Fall 2021

RepeatFS: A File System Providing Reproducibility Through Provenance and Automation

Anthony Stephen Westbrook
University of New Hampshire, Durham

REPEATFS: A FILE SYSTEM PROVIDING REPRODUCIBILITY THROUGH PROVENANCE AND AUTOMATION

BY

ANTHONY WESTBROOK
B.S., University of New Hampshire, 2002
M.S., University of New Hampshire, 2017

DISSERTATION

Submitted to the University of New Hampshire
In Partial Fulfillment of
The Requirements for the Degree of

Doctor of Philosophy
In
Computer Science

September 2021

This dissertation has been examined and approved in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science by:

Dissertation Director, Elizabeth Varki,
Professor of Computer Science

Radim Bartos, Professor of Computer Science

R. Daniel Bergeron, Professor Emeritus of Computer Science

Matthew MacManes, Professor of Molecular, Cellular, and Biomedical Sciences

W. Kelley Thomas, Professor of Molecular, Cellular, and Biomedical Sciences

On May 25, 2021

Original approval signatures are on file with the University of New Hampshire Graduate School.

DEDICATION

This dissertation is dedicated my wife, Jessica, for the love and endless emotional support. To my mom, Donna, for the encouragement throughout my life. And to all the friends and family who have helped me along the way in pursuing my dreams.

ACKNOWLEDGEMENTS

It gives me great pleasure to acknowledge the helpful advice and insightful tutelage I have received over the years from the members of my committee: Professors Elizabeth Varki, Radim Bartos, R. Daniel Bergeron, W. Kelley Thomas, and Matthew MacManes. I returned to UNH with the goal of becoming a more capable scientist, and each of them has gone above and beyond what is required to help me in that pursuit. I sincerely thank them for everything.

I wish to give special thanks to Professor Elizabeth Varki for the countless meetings and overall support. I am truly grateful for her academic guidance and thoughtful perspective on many things. I have no doubt that my time in the PhD program has been a positive and successful experience in large part because of her.

I also wish to give special thanks to Professor Kelley Thomas, who helped make my career in science a reality. The value of the experience I have gained working with him and the Hubbard Center for Genome Studies is beyond measure. Additionally, his style of considerate and respectful leadership is one I try to use as a model whenever possible. I count myself very lucky to have his academic and professional mentorship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

REPEATFS: A FILE SYSTEM PROVIDING REPRODUCIBILITY THROUGH PROVENANCE AND AUTOMATION

by

Anthony Westbrook

University of New Hampshire, September, 2021

Reproducibility is of central importance to the scientific process. The difficulty of consistently replicating and verifying experimental results is magnified in the era of big data, in which computational analysis often involves complex multi-application pipelines operating on terabytes of data. These processes result in thousands of possible permutations of data preparation steps, software versions, and command-line arguments. Existing reproducibility frameworks are cumbersome and involve redesigning computational methods. To address these issues, we developed two conceptual models and implemented them through RepeatFS, a file system that records, replicates, and verifies computational workflows with no alteration to the original methods. RepeatFS also provides provenance visualization and task automation.

We used RepeatFS to successfully visualize and replicate a variety of bioinformatics tasks consisting of over a million operations with no alteration to the original methods. RepeatFS correctly identified all software inconsistencies that resulted in replication differences.

CHAPTER 1

INTRODUCTION

The foundation of science is built upon the acquisition and analysis of empirical data. The formulation and testing of hypotheses are rooted in these observations. This methodology provides the capability not only to understand the world around us but also to make accurate predictions given specific conditions. These predictions require an experimental design and data analysis methods that yield reproducible results; repetition of a scientific workflow with differing results reduces confidence in the prediction. Given the importance of repeatability, the enterprise of science faces a significant struggle. A 2016 Nature survey completed by over 1500 researchers found that more than 70% have attempted to reproduce another scientist's findings without success (Baker, 2016), and more than half were unable to replicate their own results. Subsequently, numerous supporting studies have been performed across a range of biological disciplines, including genomics (Kanwal, Khan, Lonie, & Sinnott, 2017), biomedical sciences (Coiera, Ammenwerth, Georgiou, & Magrabi, 2018), and computational biology (Garijo et al., 2013), each noting challenges involved in ensuring reproducibility.

While many factors can contribute to irreproducible research, such as selective reporting or missing data, each study notes computational analysis as especially problematic, often responsible for introducing unintended variation into replication studies. Respondents of the Nature survey corroborate this concern, with over 82% noting that "insufficient computer code or protocol information" is at least sometimes involved, if not very often or always. Reasons for this

stem from the vast number of available software applications and reference databases (Davis-Turak et al., 2017); differences in versions, parameters, and configuration files (Kim, Poline, & Dumas, 2018); and a wide variety of data formats and conversion techniques (Lewis, Breeze, Charlesworth, Maclaren, & Cooper, 2016). As a typical workflow has the potential for thousands of combinations of these attributes, recording the exact environment and steps performed by a researcher is critical in later replicating the analysis. This list of steps, known as the provenance, includes every action that had a causal effect on the results of the analysis, such as writing to files or running applications.

Scientific computing software has attempted to address this reproducibility issue with limited success. Virtual environments, such as Docker (*Docker*, 2020) and Anaconda (*Anaconda Software Distribution*, 2020), ensure the versions of software match between original and repeated analyses, but cannot verify these programs are executed using the same parameters, reference databases, or other runtime options. Analysis platforms such as Galaxy (Afgan et al., 2018) and QIIME 2 (Bolyen et al., 2018) require the researcher to operate within the provided set of tools, forcing methods to be redesigned and precluding the ability to use many external packages. Generic pipeline frameworks such as Ruffus (Goodstadt, 2010), Bpipe (Sadedin, Pope, & Oshlack, 2012), Snakemake (Köster & Rahmann, 2012), and SoS (Wang & Peng, 2019), while designed to work with any program, require the researcher to write scripts to migrate their workflow into the framework. This not only requires learning an additional language and rewriting each step of the workflow but potentially introduces new mistakes. Thus, these systems have limitations that preclude them as complete reproducibility solutions.

Though these examples each have unique constraints that affect their ability to record an accurate record of provenance, there is a common conceptual methodology that they all share.

Each system requires the user to manually register his or her action to be recorded, either directly through API, or indirectly through using a predefined set of platform compatible tools. By requiring continual human interaction or instruction to achieve functional correctness, the system is susceptible to missing or erroneous data and therefore is likely to provide a partial or incorrect record of provenance. These inconsistencies may be random, adding noise to the provenance signal, or biased in nature, presenting a misleading account of the actions performed by the researcher.

In order to guarantee a correct and complete history of actions, the system must be designed without dependence on human direction; it must not require manual instruction to register operations, nor confine the user to a subset of applications that may be ignored in lieu of external, incompatible tools. The system must record provenance automatically, transparently, and completely, thereby eliminating the potential for loss of historical record. Additionally, it must do so in an application agnostic manner, ensuring that the user does not introduce information loss through executing an incompatible program.

The RepeatFS methodology proposes achieving this through two related conceptual models, each appropriate for different use case scenarios. The **reactive model** that directly automates the process of recording provenance as outlined above, and the **proactive model** that also automates the actual actions performed by the user. While the reactive model produces a complete list of actions after they are performed, the proactive model will produce this list of actions prior to executing them. The reactive model allows customization and freedom, while the proactive model provides systematic and uniform processing.

1.1 Reactive Model

Since the RepeatFS methodology requires automatic registration of actions performed by a user, any implementation must be notified of each of these actions in some systematic way. To identify potential mechanisms that can provide this functionality, it is necessary to first define what type of actions must be recorded to provide a complete provenance record for computational processing and analysis. As this methodology focuses on provenance for ensuring reproducibility, only actions derived from a deterministic process, or from a stochastic process whose random variables are seeded by deterministic values, must be recorded. A researcher usually begins with a set of input files, processes these files by executing one or more software applications, and receives a set of output files from the software. Additionally, there may be intermediate and temporary files created by the software which may or may not be of interest. As operations of interest are deterministic, input from outside of the system is considered non-deterministic; recording and replicating data from these sources is considered outside the scope of the RepeatFS methodology.

Given the requirement of creating a record of each process and file that is read or written from a starting set of input files to the ending output files, actions are defined as any process that performs I/O in a way that has a causal effect on the set of files representing the processed output (referred to as “result files”). These are not only the processes that directly access the result files, but also those that access input and intermediate files, as these ultimately have a causal relationship with the result files; changes to the input files normally have a direct impact on the data present in the result files.

The RepeatFS methodology therefore requires a mechanism to inform the implementation of when a process interacts with the file system, though the reactive model only requires being

notified of such an action while or after it is performed. Notifications must provide information about the process performing the operation and the file it is accessing, and this information must be sufficient to replicate the operation. Since the current scope of the RepeatFS methodology is confined to a single system with a shared clock, causality may be inferred by total ordering derived from this clock. Though the RepeatFS methodology is agnostic to the mechanism used to provide process and file information, selection of this mechanism should also take performance into consideration.

1.2 Proactive Model

In addition to recording provenance in an automated fashion, the proactive model provides automation of user performed tasks. Not only must it meet the requirements outlined for the reactive model, but the proactive model must also be aware of actions prior to execution. The reactive model is only aware of provenance once a result file's entire workflow has been performed. Should a user wish to repeatedly perform this workflow for different result files, the user would initially need to execute all steps in the workflow for one result file to record the provenance; this provenance could then be used to replicate the workflow for the other result files. Not only is this a time-consuming and redundant step for commonly performed, vetted workflows, but it also allows for unforeseen or erroneous actions to be included in the provenance record. All subsequent replications of this workflow for other result files would therefore contain these unwanted actions as well. Performing quality and adapter trimming on read files from a DNA sequencer is one such example. This workflow is performed often and has numerous parameters that affect the number of nucleotides trimmed from the read files. Using the reactive model, the provenance associated with trimming the first file would need to be recorded before replicating

the workflow with the remaining files. However, any erroneous parameters used during the first execution would be replicated in all future executions.

To address these limitations, the proactive model requires the actions in a workflow be defined within the implementation prior to execution. When a user wishes to perform the workflow associated with producing a result file, he or she directs the implementation to execute these actions. Unlike the reactive model, no initial provenance is required to create a result file. Additionally, the chances of unwanted actions to be included in the provenance is significantly reduced by executing a predefined set of actions. Since the proactive model also meets the requirements of the reactive model, these actions can also be verified by examining the provenance to ensure only the expected processes are present during the execution of the workflow.

This model aligns itself especially well to repetitive tasks that are commonly performed by the researcher, such as converting file types or filtering data. The proactive model ensures that these tasks are always performed uniformly, reducing the risk of erroneous or unexpected results. The implementation of the proactive model should allow the user to define and execute workflows easily while maintaining compatibility with the reactive model; aside from initially providing the workflow definitions, functional correctness must not depend on manual steps performed by the user.

1.3 File Operation Notifications

In order to test the RepeatFS methodology and provide a practical software solution, we implemented both the reactive and proactive models. These models require an automated mechanism for receiving notifications of file operations; choosing the appropriate interface to provide this information was a critical decision in the design process. These mechanisms are

closely tied to the operating system, with different and unique API available to each, so we chose to investigate the options available for Linux. Not only does the Linux kernel provide a robust set of libraries for interfacing with the operating system at different levels of detail and control, it is also a widely used platform in scientific computing, making it an appropriate choice.

File operation awareness can be provided using different methods in Linux. When a Linux process issues a file system operation, the user mode function makes a system call since file I/O is a protected action performed exclusively by the kernel. Once the transition from user space to kernel space has occurred, file system operations are sent to the Virtual File System (VFS) (Gooch, 2005). This abstraction layer is responsible for routing I/O operations to their appropriate file system drivers. Each file system driver is then responsible for handling the request per the particular file system specification and may relay these requests to a variety of lower layers, such as block device drivers for disk storage or memory related API for RAM storage. Depending on the system configuration, many of these subsystems notify other kernel modules of actions they receive, such as the kernel informing the inotify module of certain types of file activity.

Since every user mode initiated kernel operation requires a system call, interfacing RepeatFS at this level would be problematic. Though the required file operation information, in addition to potentially other useful related data, would be available through this method, it would be subject to a vast number of unrelated calls. Not only would this require filtering routines to ensure only the desired calls were monitored, but extreme care would also need to be taken to ensure that system-wide performance or stability issues were not introduced from working at such a universally utilized portion of the kernel. Conversely, interfacing with each individual file system driver would not be practical, as RepeatFS would need contain specialized code to support each one individually and would require constant upgrades to provide support for any future file

systems. Lastly, while making use of notification-oriented modules such as inotify resolves both issues, these modules are not always present or configured in most Linux systems, and require administrative access and expertise to install. For these reasons, we decided to build RepeatFS using FUSE (Szeredi & Rath, 2019), a popular mechanism that directly interfaces with VFS and does not require administrative access. To gain access to a large variety of included functions and external libraries, RepeatFS was implemented using Python. The file system structure and FUSE integration are detailed in chapter 2.

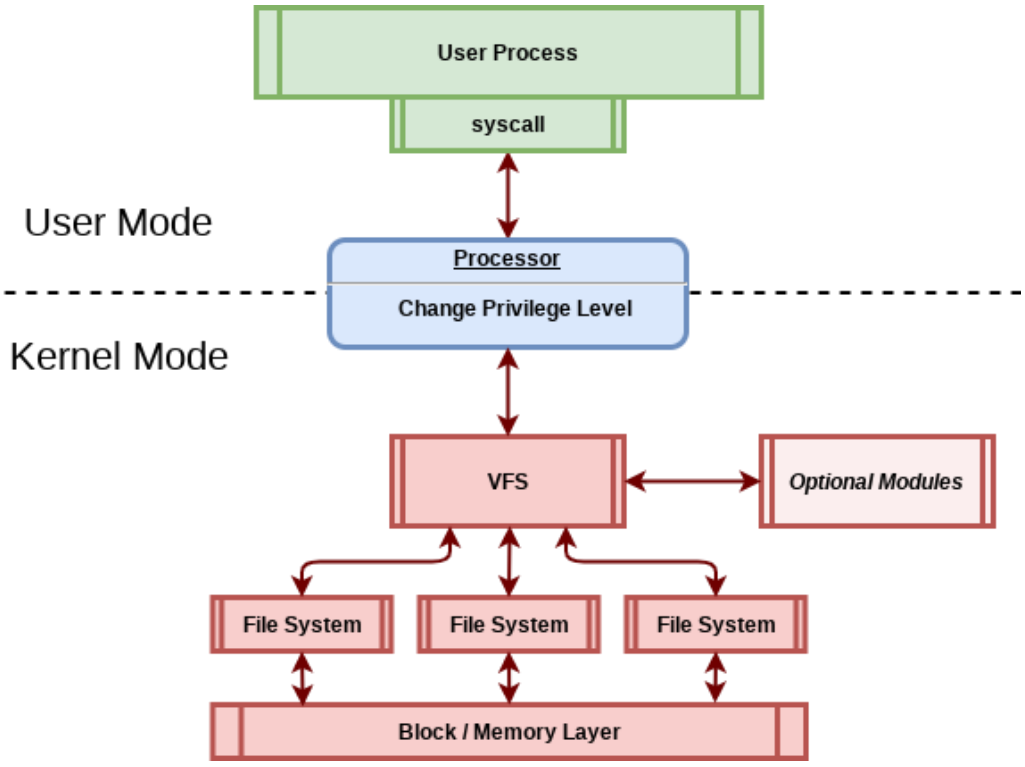


Figure 1: System modules involved in file I/O

1.4 Reactive Model Implementation

RepeatFS implements multiple core features helpful for typical analyses, such as visualization and replication. These features are critical in evaluating the efficacy of the RepeatFS

methodology and allowed us to test the hypothesized abilities of our models. RepeatFS seamlessly records all file system operations of any program automatically within a database without the need for manual intervention. The recorded software is not required to make any special API calls, nor must the user manually register any operations during their workflow. Also, RepeatFS provides an interactive visualization of the provenance associated with a result file, allowing the researcher to view all the programs involved in the creation of a result file, the order they were executed, command line arguments utilized, temporary files produced, etc. A visual depiction of the historical record is useful for verifying that no unintended actions or effects were produced during the lifetime of the workflow.

