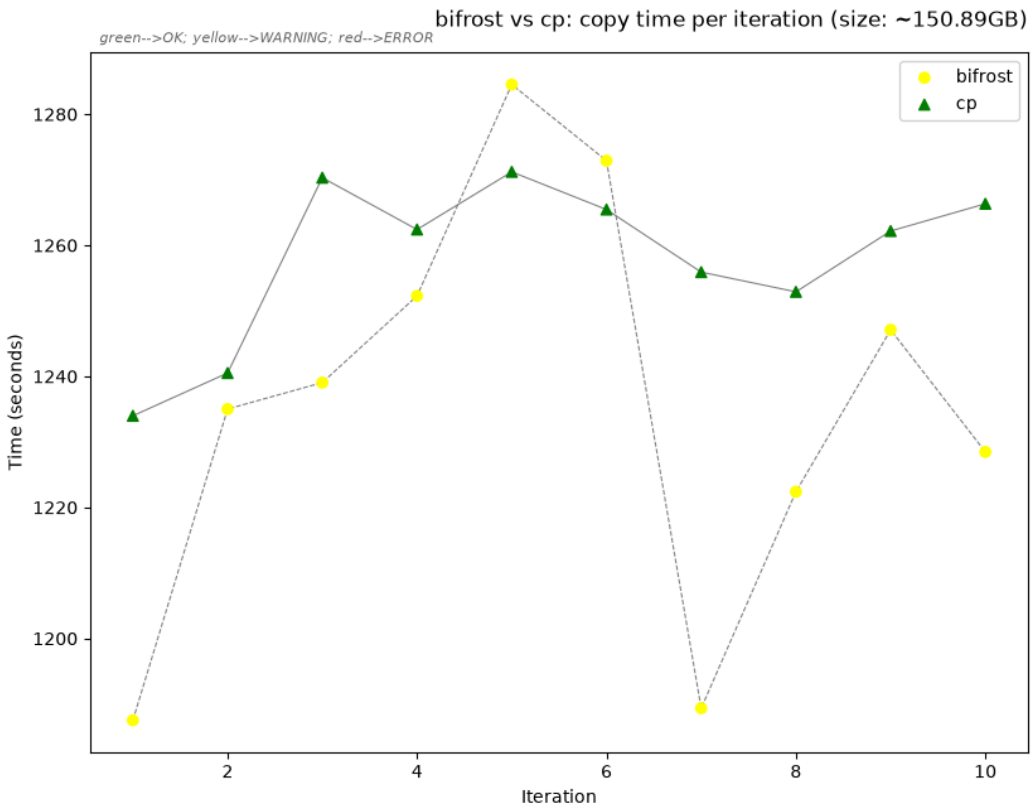


Figure 11: Copy time on Minerva with Bifrost TS scheduler for the 140GB dataset compared against cp.

5.2.2 ISAC — the possible role of the storage controller

ISAC is a separate personal server also carrying an Immich library of approximately 240GB. Running the 5 run protocol on a subset of this library which has a size of approximately 64GB, against Minerva, showed a second effect that, I believe, was independent of thermal behavior. Samsung's (Minerva SSDs) in-house controllers absorbed Bifrost's many concurrent write streams well and preserved a consistent speed-up, where on a drive (ISAC SSDs) using a more common third-party controller the advantage narrowed and under load was mainly lost. Because this difference appeared on a cold-run with throttling ruled out, it points to the controller as possible main factor. Essentially, how well the destination device handles concurrent writes controls how much of Bifrost's parallelism translates into wall-clock gain. Given the small number of drives used, this is reported as an observed effect worth further investigation rather than a general claim.