In addition to simply visualizing provenance, RepeatFS also provides functionality for fully replicating all operations in a result file's provenance, allowing the researcher to repeat their informatics pipeline with an identical execution to the original run. As the database contains an ordered record of all processes that have a causal relationship with the result file, RepeatFS can effectively reconstruct missing pipeline script files by directly executing each program originally run within the script. RepeatFS can also verify the fidelity of subsequent replications against the original run and notes any inconsistencies that may result in a deviation in the expected results, such as differing versions of software or programs failing to execute as expected from resource exhaustion. The provenance related details of RepeatFS are discussed in chapter 3.

1.5 Proactive Model Implementation

Since the proactive model requires predefined lists of commands to be executed, RepeatFS requires the ability to store and execute these workflows. To provide a consistent and seamless user experience, these functions are integrated into RepeatFS as memory-resident files called

Virtual Dynamic Files (VDFs). Each VDF is associated with a particular list of commands to execute; accessing the VDF through normal file system operations will automatically cause these commands to run, and their output will be cached within the VDF's memory allocation. This data is then returned to the calling program, and future requests for this same data will be retrieved from the cache. From the user's perspective, reading a VDF retrieves data in the same way reading any disk file would. Unlike normal files however, the data is not already present on disk – it is instead generated from the process output in real-time when accessed.

By ensuring that predefined workflows are always executed in a uniform fashion, VDFs are well suited for common or repeatedly run tasks across many areas in scientific computing, such as converting between file formats, extracting tabular data, or indexing references. Adhering to the proactive model reduces the chances of inconsistencies across each iteration of the task being performed. Since VDFs are part of the file system, all VDF file I/O is recorded in the database, ensuring reactive model compatibility as well. Full provenance for any VDF result file will be available after accessing the VDF, and may be visualized, replicated, and verified, the same as any normal disk result file. They may also be copied to normal disk files should the user desire a non-volatile copy of a result file. VDFs contain a caching and process handler system that is detailed in chapter 4.

1.6 Performance Testing

Though reproducibility is the primary focus of RepeatFS, the importance of efficiency and performance in scientific computing cannot be ignored. Typical genomics pipelines running on High Performance Computing (HPC) platforms can often take weeks to complete (Henson, Tischler, & Ning, 2012). Introducing too large of a delay to I/O-intensive processing and analysis

can magnify already long timeframes into unacceptable ranges. For this reason, performance is a practical consideration that cannot be ignored. We evaluated RepeatFS across a number of metrics for both normal files and VDFs. These results are outlined in chapter 5.

CHAPTER 2

FILE SYSTEM FRAMEWORK

2.1 Introduction

The central and most critical requirement of RepeatFS is being notified of each file system operation that could potentially have a causal relationship to any result files. This is not only a necessity to fulfill the requirements of the conceptual RepeatFS methodology but also a seamless way of implementing practical user-facing features such as provenance visualization and replication verification. By designing RepeatFS as a FUSE file system, the framework operates as an event-driven architecture in which internal routines are triggered in response to particular file system operations, such as *truncate, read, write,* and *close* system calls. Upon receiving a file system request, RepeatFS performs the appropriate actions and then return the expected data structures to the operation system. These actions may be performed in response to all file system requests such as recording provenance, as any I/O operation could potentially be causal to the result file. However, specialized responses may also be state and I/O operation dependent, including initializing data structures in response to an *open* operation. Though RepeatFS does provide an API service for externally directing its operation, the core functionality is modeled around directly receiving and responding to file system operations.

2.2 FUSE Overview

Filesystem in Userspace (FUSE) is a widely distributed framework for implementing file systems on UNIX/Linux compatible systems, with over 50 open-source projects built upon it (awesomeopensource.com, 2021). In addition to being available in all major system package

managers, it has libraries available for most popular languages, such as C, Python, and Java. Unlike file system drivers implemented as kernel modules, FUSE based file systems can be installed and loaded in user space, allowing the user to easily manage them without requiring system administrator privileges or alterations to the kernel. FUSE allows for user mode operation through two components: a kernel module that directly registers with the VFS as a standard file system driver, and the libfuse user library that interfaces with the kernel module, providing communication between the FUSE file system and the FUSE kernel module. Upon mounting a FUSE based file system, that instance of the file system will use libfuse to begin communication with the kernel FUSE module; it will discontinue this communication when the file system is unmounted. Assuming the file system itself allows for it, multiple instances of the same FUSE file system may be mounted concurrently – each will have its own unique mount point path.

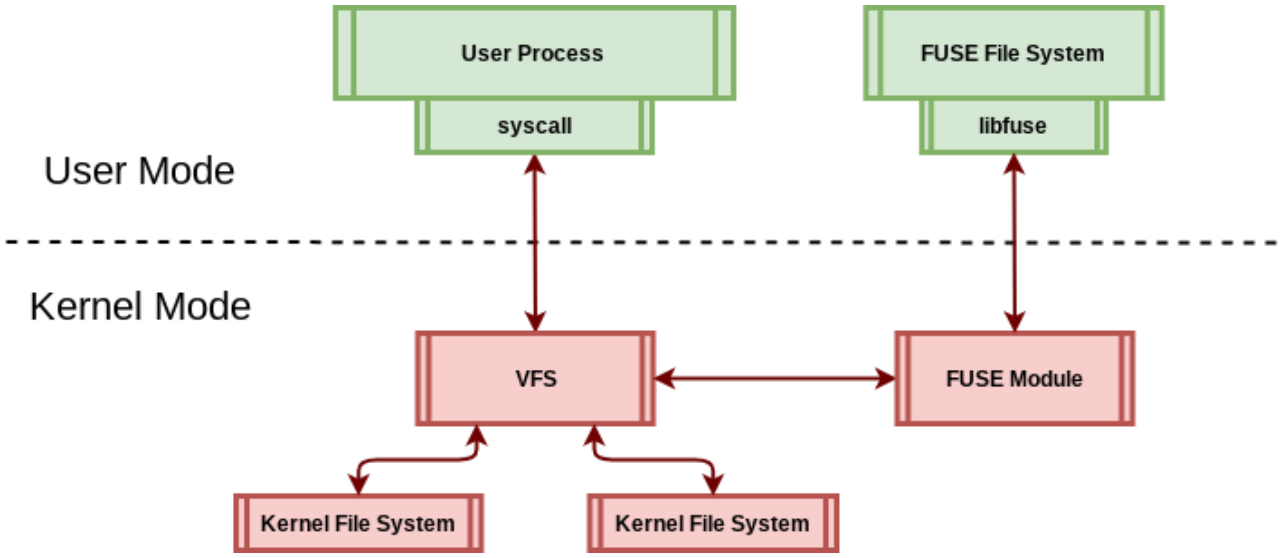


Figure 2: FUSE overview

Since the FUSE kernel module receives operations from the Linux VFS, and the VFS is responsible for interfacing with Linux page cache, then I/O operations targeting FUSE file systems

are automatically sent to the page cache when configured to do so. This is critical for acceptable performance, since reading from the page cache is an optimized, kernel-based memory operation, which is orders-of-magnitude more efficient than direct IO. FUSE includes support for configuration attributes associated with its file descriptors that direct the VFS whether to enable or disable cache support for read operations targeting the descriptor. FUSE also supports writeback support for asynchronous writes to the page cache, though not all libfuse wrappers support this functionality.

Implementing a FUSE based file system involves registering a callback function within the file system code for each system call the file system will handle. These functions are automatically called by the kernel FUSE module when an operation targeting a file on a mounted FUSE file system occurs. Information relevant to the I/O operation, such as descriptor identifiers, file positions, and access modes, are passed as parameters into each function, and the function must return the appropriate data structures and status code expected by the FUSE kernel module. Assuming these requirements are met, the actual actions performed in response to any operation are at complete discretion of the FUSE file system itself. The file system may choose to relay operations to another file system, service them internally, ignore them, or any combination of these. FUSE also maintains data structures containing other contextual metadata, such as the process ID (PID), user ID (UID) and group ID (GID) associated with the process issuing the file system request.

2.3 Python Interface

FUSE's libfuse is distributed as a native shared Linux library with standard headers intended for C/C++. Though Python supports calling functions within shared system libraries via

the ctypes module, doing so is unwieldy for all but the most trivial cases, as it requires effectively duplicating C structures and function prototypes within Python: specifying the function name, parameter and return data types, whether to use values or references, etc. For a framework like FUSE, this is a considerable task since there are numerous functions and complex data structures involved in even its basic operation.

To avoid this unnecessary complexity, we decided to use the publicly available fusepy Python module (Honles, 2018) to provide FUSE support. Fusepy internally makes use of ctypes to provide almost full support of all FUSE features to Python programs, including callbacks for all file system operations, page cache read compatibility, and contextual metadata. As of the current version, fusepy does not support asynchronous page cache writeback. Since fusepy calls functions within libfuse, this library must still be present on the system; fusepy does not take the place of libfuse, but instead wraps its functionality with Python functions and classes.

Fusepy encapsulates all of the libfuse function callbacks as instance method stubs within a parent class. By default, these stubs simply return a success error code to the respective libfuse function but perform no action. We extended this class with our own version, and each instance method is implemented to provide the appropriate functionality for the respective file system call. Additionally, fusepy supports concurrent I/O and will execute each callback with a unique thread ID. We designed RepeatFS to be thread-safe and implemented a variety of locks and prioritization schemes to ensure safety and liveness properties. Fusepy also catches and handles FUSE specific Python exceptions which are then communicated in the appropriate manner to libfuse error handling. Accordingly, RepeatFS confines and converts all exceptions to these types where applicable.

2.4 RepeatFS Passthrough Routing

Like all file systems, RepeatFS instances are mounted at a specific location in the directory tree. Prior to mounting, this path typically refers to an empty directory in an underlying file system. After an instance of RepeatFS has been mounted, this path refers to the root of that instance's directory tree. Since FUSE will only send notifications regarding operations that target that instance of the FUSE file system, the user is required to ensure all processes refer to paths under the RepeatFS mount point. Any processes that attempt to access a file located under such a path will be intercepted by RepeatFS via FUSE and handled accordingly.

With the exception of VDFs discussed in chapter 4, we did not design RepeatFS to manage and store its own files. If the file system were designed in this manner, it would require the user to copy disk files to RepeatFS prior to accessing them during analysis; since RepeatFS only receives notifications for files under its mount point, historical provenance would only exist for files located at this path. This methodology would be inefficient and cumbersome for the end user. Instead, we designed RepeatFS to send all operations in a passthrough manner to a "source directory" or directory on another file system at a different path in the directory tree. This source directory and the mount point are specified by the user when starting a new instance of RepeatFS. When a request is made to list files within the RepeatFS mount point, RepeatFS will instead list the files located within the source directory. The user may also perform I/O operations upon the files shown under the RepeatFS mount point; RepeatFS will record the operation and perform other relevant processing, calculate the destination of the operation, and relay the operation to the actual file located under the source directory if applicable.

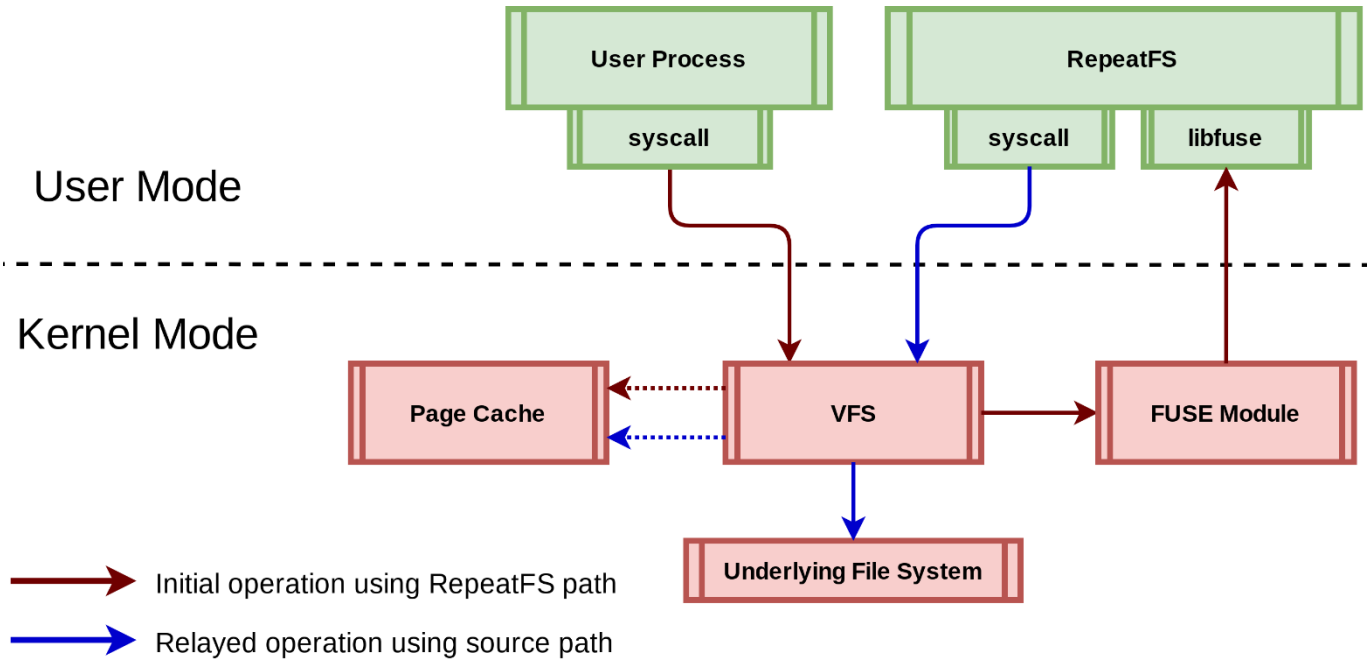
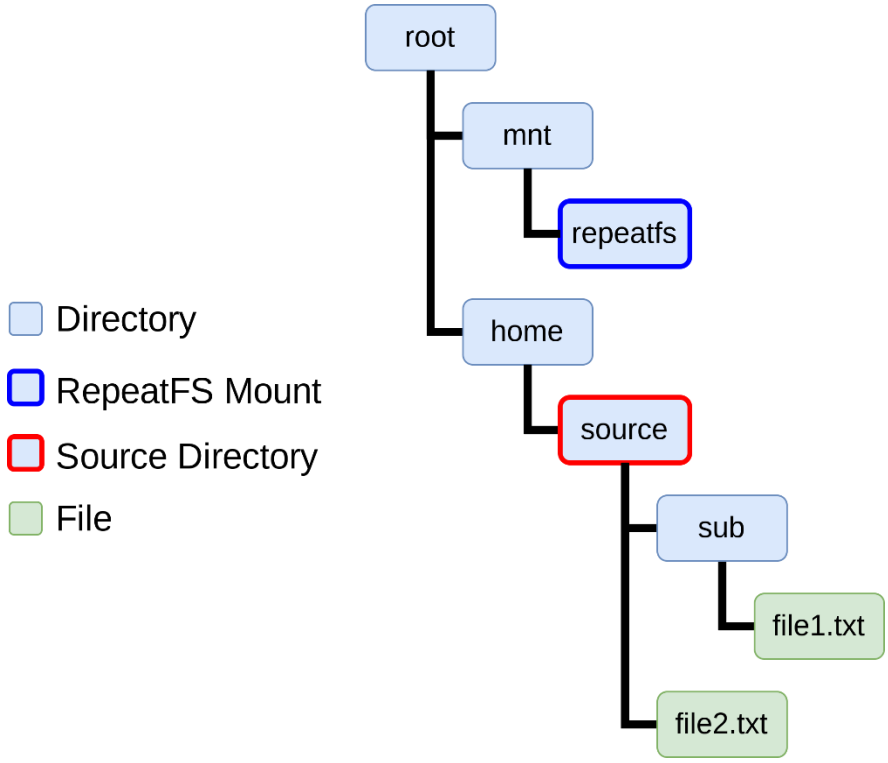


Figure 3: RepeatFS passthrough interaction with the kernel

System calls in Linux related to file operations may be organized into two broad categories (Table 1): those referencing a file path, and those referencing a file descriptor. Immediate and atomic operations that do not require maintaining a state, such as moving or deleting a file, refer directly to the file path. However, since state information regarding the access mode, position, and other flags is recorded for open files, each instance of an open file receives a descriptor that references this state data. The *open* call is responsible for creating this file descriptor; it takes a file path and returns a file descriptor.

Since many file system calls take only a path as a parameter and therefore do not maintain state information, RepeatFS contains functionality for analyzing paths to determine how to handle and route the operation (Figure 4). The path analyzer first removes the mount point path portion from the absolute path being analyzed, leaving a path relative to the root of the mount point. This relative path is then appended to the absolute path of the RepeatFS source path. If this results in a

valid path on the underlying file system, the operation is repeated using this new path as the target. If the resulting path is invalid, RepeatFS checks if the path ends with a designated special character, by default a plus sign. If it does, this is considered a path handled by RepeatFS internally, such as for VDFs or API calls (Sections 2.5, 4.1), and routes the operation to other subsystems within RepeatFS (Figure 5: Operation Processing). If the path also fails this check, it is considered an invalid path and the appropriate error condition is returned to FUSE.



Absolute Path	Relative Path	Translated Path	Routing Destination
/mnt/repeatfs/sub/file1.txt	sub/file1.txt	/mnt/source/sub/file1.txt	Underlying file system
/mnt/repeatfs/file2.txt	file2.txt	/mnt/source/file2.txt	Underlying file system
/mnt/repeatfs/file2.txt+	file2.txt+	/mnt/source/file2.txt+	RepeatFS
/mnt/repeatfs/file3.txt	file3.txt	/mnt/source/file3.txt	Invalid path

Figure 4: RepeatFS path translation and routing

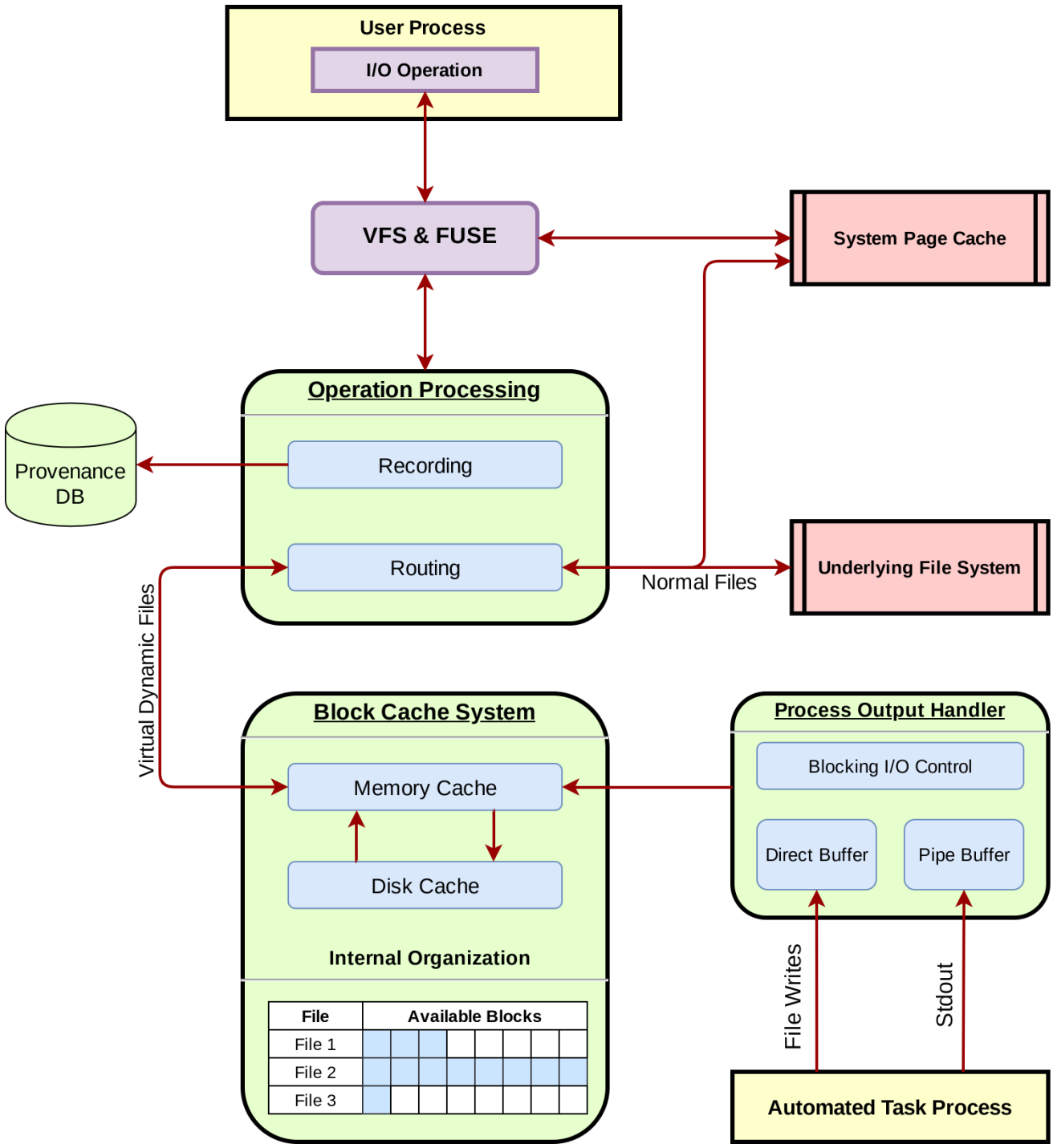


Figure 5: RepeatFS internal flow

For descriptors associated with open files, RepeatFS caches the path translation information within a class instance and attaches it to the descriptor record, avoiding the need to continually reparse the same path for each I/O operation performed upon a descriptor, such as *read* and *write* calls, which may be issued in high volume for large files. The translation instance contains multiple combinations of absolute and relative paths, each starting with the mount path or the source path. These paths are all calculated at instantiation time to improve performance, since each version of the absolute and relative path are used numerous times in other parts of RepeatFS depending on the file system operation being processed.

Table 1. FUSE and RepeatFS system calls

FUSE	RepeatFS	Parameter	FUSE	RepeatFS	Parameter
access	get_access	path	open	open	path
chmod	change_mode	path	create	open	path
chown	change_owner	path	read	read	descriptor
getattr	get_attributes	path	write	write	descriptor
opendir	open_directory	path	truncate	truncate	both
readdir	get_directory	path	flush	sync	descriptor
readlink	get_link	path	release	close	descriptor
mknod	create_node	path	fsync	sync	descriptor
rmdir	remove_directory	path	getxattr	n/a	n/a
mkdir	create_directory	path	setxattr	n/a	n/a
statfs	fs_stats	path	listxattr	n/a	n/a
unlink	unlink	path	removexattr	n/a	n/a
symlink	make_symlink	path	ioctl	n/a	n/a
link	make_hardlink	path	bmap	n/a	n/a
rename	rename	path	lock	n/a	n/a
utimens	utimens	path			

FUSE supports all standard file I/O related system calls in Linux. RepeatFS currently supports the majority of these, with a focus on implementing common calls likely to be made by software run by a typical researcher. The absolute path to the file is used in most calls, while descriptors are used for all calls related to open files. Calls marked “n/a” are currently unimplemented.

2.5 API System

Although most RepeatFS actions are responses to file system requests, there are deliberate, out-of-band actions not associated with a specific system call; these actions must be controlled by signaling that occurs outside of normal operation processing. Examples include terminating a RepeatFS instance, initiating provenance replication, or adjusting runtime configuration options. These are all actions that control or configure RepeatFS itself and do not target a file system; none of them are an appropriate response to any particular file operation, such as *open*, *rename*, or *write* system calls.

To accommodate this type of functionality, RepeatFS provides an API system and accompanying client for sending and receiving commands that are unrelated to specific system calls. Interfacing with the API system is provided by reading and writing a hidden "API control" file. This API control file is virtual and does not correspond to a real file on disk. Instead, file system operations targeting the API control file's name, regardless of the parent path, will always succeed for any parent path under the RepeatFS mount point. RepeatFS provides two methods of interacting with the API control file.

All features of the API system can be utilized using the "direct" access method. The direct method consists of opening the API control file, writing to the file to indicate which API function to execute, and reading from the file to receive a response. Reading from the API control file operates as a pipe and will block until there is more data present, or the operation is complete. This allows the API system to be used in an asynchronous manner and receive results from any requested actions in real-time. API sessions and threads are created per file descriptor, allowing the API control file to be called concurrently and process multiple actions simultaneously, even by the same calling thread. All input read by, and output written to, the API control file is encoded

in Unicode and organized using the standard JSON format (Bray, 2017), promoting interoperability with other third-party software interfacing with the API system.

RepeatFS also provides an "indirect" access method for the API control file. Unlike the direct method, the indirect method does not support per-descriptor sessions and uses a single, global session. Additionally, it is only compatible with a subset of API functions that do not require real-time I/O, such as those that perform a single, simple action, like terminating RepeatFS or setting a configuration option. Indirect access of the API control file does not require writing to the file; instead, commands are sent to the API system by appending special control characters and the JSON command data to the end of the API control file name when opening it for reading. When RepeatFS parses these special control characters while parsing the file's path information, it relays the JSON command to the API system, which then immediately executes the command. The results of the command may then be read from the API control file. Although more limited in available actions, the indirect method allows RepeatFS to be controlled by systems that do not have write capability, or block it artificially for security reasons, such as modern web browsers that do not allow writes to be made to the client's system. This method is utilized for visualizing provenance as explained in chapter 3. The indirect method allows a read operation to effectively act as a write operation by encoding the command into the filename.

2.6 API Client

To assist users performing API commands, we provide an API client integrated into the main RepeatFS script. This client accepts command line arguments from the user, creates a corresponding JSON request, and writes the request to the API control file. The client will search for the API control file in the current working directory, but the user may also specify a different

directory for manually selecting other RepeatFS instances. For commands requiring real-time I/O, the client continually reads from the API control file and displays results to the screen. The client will exit upon receiving a JSON value indicating that command execution has completed. The client currently supports two commands: a “shutdown” command for terminating a RepeatFS instance, and a “replicate” command for initiating provenance replication for a result file.

2.7 Usage and Configuration

The main RepeatFS script recognizes a set of primary commands that dictate the action to perform. Each primary command is associated with a series of command line arguments and switches that provide additional options to the command (Tables 2, 3). Commands are generally grouped into two categories: those involved in mounting a RepeatFS instance, and those calling functions within the API client. Creating a mount point for a new RepeatFS requires specifying source and mount directories, the latter of which must be empty. By default, RepeatFS will operate as a daemon (Finney, 2021) when creating a new instance and will return to the shell upon successfully mounting the requested directory. The user may also direct RepeatFS to remain in the foreground in order to monitor informational and error messages reported to the console for troubleshooting purposes. Commands associated with API calls will run in the foreground until completion, writing data to standard streams and files as directed by the user.

Table 2. RepeatFS command-line arguments

Command	Argument	Required	Description
mount			Create new RepeatFS instance
	target	yes	Target directory to monitor
	mount	yes	Mount point directory
	-h, --help	no	Display usage
	-c <PATH>	no	Configuration and database directory
	-f	no	Keep daemon in foreground
	-a	no	Allow other accounts to access mount
	-p	no	Disable provenance recording
	-v	no	Level of verbosity (output, call, debug, IO)
replicate			Replicate RepeatFS provenance
	provenance	yes	Provenance export to replicate
	-h, --help	no	Display usage
	-r <PATH>	no	Target directory to store replicated files
	-c <PATH>	no	Configuration and database directory
	-l	no	List replication commands only
	-e [EXPAND…]	no	Expand and recreate specified commands
	--stdout <PATH>	no	Redirect execution stdout to target file
	--stderr <PATH>	no	Redirect execution stderr to target file
	-v	no	Level of verbosity (output, call, debug, IO)
generate			Generate a configuration template
	-h, --help	no	Display usage
	-c <PATH>	no	Configuration and database directory
	-v	no	Level of verbosity (output, call, debug, IO)
shutdown			Unmount RepeatFS
	-h, --help	no	Display usage
	-m <PATH>	no	Mount point directory
	-c <PATH>	no	Configuration and database directory
	-v	no	Level of verbosity (output, call, debug, IO)
version			Display version information
	-h, --help	no	Display usage
	-v	no	Level of verbosity (output, call, debug, IO)

Each RepeatFS primary command supports a number of command line arguments. Where indicated, some arguments are required for a particular command, such as “mount” and “replicate”. These options are reported to the user when running RepeatFS without any options.