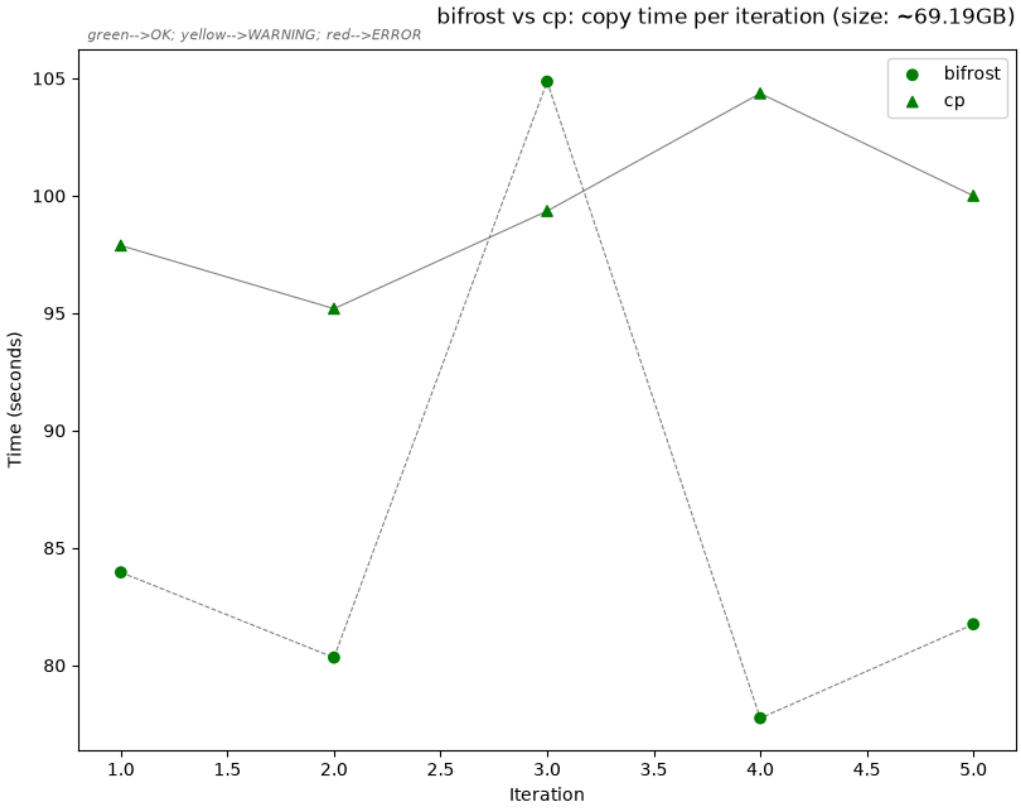


Figure 12: Copy time on ISAC with Bifrost CFS scheduler for the 64GB dataset compared against `cp`.

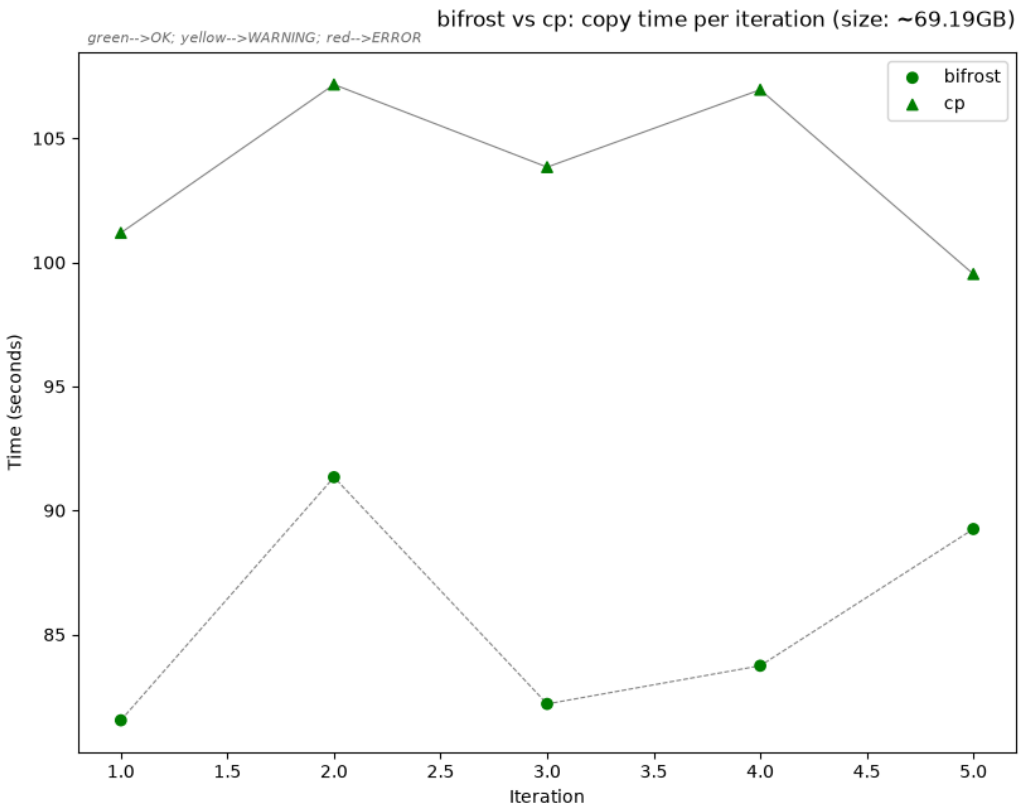


Figure 13: Copy time on ISAC with Bifrost OS scheduler for the 64GB dataset compared against `cp`.