Table 3. Example RepeatFS commands

Generate a new RepeatFS configuration file
repeatfs generate
Start RepeatFS instance with default configuration file, allow other accounts access
repeatfs mount ~/source ~/mount -a
Start RepeatFS instance with custom configuration file, keep in foreground
repeatfs mount ~/source ~/mount -c testing/custom -f
Replicate provenance, expand/recreate script by ID
repeatfs replicate -r ~/mount/replication -e comp1|16182506.47|8349 files/provenance.json
List commands in provenance but do not replicate
repeatfs replicate -l files/provenance.json
Shutdown RepeatFS instance from any current working directory
repeatfs shutdown -m ~/mount

A RepeatFS configuration file must be generated prior to creating a new RepeatFS instance or replicating a provenance file. Specifying the mount point is not necessary when issuing a shutdown command if the current working directory is under the mount point.

In addition to command line arguments, a file containing configuration values is required for many actions. The user may also direct RepeatFS to create a template of this configuration file containing default options which may then be customized as required. The configuration is organized into two sections: global options that relate to core RepeatFS functionality, and VDF options that configure each VDF that will be available within the instance (Tables 4, 5). Options not present within the configuration file or commented out will be internally replaced with default values. By default, the configuration file is created within a predefined hidden folder within the user's home directory, and RepeatFS will search for the file at this location when loading configuration values. This location may also be overridden through an optional command line argument should the user wish to make use of an alternate configuration file.

Table 4. RepeatFS configuration options

Option	Type	Default	Description
suffix	global	+	VDF directory suffix
hidden	global	False	Prepend '.' to virtual directory paths
invisible	global	True	Hide VDF directories from directory listing
block_size	global	1048576	Filesystem block size
store_size	global	1073741824	Maximum memory cache size
read_timeout	global	1.0	Read priority timeout
cache_path	global	/tmp/repeatfs.cache	Disk cache path
io_epsilon	global	7.0	Epsilon to consider IO simultaneous
api	global	.repeatfs-api	RepeatFS API and control file
match	vdf	None	Generate VDF for matching filenames
ext	vdf	None	Append extension to VDF filename
cmd	vdf	None	Command to populate VDF
output	vdf	stdout	Receive stdout or file-based output
internal	vdf	False	VDF calls internal routine

When not specified within the configuration file, RepeatFS will use the default value for that option. Options marked as "global" may only be specified once per file, while options marked as "vdf" may be specified per VDF entry within the configuration file.

Table 5. Example RepeatFS configuration

```
# Override BCS block size
block_size= 2097152

# VDF definition: Convert FASTQ to FASTA
[entry]
match=\.fastq$
ext=.fasta
cmd=seqtk seq -A {input}

# VDF definition: Sort text and tabular files
[entry]
match= \.(txt|csv|tsv)$
ext=.sorted
cmd=sort {input}
```

This configuration specifies two VDF definitions. The first will create a VDF ending in ".fasta" for any file ending with ".fastq". It uses "seqtk" to convert from the first file format to the second. The second definition creates a VDF ending in ".sorted" for any file ending with ".txt", ".csv" or ".tsv". It uses "sort" to sort each line of the file lexicographically. This configuration also overrides the BCS block size.

CHAPTER 3

PROVENANCE

3.1 Introduction

The provenance of an analysis includes every action that had a causal effect on the result file. The RepeatFS methodology needs a complete record of this provenance provided in an automated fashion to fulfill the requirements of the reactive and proactive models. These historical records provide the foundation for visualization, replication, and verification. The file system framework provides a mechanism for receiving each file system operation, but storing, organizing, and processing these operations is a nontrivial task. Each file operation must be stored in a way that not only records information particular to that action but also allows for reconstructing the causal relationship between these actions.

A file's contents are the result of a running process performing a write operation on that file at some time instant. In the context of the RepeatFS methodology, time is treated as an absolute value as measured by a single, shared clock on a single system. Many write operations may be performed, and each has a potential causal relationship with the file's contents. Though a write operation may leave a file unchanged by writing zero bytes or overwriting with identical data to what is already present within the file, a typical write operation will add or overwrite with new data, thus changing the file. This process has a direct causal relationship upon the file's contents; if the process had not run, the file's contents would likely be different. The time instant at which a process performs a write operation is of critical importance in calculating causality as well. If process *A* writes to a file at time instant 1, a snapshot of the file is taken at time instant 2, and

process *B* writes to a file at time instant 3, only process *A* has a causal relationship to the state of the snapshot, whereas process *B* cannot (Figure 6).

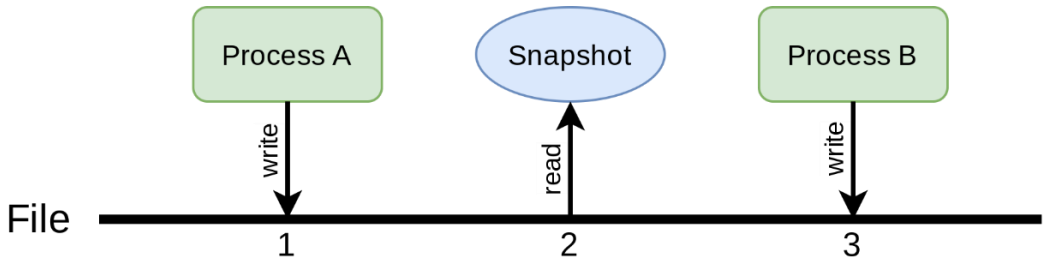


Figure 6: Causality from write operations

Processes may also read the contents of files, normally with the purpose of using the data in some manner, such as displaying it, transforming it, calculating values from it, storing it, transmitting it, controlling program flow depending on the contents, etc. Though the ways in which the data might be used are diverse, the actual specific data values are relevant in all cases. While a process may read a file and completely ignore the data read during the operation, this is redundant, not typical, and causally irrelevant. It is generally correct to assume that if a process reads a file, it is using the data in some consequential manner.

Since the data obtained from a read operation has an impact on a process's state and operation, a causal relationship exists transitively between the state of a file when read by a process and the state of a file written at a later time by the same process. The files read and written may be the same file or different files. If process *A* reads from file *X* at time 1, writes to file *Y* at time 2, and reads from file *Z* at time 3, then the state of the file *X* at time 1 is assumed to be causal to the state of file *Y* at time 2, whereas no states of file *Z* can be causal to either file (Figure 7).

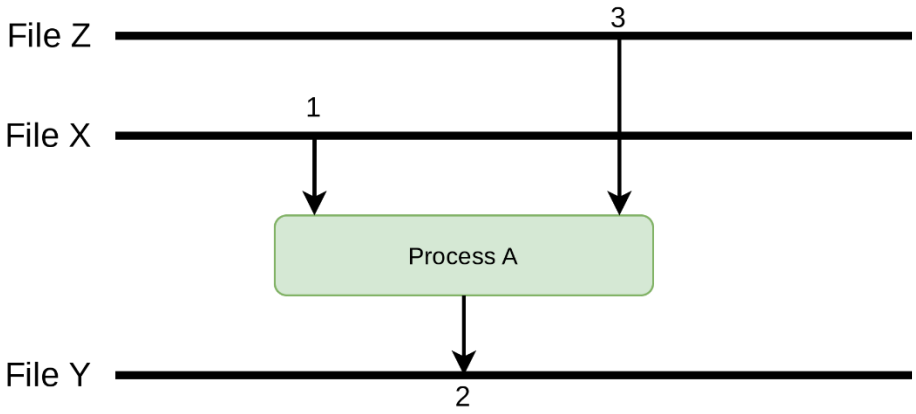


Figure 7: Transitive causality from read and writes

This transitive relationship may be formally expressed by representing processes and files as functions *P(t)* and *F(t)*, respectively. Argument *t* is a time instant, and the function's output is the state of the process or file at that time instant. Multiple processes and files are notated with subscripts, such as $P_a(t)$ and $F_x(t)$. Causality, represented by a single arrow operator, is assumed between earlier times and later times for the same process or file (Equation 1). Assuming error-free hardware, a file's state will not change over time unless altered by a process. It is also assumed that a process's state will be changed when it performs a read operation on a file. Given these two assumptions, a read or write infers causality for the same time stamp given a source and destination: process to file, file to process, or process to process in the case of interprocess communication (Equation 2). Transitivity between a read file state at an earlier time instant and a written file state at a later time instant when the same process is responsible for both operations can then by inferred (Equation 3).

$$P(t) \rightarrow P(t+1) \rightarrow P(t+2) \rightarrow \cdots \rightarrow P(t+n) \quad (1)$$

$$Read: F(t) \rightarrow P(t) \quad Write: P(t) \rightarrow F(t) \quad IPC: P_a(t) \rightarrow P_b(t) \quad (2)$$

$$F_x(1) \longrightarrow P_a(1) \wedge P_a(2) \longrightarrow F_y(2) \Rightarrow F_x(1) \longrightarrow F_y(2) \quad (3)$$

Given that the states of files can have causal relationships with each other due to a process reading and writing these files, complex provenance networks quickly arise. Each read operation made by a process causes it to be causally linked with the historical provenance of the file being read, and should this process then write to a file, the written file's state will then be causally linked to this provenance as well (Figure 8). A file is usually read at some point during its lifetime after being written. Additionally, processes often read and write multiple files, sometimes thousands or more. Together, these usage patterns cause the provenance network to quickly grow each time a new process is executed. For this reason, the method and design used by the implementation to store this data is critically important.

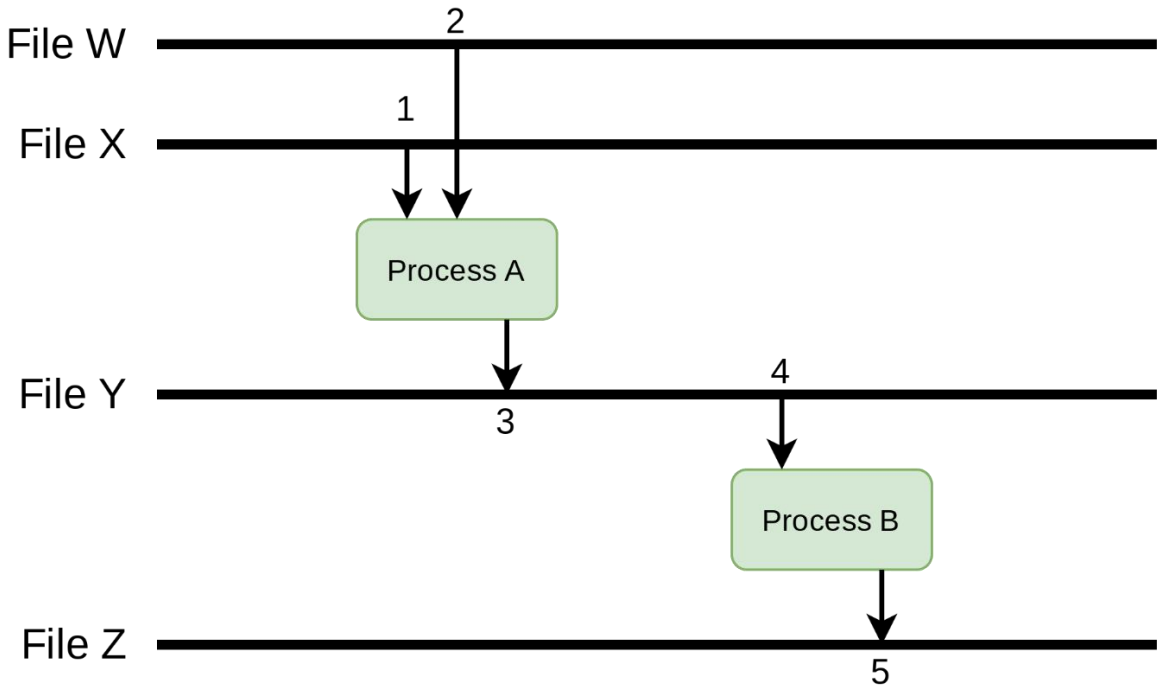


Figure 8: Provenance complexity due to multiple processes

3.2 Database Management System

Since causality may be viewed as a relationship between multiple entities within a temporal context, storage of this data requires a system capable of saving and querying relational and timestamp values. Although graph and NoSQL databases can represent the relationships between entities, it is difficult to assign a temporal context to these relationships. Given the simple provenance of a process that writes the same file twice, a graph database can represent the process and file as nodes and the write as a directed edge. However, many graph databases do not support multigraphs and therefore cannot store multiple edges in the same direction between two nodes. This necessitates merging the edges, thereby losing the order of the write operations. For graph databases that do support multigraphs, querying multiple edges typically involves iterating over a collection, which is inefficient compared to indexed lookups.

After evaluating these options, we decided to use a relational SQL database for storing provenance related information. SQL databases can represent entities as tables and relationships as matching primary and foreign keys (Section 3.3). Multiple rows can represent multiple operations, and each row can store a timestamp. Unlike graph databases, queries may be performed against an index, so lookups can be performed efficiently. RepeatFS can also reconstruct a directed graph using the SQL database (Section 3.4), so no features are lost for this gain in performance.

Though all major SQL database management engines support extensive functionality for working with this relational and timestamp data, SQLite (Hipp, 2020) was chosen for the implementation for a few reasons. Most importantly, as RepeatFS is an open-source Linux file system, distribution requires the RDBMS to share these characteristics, ruling out management systems with commercial or restrictive licensing, or those exclusive to other platforms, such as Oracle Database (*Oracle Database*, 2019) and Microsoft SQL Server (*Microsoft SQL Server*, 2019). Additionally, to achieve the goal of allowing the user to install RepeatFS without administrator privileges or complex installation instructions, only embedded management systems were considered. Though popular server-based systems such as MariaDB (*MariaDB Server*, 2021), MySQL (*MySQL Server*, 2021), and PostgreSQL (*PostgreSQL Server*, 2021) do have embedded versions, they do not have widespread support or stable libraries for Python and other languages.

Given these requirements, RepeatFS uses SQLite 3.x for storing and querying provenance data. SQLite is a serverless, embedded database management system with extensive and robust support across multiple programming languages, including Python. Additionally, though many Linux distributions install SQLite by default, the user can install it without administrator privileges if it is not present on the system. SQLite stores each database as a normal disk file, allowing the

user to easily move, delete, or maintain multiple copies of the file if desired. SQLite also provides many date and time related functions for comparisons, aggregation, and analysis.

3.3 Database Access and Structure

RepeatFS contains a database management module responsible for creating, querying, updating, locking, and otherwise maintaining the provenance database. This module contains the definitions of the database schema and related DDL commands, and automatically creates the SQLite database if not found when the user mounts a new instance of RepeatFS. It also creates the connection to the database and maintains all cursors associated with the connection, ensuring all database access is performed in a systematic and thread-safe manner. RepeatFS disables SQLite synchronous processing when establishing the initial connection, which drastically improves performance at the cost of potential data loss should there be a power loss or system crash. Since the user would likely need to repeat the analysis step being performed when a crash occurs, the provenance would be recaptured at this time. The database management module also contains an internal buffer for aggregating individual values that are written to the database in batch. The database consists of six tables and a variety of indices and constraints (Figure 9).

3.3.1 Database Table: Mount

The *mount* table is responsible for storing the directories that were targeted when creating a new instance of a RepeatFS mount. Its primary key is an artificial numeric identifier, and it stores the source (root) directory and mount directory specified by the user. These two directory fields are also uniquely indexed, ensuring that subsequent instances of RepeatFS targeting the same directories utilize the same mount record. Information in this table is required for replicating

provenance (Section 3.8), as paths must be converted between those starting with the RepeatFS mount directory and those starting with source directory on the underlying file system.

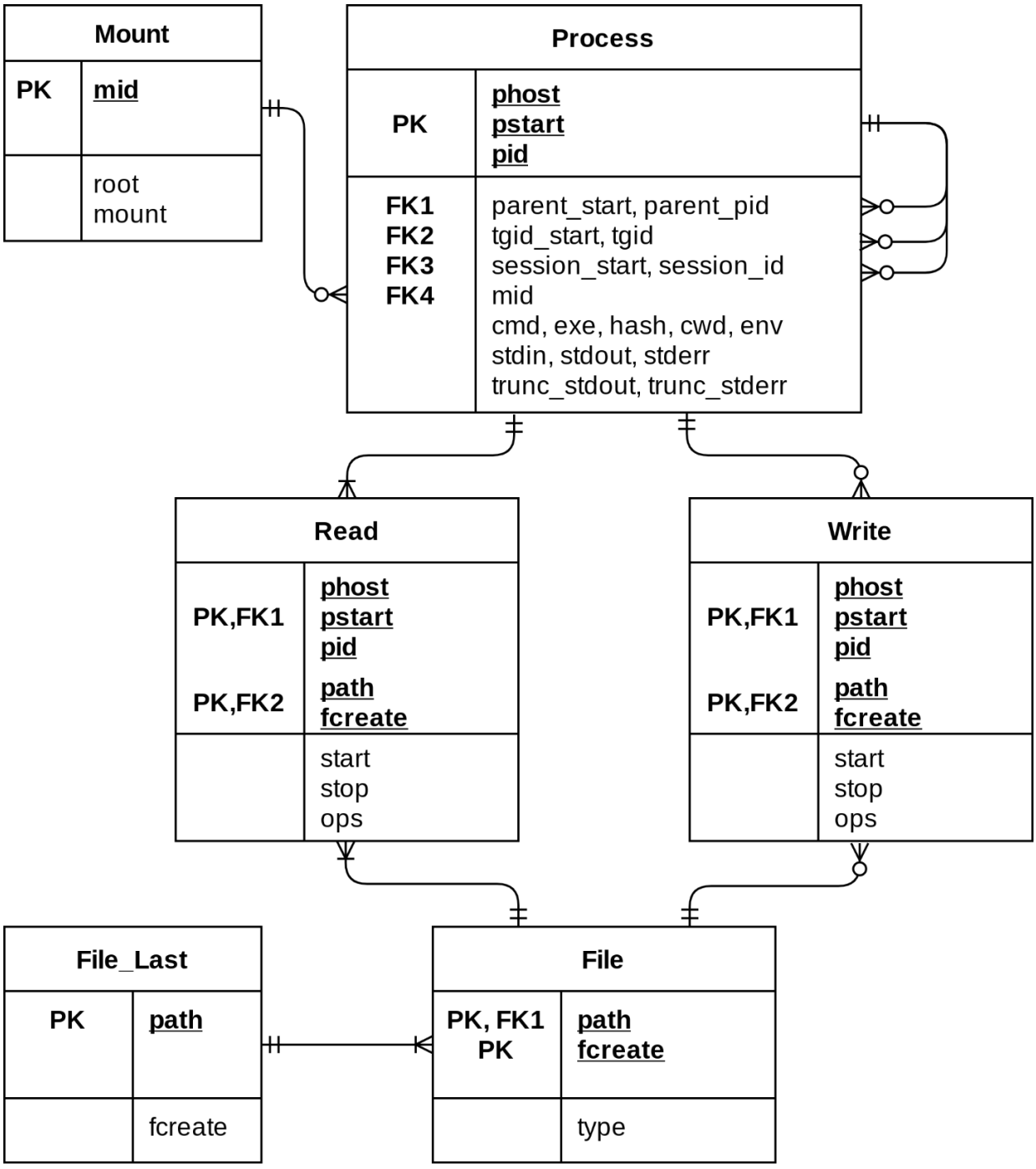


Figure 9: Database schema

3.3.2 Database Table: Process

The *process* table is responsible for storing information on any process that issues a system call targeting a file monitored by RepeatFS. This includes processes that perform an I/O operation, as well as other processes related to that process: ancestor processes that spawned the process, leaders of thread groups the process belongs to, and leaders of shell sessions the process was run within. Regardless of if the process was directly or indirectly involved, the *process* table stores information necessary to maintain a historical account of how the process was run and to replicate it with complete fidelity if requested.

The *process* table's primary key consists of three fields: the name of the system that executed the process, the timestamp when the process was spawned, and the process ID (PID). These are necessary for unambiguously identifying a process, since PIDs are recycled even on the same system and are not unique. Additionally, RepeatFS is designed with limited support for shared storage and potential future support for distributed storage, and it is possible for different processes on two different systems to have the same start time and PID, thus requiring the system name to be part of the composite primary key. Fields and constraints exist for parent processes, thread group leaders, and session leaders.

The remaining fields are necessary for replicating a process in an identical fashion to when it was first run. These includes the path to the binary that was executed, the MD5 hash of that binary, the command line parameters specified, the environment variables set at execution, the current working directory, and any redirection of the standard streams. The *process* table also maintains a foreign key to the mount table, allowing paths present in parameters and environment variables to be translated correctly between those beginning with the RepeatFS mount directory and those referring to the underlying file system.

Entries in the *process* table are created when the RepeatFS operation processing system receives any system call targeting a file system path, such as *mkdir*, *rename*, and *open*. System calls targeting a file descriptor such as *read* and *write* will have had their calling process registered during the initial *open* call and will not normally register or update information for the same process multiple times, with two exceptions. First, in the case of a parent process sharing its open descriptors with child processes, the child process is registered when it issues its first system call using that descriptor. Second, processes with stdout or stderr redirected are initially forked by the shell to create a new process. When a process is forked, the operating system associates the parent's process information with the newly spawned process. The shell then normally updates the spawned process's information to reflect the executed command, but before doing so, it creates the redirection file on disk using the new process's PID. This causes RepeatFS to record process information associated with the shell, such as bash, and not the executed binary. This gives the erroneous impression that all subsequent writes to this file were performed by the shell, when in fact they were performed by the program being run. To address this issue, RepeatFS refreshes the process information for the first N system calls made using a file descriptor, where N is less than a threshold amount. Eventually the shell will update the process information for a PID to reflect the executing binary, and the executing binary will write or flush stdout or stderr. This *write* call will occur within the first N operations, and RepeatFS will refresh the entry with the child process's correct information. We evaluated the maximum number of writes performed by multiple shells in a variety of scenarios and found 20 to be a sufficient threshold value.

RepeatFS registers and updates process information by parsing relevant details from the Linux procfs virtual filesystem mounted at */proc/pid* using files such as *status*, *cwd*, and *exe*. These files are created by the procfs kernel module automatically and universally for every forked

process, making them a good choice for obtaining this information in a reliable manner. These files must be processed immediately since they are deleted once the process terminates. RepeatFS also recursively records the process information for every parent process, ensuring the entire process tree is saved for every process performing file I/O.

Lastly, RepeatFS records processes that send or receive piped stdout or stderr data from a process that performs file I/O. Since piped processes often read from stdin and/or write to stdout and do not issue file system calls, RepeatFS would normally be unaware of them. To address this issue, RepeatFS checks during process registration if stdin, stdout, or stderr have been piped to a different process. If so, the piped process is also registered, and this registration recursively repeats for each pipe created by the user.

3.3.3 Database Table: File

The *file* table is responsible for storing information on any file or directory monitored by RepeatFS that is targeted by a system call. Its primary key consists of two fields: the absolute path to the file on the underlying file system, and the timestamp when the instance of the file was first recorded by RepeatFS. A file instance is defined as each file created with the same filename in the same directory. Since a file may be created, deleted, and then recreated using the same filename, the state of the first version of the file is not related to the state of the second. Therefore, RepeatFS uses the concept of file instances to isolate provenance for each version of a file at any particular pathname from other versions of the file. The *file* table also contains a field noting the type: either file or directory.

Entries in the *file* table are created when the RepeatFS operation processing system receives an *open* or *create* system call. RepeatFS only creates a new file entry for the first *open* call received

for a file instance; subsequent *open* calls for instances already present in the database are ignored. Conversely, new file entries are always registered when receiving a *create* call, as successful execution of this function indicates a new instance of the file has been created. If the file did not exist previously, the *create* call creates it, and if it already exists, the *create* call truncates it, effectively creating a new instance.

3.3.4 Database Table: File_Last

The *file_last* table is responsible for caching the latest instance of any file stored in the *file* table. Its primary key consists solely of the path to the file. Since there may be only one latest instance for any path, this is sufficient to uniquely identify it. The table also contains a field with the timestamp of when the latest instance was first recorded by RepeatFS. Entries in the *file_last* table are created or updated whenever a new record is added to the *file* table.

Two relationships exist between the *file_last* and *file* tables. There is a one-to-many relationship in which all previous instances of a file directly relate to the latest instance of that file; this allows RepeatFS to detect the latest instance for any given prior instance. There is also a one-to-one relationship in which the latest instance relates to the corresponding entry in the *file* table; this allows RepeatFS to lookup whether the latest instance of a path is a file or directory. While the latest instance could be determined from only the *file* table by sorting and filtering the instance timestamp, this information is cached in the *file_last* table to increase performance since RepeatFS requires referencing the latest instance of a file each time it processes a system call for that file.

3.3.5 Database Table: Read and Write

The *read* and *write* tables are responsible for storing information on system calls performed on a file monitored by RepeatFS. The tables are identical in schema and usage, with the exception that the *read* table stores I/O operations that make no alteration to the file system, while the *write* table stores those that modify the file system in some way. Both tables have a composite primary key consisting of the *process* and *file* table's primary keys. Since this only provides one unique record of each combination of process and the corresponding file it issues a system call for, each *read* or *write* record represents an aggregation of every operation invoked by a process for a particular file. The record contains timestamp fields representing the earliest and latest operation for each process and file interaction. To complement this, the record also contains a bit flag field representing each operation that occurs between the start and stop timestamps. Each bit in the flag is a Boolean value indicating whether the operation occurred at least once. Aggregating operations in this way provides increased performance over storing each operation individually when determining causality, and the exact time and count of each system call is rarely if ever required by researchers examining historical provenance.

The *read* and *write* tables are not directly updated during each system call; instead, RepeatFS maintains a corresponding structure in memory, noting both timestamps and the operations bit flag. This structure is created during a process's first *open* or *create* call of a file. These *open* or *create* calls return a file descriptor representing a handle to the open file. The structure's start time is set during the first system call using the file descriptor. The structure's end time is updated during every subsequent system call using the file descriptor. Whenever either timestamp is updated, RepeatFS performs a bitwise OR using the structure's current flag value and a constant value representing the operation type. The structure's flag value is then updated with

the result. When the descriptor is closed, the structure is then written to the database. Records already present in the database are updated whenever a process re-opens the same file; the database stop time is updated with the structure's stop time, and the database flag is updated by performing a bitwise OR using the database's current flag value and the structure's current flag value.

3.4 Directed Graph Construction

In order to visualize the provenance associated with a result file, a directed graph must be constructed based upon the process, file, and I/O events stored in the database. This construction methodology uses the causality inferred from each temporal event to create a graph in which nodes represent the state of a file or process, and edges connecting nodes depict a causal event between two states. An edge directed from a file to a process represents a read operation, an edge directed from a process to a file represents a write operation, and an edge directed from a process to a process represents a fork or other operation that performs interprocess communication.

RepeatFS stores graph data within a standard Python dictionary. Each type of object within the graph, such as file nodes, process nodes, read edges, and write edges, are organized within their own sub-dictionaries, and each sub-dictionary is a value within the graph dictionary. This allows RepeatFS to quickly iterate through all nodes or edges of a particular type by referencing the appropriate sub-dictionary. The keys within the sub-dictionaries match the primary keys used in the database. In this manner, all relationships are maintained since a foreign key can be used as a key to lookup a value in the appropriate sub-dictionary. Additionally, by flattening records retrieved from the database into a dictionary, this data can also be serialized into JSON or other comparable formats, providing a method of encapsulating provenance into a single, easily parsed file.

RepeatFS performs the directed graph construction and population of the dictionary using an iterative algorithm that begins with the result file (Figure 10). The latest instance of the result file is obtained from the *file_last* table for the requested file pathname, and a node is created for that instance (Figure 10: step 0). The *write* table is then queried for any events that involve the result file instance. The timestamp associated with the write is irrelevant at this step, since all operations already in the database must be in the past, and therefore all writes have a causal relationship to the current file state (Figure 6). For each write operation, RepeatFS looks up the process that performed the operation and adds these processes as nodes to the graph, with an edge directed from the process node to the file node (Figure 10: step 1).

For every process node added to the graph, RepeatFS queries the *read* table for any read operations made by the current process but filters the query to only include those read operations that happened prior to the write operation. This filter ensures only read operations that could have a causal relationship to the written file (Figure 7) are included. If a process reads from file B after it writes to file A, file B has no causal relationship to file A. For every read operation found in this manner, RepeatFS looks up the file instance that was read by the process and adds these files as nodes to the graph, with an edge directed from the file node to the process node, indicating a read (Figure 10: step 2a). RepeatFS also queries the *read* table in this same manner for each ancestor process in the current process's lineage (parent, grandparent, great-grandparent, etc). If any reads are found, nodes for each process in the lineage between the process performing the read and the current process are added to the graph, with directed edges indicating which parent process spawned which child process. The file instance node is then added in the same manner as listed above (Figure 10: step 2b).

For every file instance node, RepeatFS iteratively repeats the entire process again, treating the file instance node as the result file node. Though the write record's timestamp is ignored during the first iteration, subsequent iterations only include writes that happened prior to the previous read, as only those writes could have a causal relationship. RepeatFS continues this iterative process until no further read and/or write records are returned. Any file nodes without a write edge from a process are files that were created outside of RepeatFS. Any process nodes without a read edge from a file are processes that generated data without reading a file monitored by RepeatFS, such as processes downloading external data from a network, copying files from outside of a RepeatFS mount, reading input from non-disk devices such as keyboards, or algorithmically generating data without any input.