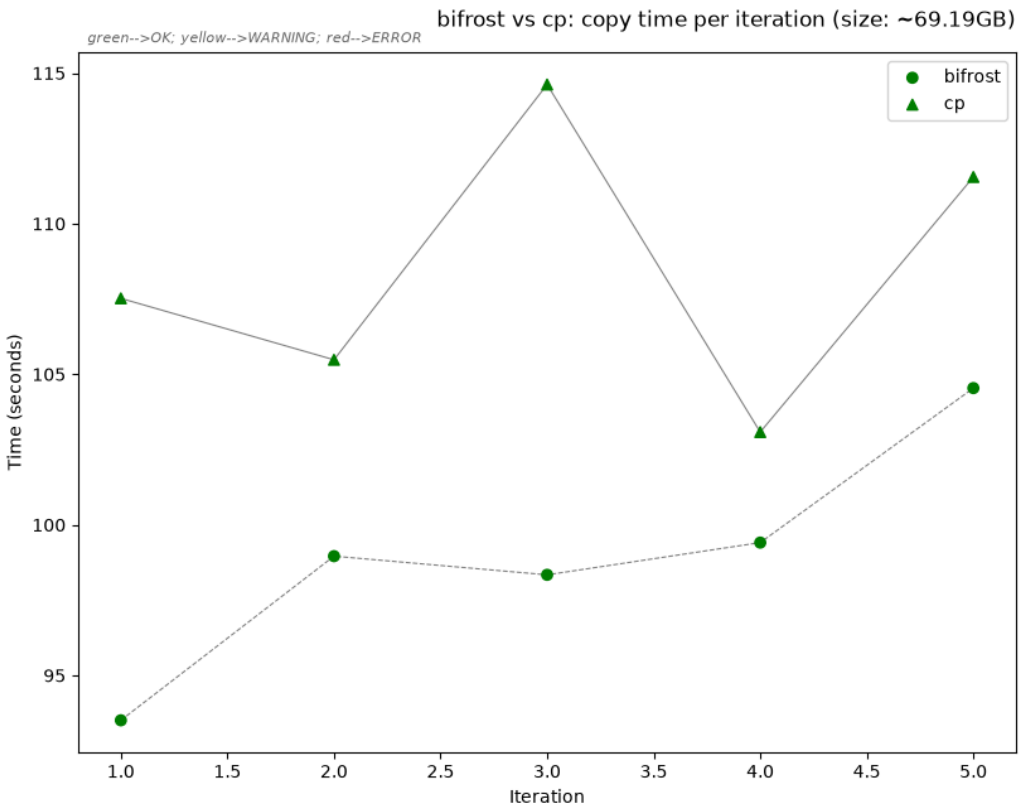


Figure 14: Copy time on ISAC with Bifrost TS scheduler for the 64GB dataset compared against `cp`.

Chapter 6

Discussion

6.1 The hardware ceiling and storage controllers (Sarunya Pumma and Balaji 2019)

To answer the research question, the results show that intelligent clustering of files provides a noticeable speed-up over sequential copying. Bifrost outperforms `cp` across all benchmarks. However, the advantage is limited by hardware. The CN79 RAM disk benchmark represents the theoretical ceiling, showing a significant speed-up when storage bottlenecks are removed and the system is constrained only by memory bandwidth.

When moving to physical solid-state drives, as seen on Minerva and ISAC, software concurrency eventually collides with hardware limits. The sustained load tests on Minerva show that under heavy, repeated writes, SSD thermal throttling becomes the primary bottleneck, reducing the multithreaded gain back to the sequential baseline. Furthermore, the variation in performance between Minerva and ISAC highlights the possible role of the storage controller.

6.2 Limitations: The straggler problem (Hongzi Mao and Alizadeh 2019)

A notable limitation in the current implementation is the handling of singular, very large files. Schedulers like CFS and OS operate by bin-packing or clustering files into distinct workgroups. If a dataset contains a massive file it creates a straggler effect. The thread assigned to that file's workgroup remains blocked long after all other threads have emptied their queues. This leads to resource under-utilization during the end of the transfer, as the asynchronous Tokio workers sit idle waiting for a single I/O operation to complete.

6.3 Future improvements: File chunking

Addressing the straggler problem points toward file chunking as a possible future improvement. Large files could be split into discrete byte ranges and distributed across multiple worker threads concurrently.

Implementing this correctly without violating the integrity of the destination directory presents significant system level challenges. Concurrent, non-overlapping writes to the same file risk data corruption or race conditions if not perfectly synchronized. A possible implementation would likely require pre-allocating the destination file using kernel system calls like `fallocate()` or using a custom, most likely sequential reconstruction algorithm that would piece the files together after the transfer.