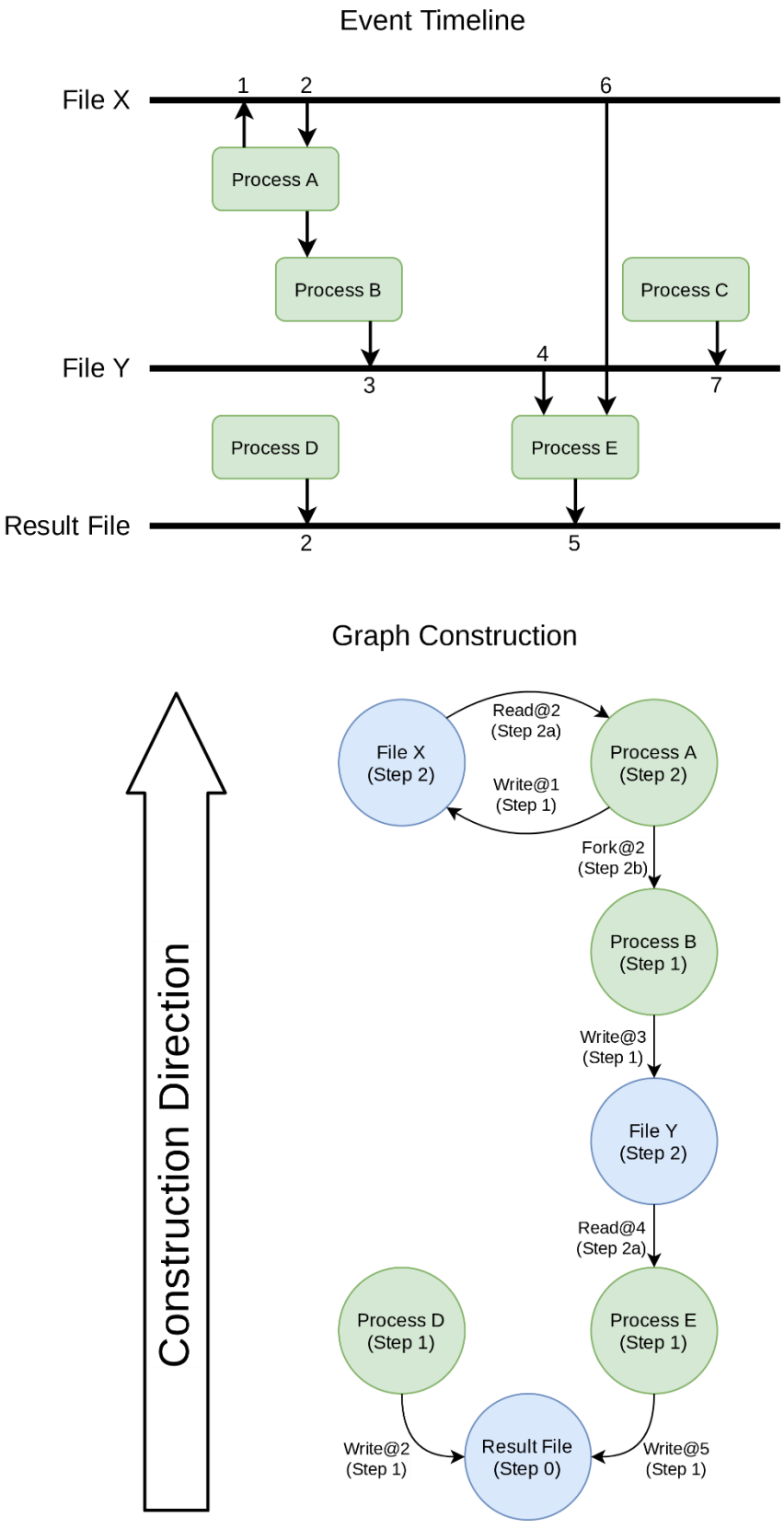


Figure 10: Directed graph construction from event timeline

3.5 Graph Contraction

Although constructing a directed graph as outlined above provides a complete model describing the provenance associated with a result file, there are two issues associated with this graph. First, for any non-trivial workflows involved in producing a result file, the provenance graphs are extremely large, complex, and difficult to follow, often containing thousands of files and processes related in intricate patterns. Many file nodes may be created by processes reading and writing temporary or interim files, and the process execution count may be iterative in nature and quickly escalate depending on the number of items processed. While a researcher may need to know that some number of temporary files were created, or that a specific process was called at least once, the names of these temporary files and exact process counts are often irrelevant.

Second, multi-process or multi-threaded programs often spawn children to perform specific tasks containing an I/O component. While the provenance graph represents this as two process nodes connected by an edge indicating the parent-child relationship, this is misleading to the researcher. A single process may be responsible for spawning hundreds of processes, but should the researcher wish to replicate the action, he or she would only run the parent process, not each child process individually. Processes executed directly by the user must be differentiated from those that were spawned automatically as children of these processes (section 3.6).

Both complexity issues may be remediated by contracting nodes in the graph. Because temporary or interim files are often created in batches for the same purposes, members of these groups tend to appear in the graph as file nodes all having the same number of read and write edges to the same process nodes. When this occurs, all edge information is the same for all nodes, and they may be combined into a single node representing a group of N files (Figure 11).

Complexity arising from child processes can also be addressed by contracting nodes. Since the researcher is typically concerned with the executed command and not subprocesses spawned by the program, all descendent process nodes under the user command process node may be combined into a single group process node. Unlike combining file groups in which all read and write edges are identical for each node, child process nodes may have different read and write edges from each other. To account for this, the union of all the child process edges are combined and reconnected to the new group node, and duplicate edges are eliminated (Figure 12). Process group and file group node contraction greatly simplify the provenance graph in scenarios involving many temporary files or child processes.

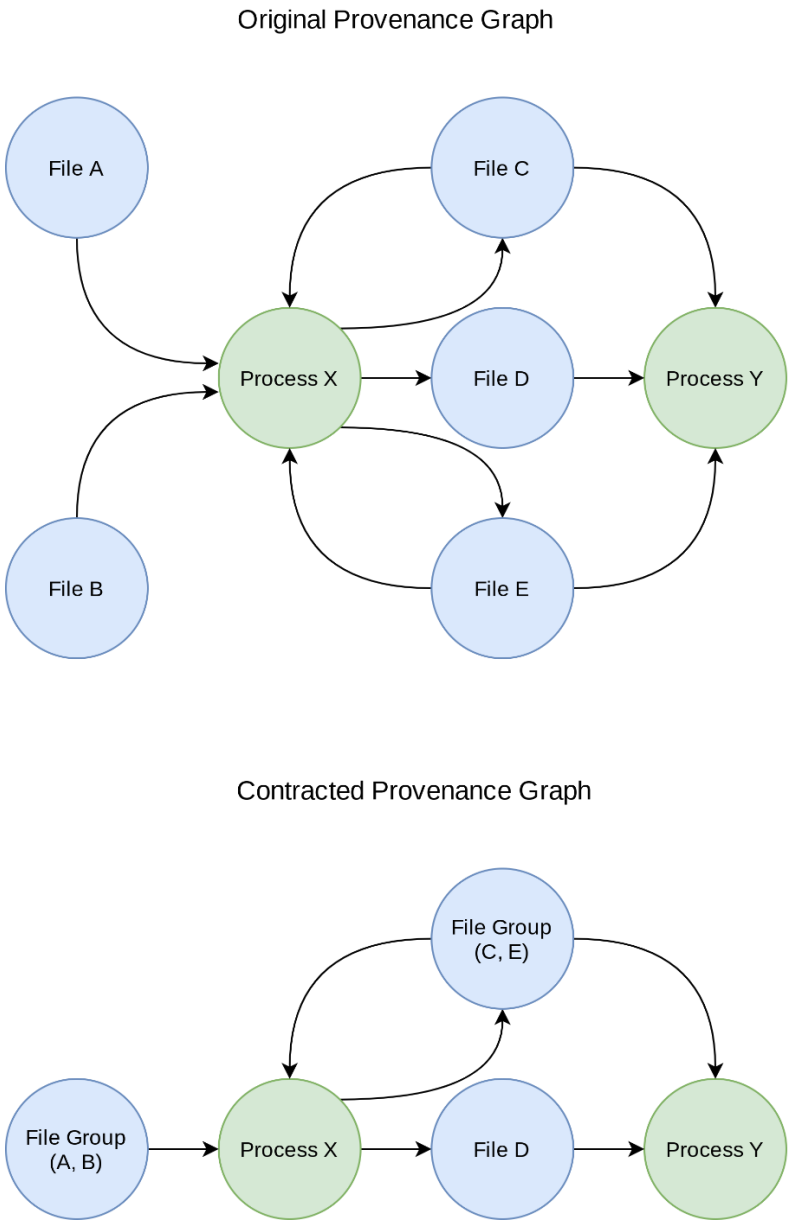


Figure 11: File group graph contraction

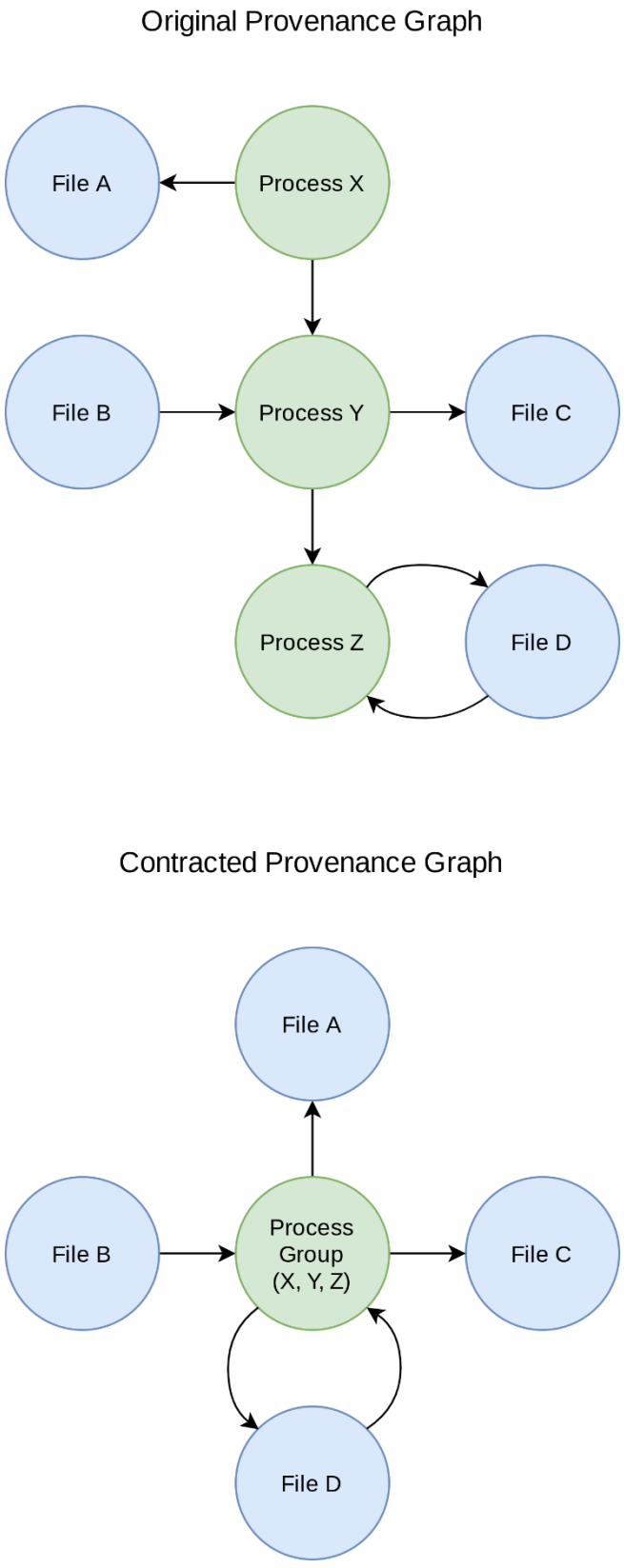


Figure 12: Process group graph contraction

3.6 User Process Identification

Differentiation of processes is not a trivial task. Every process in Linux has a parent process with a lineage that eventually traces back to the scheduler root process (Figure 13). Regardless of whether the process is directly run by a user or as a child process, it always traces back to the root. To differentiate these processes, tracing must account for shell processes. For typical researcher activity, a Linux shell spawns the user's processes, but child processes have a non-shell parent process. Although it is necessary to check if a process's parent is a shell, it is not sufficient; many shell processes are often the parent of processes not directly run by the user, such as shell processes executing shell scripts, or those running shell commands from other scripting languages, such as Python executing commands through the subprocess module. Checking if the shell is in interactive mode is not a reliable solution either, as there is no universal method of obtaining this information for all shells, and some commands directly issued by a user may not be part of an interactive session, such as those issued to HPC job schedulers. Calculating the number of generations from the root process also cannot be used for consistent identification, as shells may execute at different levels in the process tree. Additionally, for orphaned processes whose parents have been killed prematurely, Linux reassigns a parent process to the orphan, potentially at a different level in the process tree.

Instead, processes issued by the user may be differentiated from child processes by checking if their parents are shell session leaders. Linux marks shells attached to terminal or pseudo-terminal devices as session leaders. While users of modern Linux systems rarely connect to physical terminal devices, all X11 and SSH shells are connected to their own unique pseudo-terminal device, causing that shell to be marked as the session leader. Additionally, though job schedulers may not use a pseudo-terminal device unless requested, they typically mark the shell

executing the user's script as a session leader. All other shell processes, such subshells or those running scripts, are not marked as session leaders. Therefore, any process whose session leader PID matches their parent PID indicates the process was run directly by the user, while those whose PIDs do match were child processes.

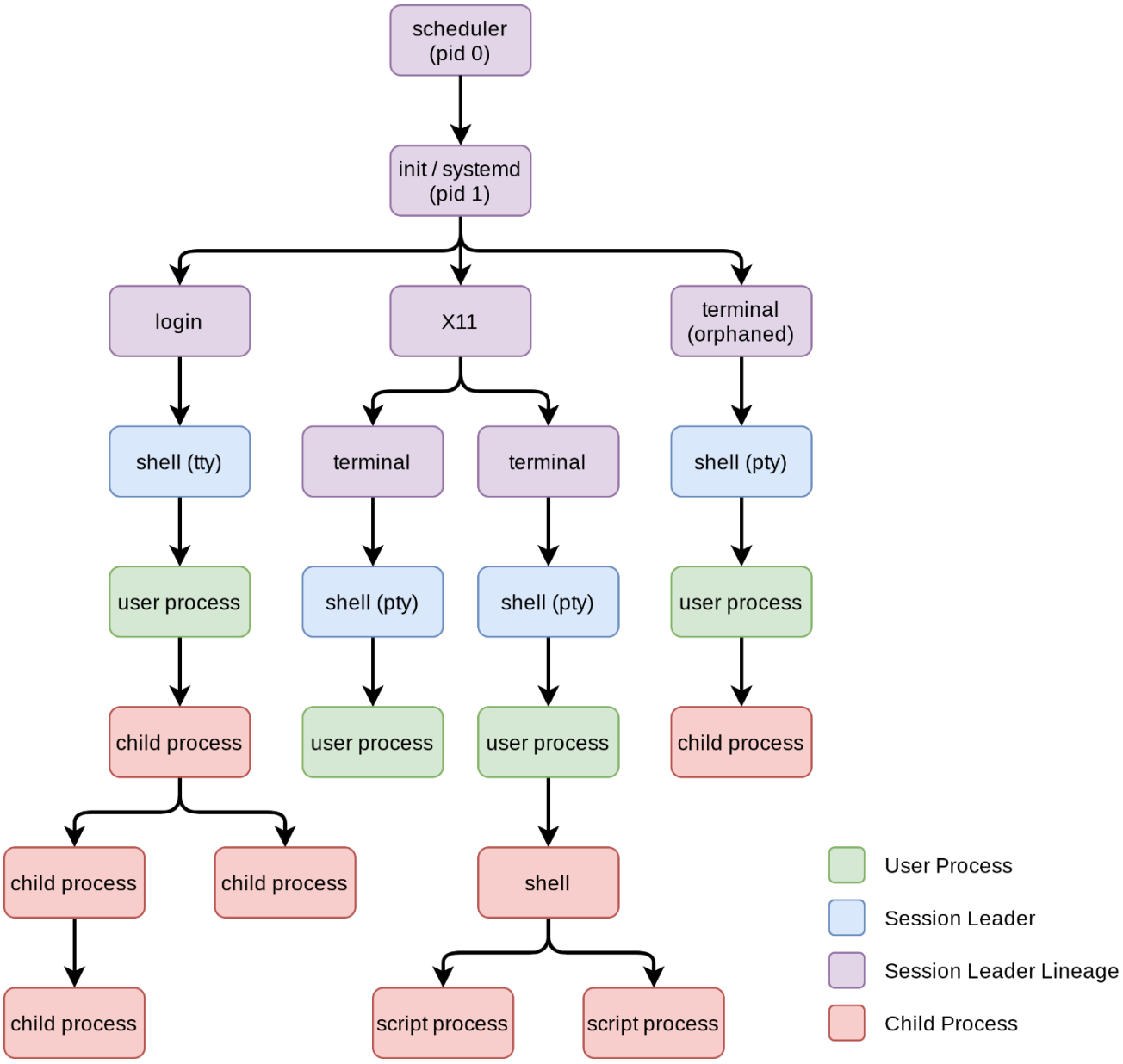


Figure 13: Example process tree

3.7 Provenance Visualization

A primary feature of RepeatFS is the ability to interactively visualize the provenance associated with a monitored file. This service is provided by visually depicting the directed provenance graph as an HTML/JavaScript document. The user can view this document in a standard web browser or distribute it to others as a single file. Process, file, and result file nodes appear as red, blue, and green nodes on the graph, respectively, while directed read and write edges are shown as black arrows. Parent processes spawning child processes are depicted with red arrows showing the direction of the relationship (Figure 14). Each node and edge can be clicked to view extended information or manipulate the graph.

Provenance HTML files are presented as VDFs (Chapter 4) associated with every file under the RepeatFS mount. When creating the HTML VDF, RepeatFS internally generates the directed graph using the provenance of the associated file. After the graph has been generated, process node contraction is performed to merge any child process nodes with their parent process nodes. Then file node contraction is performed on the contracted graph to potentially shrink the graph further. Contraction is performed in this order as grouping file nodes has no effect on process grouping, since process grouping is agnostic to file nodes. Conversely, grouping file nodes depends on the number of edges connected to each node, and the contraction is affected by the number of process nodes. By contracting process nodes first, fewer process nodes remain on the graph, and the probability of a file node grouping with another file node increases, thereby producing smaller graphs.

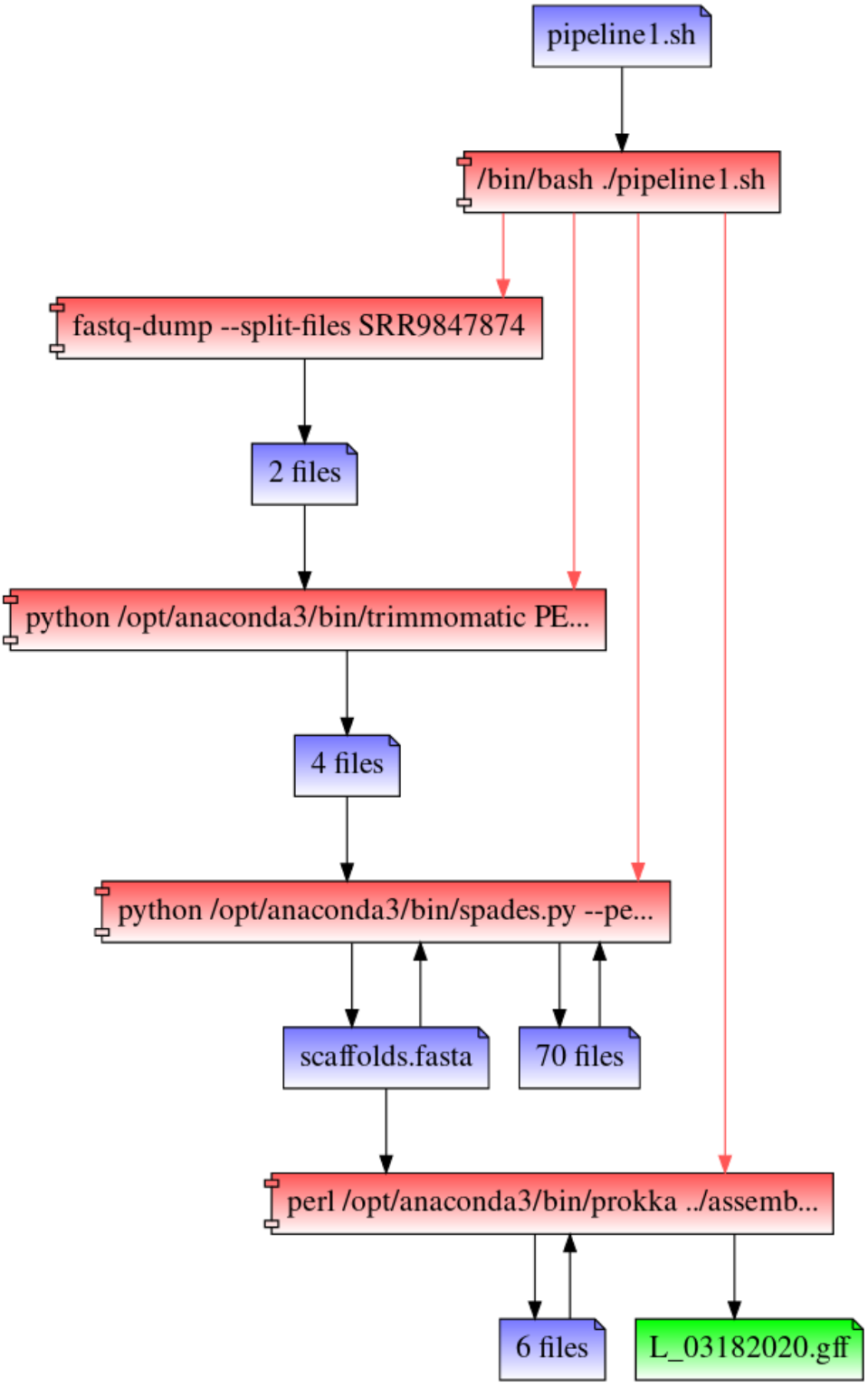


Figure 14: Example provenance visualization with expanded process

Once constructed, the contracted graph is used to configure and populate a Graphviz (*Graphviz*, 2021) object using the PyGraphviz (Hagberg, Schult, Renieris, & Millman, 2021) module. Graphviz processes the logical layout of the graph, calculating a reasonable and efficient physical layout for each node and edge. When complete, it encodes this physical layout into SVG (SVG Working Group, 2021), a scalable vector graphic XML (Hollenbeck, Rose, & Masinter, 2003) based format compatible with HTML. RepeatFS reads an HTML template and embeds the SVG code into body of the document, while also embedding a JSON representation of the original full provenance graph into the JavaScript portion of the document. The HTML template contains a layout for displaying the graph on the left side of the page, while displaying an information area anchored to the right-hand side of the page. It also embeds the necessary CSS to set font sizes, colors, and other relevant visual attributes.

The template also contains JavaScript functions designed to perform an action when any objects on the graph are clicked by the user. Selecting a node on the graph calls the JavaScript function, passing the ID of the node or edge as a parameter. The function then looks this ID up within the provenance structure derived from the embedded JSON and reads all attributes associated with the node, such as time of operation, file name, process ID, etc. It then manipulates the Document Object Model (DOM) of the HTML page to update the information box with the object's attributes. This allows the user to easily obtain details about any file, process, or I/O process by clicking on the object in the graph and viewing these details in the information box.

Though process node contraction groups all child processes into the parent process node on the graph, a user may wish to expand the group to view some or all of the subprocesses. This is especially relevant for pipeline and other shell scripts that run a series of programs as part of a workflow. Since the shell process running the script is the parent process, process contraction

results in only this process being shown on the graph, hiding each program the shell script executes. By expanding the shell process, the user can view each program executed by the script. RepeatFS provides this functionality through a button in the process information box in the web interface. When this button is clicked for a collapsed process group, RepeatFS makes an indirect call to the API file located in the same directory as the HTML VDF. The call must be made in indirect mode, as modern browsers do not have write access to the client file system for security purposes which is required for direct API calls. Instead, the visualization webpage encodes an API call requesting the expansion of the process node, encodes this as JSON, appends the JSON to the API filename, and then makes a read call to this filename in the same path as the HTML file. Upon receiving this indirect API call, RepeatFS recalculates the provenance graph, skipping contraction of the top level of the indicated process group. Child nodes with their own children continue to contract in the normal fashion. (Figure 15). The webpage then reloads itself, displaying the new provenance graph with the process group expanded. The “expand” button in the information box is now replaced with a “collapse” button, that performs the reverse operation.

In scenarios where the user copies the provenance HTML file to another location outside of RepeatFS, such as copying to removeable storage or sending to a colleague, the API file is not present to process commands and change process grouping. To accommodate this, the first time the webpage is loaded for a particular browser session, it generates a unique session ID and performs an indirect API call to instruct RepeatFS to encode this session ID into the provenance HTML. The webpage then reloads itself. If this session ID is present, API contact was successful, and the expand/collapse buttons are present in the information box. If the session ID remains unchanged, API contact failed, indicating the HTML file is not present under a RepeatFS mount, and was instead copied to another location. In this case, the information box notes the web interface

is in “snapshot” mode and modifications to process grouping cannot be performed. This allows the user sending the HTML file to select the desired grouping before copying the file, which is frozen upon making a copy of the file.

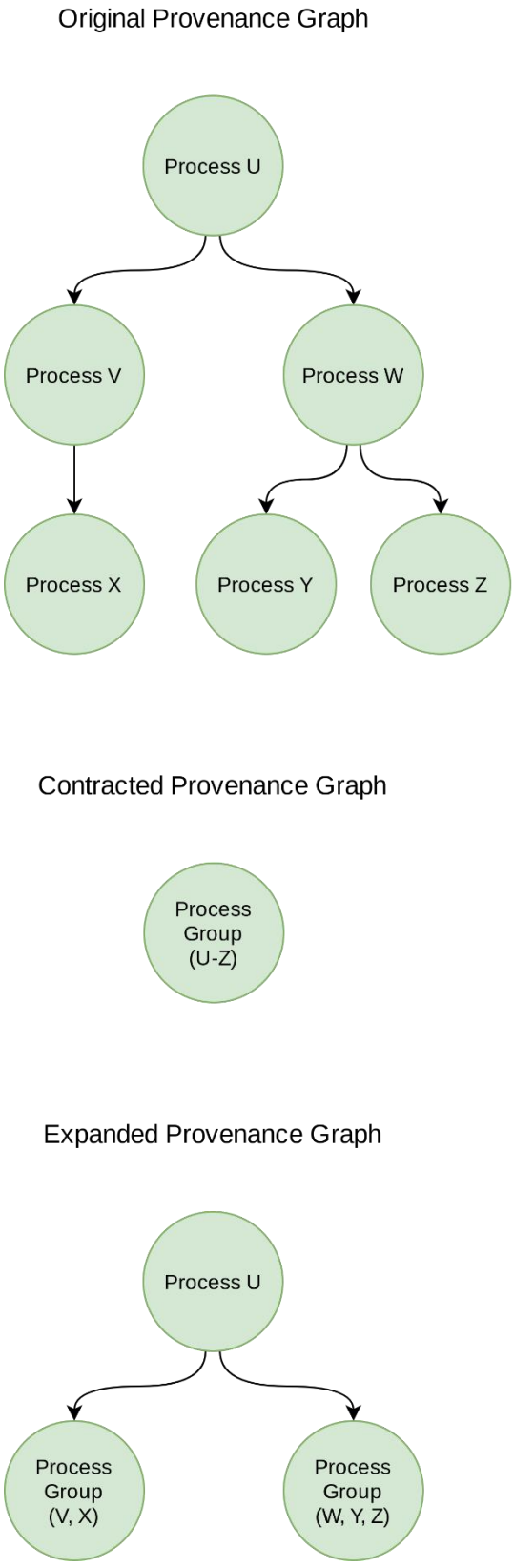


Figure 15: Process group expansion

3.8 Replication

In addition to visualizing provenance, another primary feature of RepeatFS is replicating the execution steps involved in producing a result file. Even for simple historical provenance, manually repeating each step obtained from the visualization graph is cumbersome and prone to error, and would require the user to repeatedly copy and paste commands, parameters, environment variables, etc. Instead, RepeatFS automates this process by performing each step in the correct order, eliminating errors introduced from copying commands.

Preparation for the replication process begins by obtaining a snapshot of a result file's historical provenance. Like the visualization HTML VDF, RepeatFS provides a provenance VDF for every file under the mount. When the user accesses this file, RepeatFS generates a full, non-contracted directed provenance graph using the same methods as outlined in section 3.7. The graph is not contracted as RepeatFS must be aware of every process for replication and verification; this would not be possible if any files or processes are removed due to merged nodes. Instead of processing this graph to create a visual representation, RepeatFS converts the graph into JSON formatted text, and provides this as the contents of the VDF. The user may replicate directly from this VDF at its original location or may copy the file to another location outside of RepeatFS.