6.4 Evolving the Temporal Scheduler (Engin Ipek and Schulz 2006; Ibrahim Umit Akgun and Zadok 2021; Prabhakar and Mishra 2025)

The Temporal Scheduler (TS) represents a good foundation for machine-learning based scheduling, but currently relies on a simple linear regression model derived from static, user-provided disk speeds. Real world I/O costs are mostly non-linear due to cache hits, controller saturation, and varied file types.

Future iterations could move beyond static regressions by gathering telemetry during the transfer. More advanced implementations could even leverage local AI inference to predict completion times.

Additionally, exploring other scheduling algorithms, such as Earliest Deadline First (EDF), could allow the scheduler to dynamically prioritize chunks or files based on approaching I/O saturation thresholds.

Chapter 7

Conclusions

This thesis set out to determine whether a scheduler-driven architecture, running on an asynchronous runtime, could outperform established sequential tools for file copying. Through the development and evaluation of Bifrost, the answer is yes.

The implementation of the Completely Fair Scheduler (CFS), Ordering Scheduler (OS), and Temporal Scheduler (TS) proved that grouping files by size or predicted operating time before transfer gives noticeable performance gains. In optimal, memory-bound conditions, this intelligent clustering allowed Bifrost to achieve a significant speed-up over `cp`. However, benchmarking on physical solid-state drives revealed that these software-level gains are ultimately bounded by hardware, specifically thermal throttling and the throughput capabilities of the underlying storage controllers.

In the end, while sequential tools leave modern hardware capabilities unutilized, maximizing concurrent throughput requires a balance between software scheduling and physical hardware limits. Future developments, such as file chunking and the integration of machine-learning based scheduling models, could show potential to improve the program even further. Bifrost serves as a successful proof-of-concept that the future of system utilities could be in asynchronous, hardware-aware execution.

Bibliography

Anvari, Sana Taghipour, and David Kaeli. 2026. *A Granularity Characterization of Task Scheduling Effectiveness*. https://arxiv.org/html/2602.20561.

Behren, Feng Zhou Rob von, Jeremy Condit, George C. Necula, and Eric Brewer. 2003. "Capriccio: Scalable Threads for Internet Services." *Journal of the ACM,*. https://dl.acm.org/doi/10.1145/945445.945471.

Blumofe, Robert D., and Charles E. Leiserson. 1999. "Scheduling Multithreaded Computations by Work Stealing." *Journal of the ACM,*. https://doi.org/10.1145/324133.324234.

Bramas, Bérenger, and Alain Ketterlin. 2020. "Improving Parallel Executions by Increasing Task Granularity in Task-Based Runtime Systems Using Acyclic DAG Clustering." *Peerj Computer Science,*. https://peerj.com/articles/cs-247/.

Dósa, György, and Leah Epstein. 2018. "The Tight Asymptotic Approximation Ratio of First Fit for Bin Packing with Cardinality Constraints." *Journal of Computer and System Sciences,*. https://www.sciencedirect.com/science/article/pii/S0022000018302319.

Engin Ipek, Rich Caruana, Bronis R. de Supinski, Sally A. McKee, and Martin Schulz. 2006. "Efficiently Exploring Architectural Design Spaces via Predictive Modeling." *ACM SIGPLAN Notices,*. https://doi.org/10.1145/1168918.1168882.

Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Zili Meng, Malte Schwarzkopf, and Mohammad Alizadeh. 2019. "Learning Scheduling Algorithms for Data Processing Clusters." In "Proceedings of the ACM SIGCOMM 2019 Conference." Special issue, *Proceedings of the ACM SIGCOMM 2019 Conference,*. https://doi.org/10.1145/3341302.3342080.

Ibrahim Umit Akgun, Aadil Shaikh, Lukas Velikov, Ali Selman Aydin, and Erez Zadok. 2021. "A Machine Learning Framework to Improve Storage System Performance.." *Journal of the ACM,*. https://doi.org/10.1145/3465332.3470875.

Prabhakar, Karthik, and Durgamadhab Mishra. 2025. *Predictive Modeling of I/O Performance for Machine Learning Training Pipelines: A Data-Driven Approach to Storage Optimization*. https://arxiv.org/pdf/2512.06699.

Sarunya Pumma, Wu-Chun Feng, Min Si, and Pavan Balaji. 2019. "Scalable Deep Learning via I/O Analysis and Optimization." *ACM Transactions on Parallel Computing*,. https://doi.org/10.1145/3331526.

Shintaro Iwasaki, Pavan Balaji, Kenjiro Taura, and Shumpei Shiina. 2021. "Lightweight Preemptive User-Level Threads." *Journal of the ACM,*. https://ieeexplore.ieee.org/document/9018074.

Vazhkudai, Sudharshan, and Jennifer M. Schopf. 2003. "Using Regression Techniques to Predict Large Data Transfers." *International Journal of High Performance Computing Applications,*. https://arxiv.org/pdf/cs/0304037.