To perform the replication, the user runs the RepeatFS client "replicate" command and provides the JSON provenance VDF as an argument. The client builds a JSON formatted API command to initiate the replication, embeds the JSON from the provenance file into the API command, and then sends it to RepeatFS via the API control file. The client then begins reading from the control file to process status messages of the replication process. It also receives the stdout and stderr streams obtained from each process executed during replication, and the client relays

these to the current terminal's stdout and stderr, or saves them to files when instructed by the user through command line arguments.

Upon receiving a replication request, the RepeatFS API system extracts processes from the graph by filtering for process nodes whose parent processes are shell session leaders (section 3.6). This ensures that only user commands are executed by RepeatFS during the replication process, and RepeatFS does not directly run child processes that would be run by their respective parent processes. An example would be a shell script that runs three programs – if RepeatFS did not filter out the child processes, during replication it would run the shell script and the three programs directly. Then the shell script would also execute the three programs, resulting in duplicate runs of each of the three programs. By filtering child processes out, RepeatFS only runs the shell script in the example scenario.

However, in cases where the original shell scripts are lost or not available during replication, RepeatFS provides a client command line argument to receive a list of process IDs to exclude from the user command list. For any process IDs specified by the user, RepeatFS does not execute them, nor does it filter their child processes. In the case of the above example, RepeatFS does not execute the shell script, and instead executes the three programs directly. This allows the user to reconstruct missing shell scripts by instructing RepeatFS to replicate their behavior. The user may specify multiple process IDs, including those of child processes, to include multiple scripts in the filtering process.

Once RepeatFS has assembled the full list of commands to execute, it checks the attributes associated with each process node for cases in which stdin, stdout, or stderr were redirected to other processes during the original execution run. These represent processes that make use of pipe redirection; processes with stdout redirected to another process are on the sending side of the pipe,

while processes with stdin redirected to another process are on the receiving side of the pipe. Our algorithm searches the provenance graph for each process that is the target of this redirection, ensuring it has record of it, then links the source and destination processes together. RepeatFS iteratively performs this linking algorithm for all processes. This results in command chains consisting of one or more commands connected by pipes, where the number of pipes is one less than the number of commands. RepeatFS sorts these command chains by their original execution timestamp of the first process in the chain, ensuring each group of programs is run in the correct relative order.

For each command in the chain, the current working directory, command line arguments and environment variables are parsed for substrings containing the original mount path of the RepeatFS instance that recorded the provenance. The mount portion of the path is removed, and the mount path of the current RepeatFS instance is substituted, ensuring the programs will run with paths that are valid on the system executing the replication. If stderr, stdout, or stderr were redirected from/to a file during the original run, these paths are also corrected in a similar way, and the same file redirection is configured for replication. Finally, each command in each group is executed in sorted order using the calculated current working directory, command line arguments, environment variables, pipes, and file redirection. RepeatFS records the new PID associated with each new process and maps it to the PID of the corresponding original process. Once all processes have completed, verification is performed (section 3.9) and the API system is notified that replication is complete.

During execution of each command, for those processes whose stdout or stderr have not been redirected to other processes or files, RepeatFS sends these output streams asynchronously back to the API system. The API system formats this as a JSON response and buffers it, ensuring

the next read request made to the API control file by the client receives the data. The client then displays the output on the screen or redirects it to a file, as directed by the user. The client continually retrieves and displays this data until the JSON response indicates the replication process has completed.

Although full replication and verification requires a RepeatFS instance on the replicating system, the RepeatFS client also contains an argument instructing the replication module to solely generate a list of commands it would run in a format that can be easily transformed into a shell script. During this process, RepeatFS performs the procedure as listed above, with the exception that it sends back the commands it would have run but skips execution and verification. This allows the original user to save the commands necessary to perform replication as a shell script and distribute it to others. Though this provides a less reliable method of replication, it is a useful feature since it allows users without access to RepeatFS to still perform the workflow in an automated fashion.

3.9 Verification

Following replication, RepeatFS verifies that the provenance associated with the processes and files recorded during replication matches the original provenance used to initiate replication. This is a significant reason why RepeatFS is required for replication and verification, and why this functionality cannot be reproduced by simply generating a shell script that executes the commands associated with a result file's provenance. The original and replication provenance graphs must be compared, and the replication provenance can only be obtained if an instance of RepeatFS is running on the replication system, thereby recording provenance during replication.

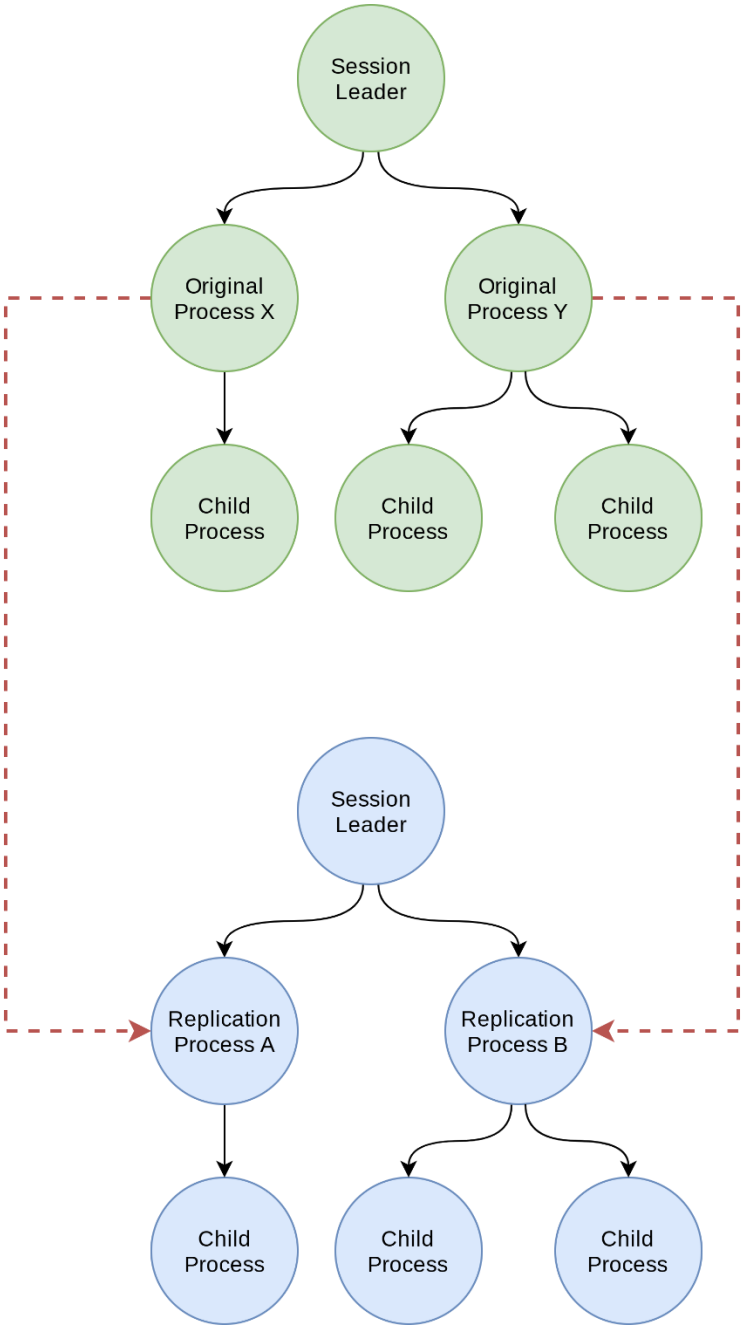
Comparison of the original and replication provenance graphs is currently confined to process nodes only, since the goal of verification is to ensure the programs run in the same manner while still allowing the replicating user to begin the process with different input files if desired. RepeatFS begins the comparison by iterating through each command in each command chain that was executed during replication and looks up the corresponding process node in the original provenance graph. It then uses the PID map generated during replication to find the corresponding replication process PID associated with the original process PID. Finally, it uses the replication process PID to look up the corresponding process node in the replication provenance graph (Figure 16).

Once the original and replication process nodes have been obtained, RepeatFS compares the MD5 hash field associated with these two nodes. This hash value represents the program's version. Since many applications use different methods to report their version information, such as in response to a particular command line argument, or as part of normal output, or not at all, there is no universal way of obtaining a program's version. Since the MD5 hash reflects the binary composition of the executable file, different versions of programs contain internal differences, and their MD5 hash values are different. Though this method does not indicate which is the newer version, it does indicate version mismatches. If RepeatFS encounters an MD5 hash mismatch between the original process and replication process, it relays a warning to the API system indicating the application on the replication system is a different version than the original.

The final step of verification confirms that the replication process spawns the same child processes as the original process. Failure to spawn the same number or type of processes indicates the replication differed or failed in some way. Although the original and replication process can be directly mapped using the PID that is returned when executing the replication process, child

processes must be matched using a different strategy. The Python subprocess module is responsible for executing each replicated user process and has access to the PID, but any child processes are spawned by these parent processes and not Python, and therefore the subprocess module does not have access to any child PIDs.

Instead, RepeatFS attempts to infer a match by ordering the child processes of the original process by execution timestamp. It then checks the corresponding replication process to confirm if it has the same number of child processes arranged in the same order with the same executable names. If these details match, RepeatFS adds to the existing process map, mapping each original child process to the corresponding replication child process in order (Figure 17). It then recursively repeats the verification process for each child process, checking executable versions and child processes. RepeatFS continues this until all process nodes in the original provenance graph have been verified, and finally notifies the API system that verification was successful. In the event the number or order of children differs (Figure 17: processes V, W, and C), it relays a warning to the API system indicating the replication differed in some manner. RepeatFS then terminates the verification process for the current branch of the provenance graph but continues for branches with confirmed matches. For processes that fail, the user may then check the stdout or stderr streams to investigate the cause of the deviation.



Process Map	
Original	Replication
X	A
Y	B

Figure 16: Replication parent process mapping

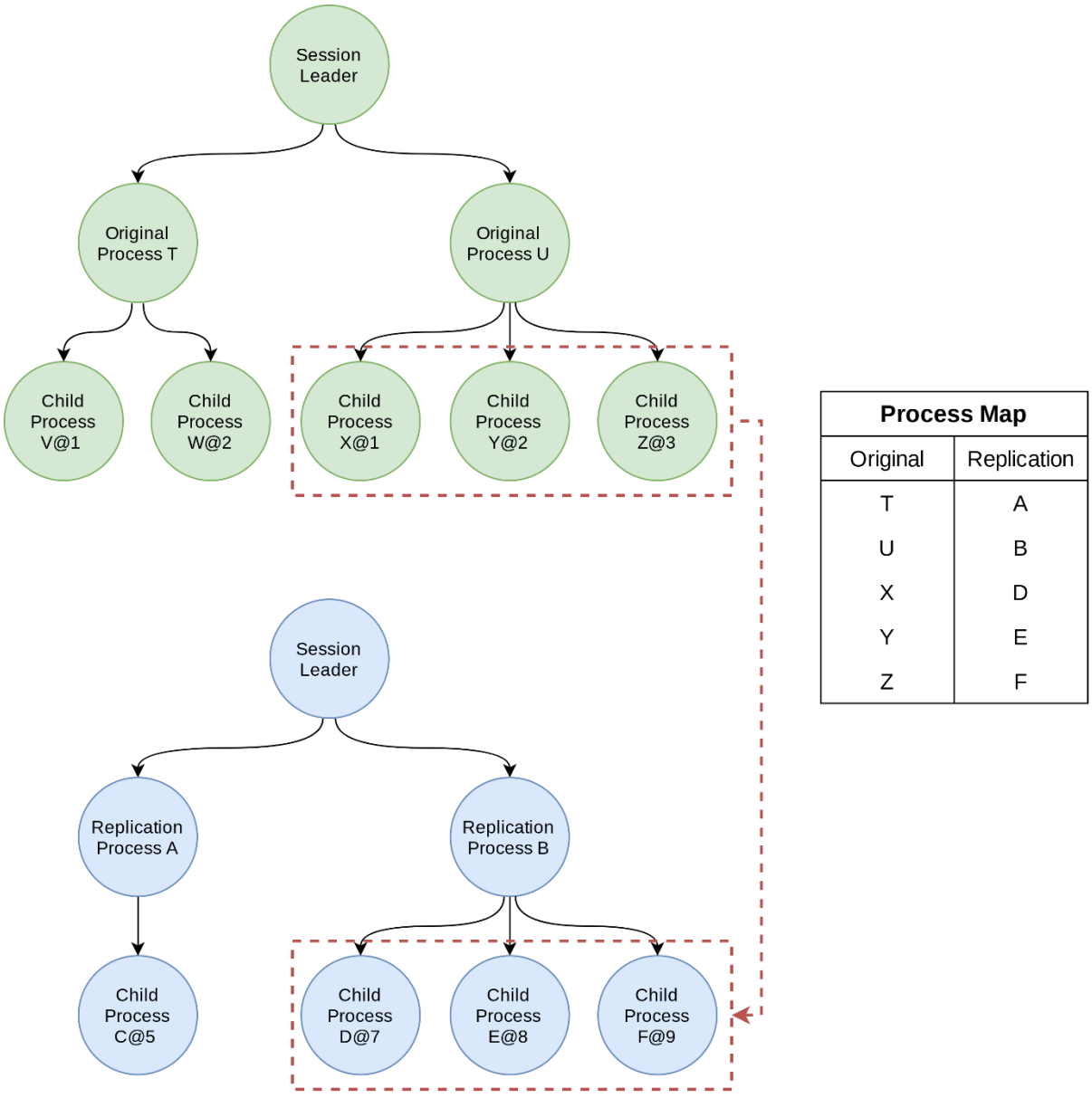


Figure 17: Replication child process mapping

CHAPTER 4

VIRTUAL DYNAMIC FILES

4.1 Introduction

The capability to record and process the provenance associated with result files fulfills the first requirement of the RepeatFS proactive model. Like the reactive model, it requires an automated mechanism to receive notifications from the kernel indicating a file system call has been issued. Although this is sufficient for the reactive model, the proactive model also requires awareness of the actions prior to execution. RepeatFS VDFs provide this awareness by allowing the user to predefine a list of commands associated with a particular task. When the user wishes to perform that task and obtain the output of the commands, he or she can read the contents of the VDF, causing the commands to be executed and their output to be returned to the user. Since VDFs are accessed from within a RepeatFS instance, all IO operations performed by the commands associated with the VDF are recorded in the normal manner, ensuring provenance is still visible to the user.

VDF functionality fulfills the conceptual requirements of the proactive model. However, practical considerations must be taken into account to ensure VDFs are viable in typical use cases. Computational analysis often does not generate data from scratch, and instead requires one or more files as input to perform calculations desired by the user. Therefore, every VDF is associated with a particular "source file", or file that is used as input by the commands that VDF executes when accessed. To choose which VDFs are generated for any source files, the user provides VDF definitions (Table 5) that each contain a regular expression pattern. RepeatFS attempts to match this regular expression against the source file name; if the match succeeds, a VDF is created. Since

files are typically named using file extensions or prefixes that identify their file type, users can incorporate these naming conventions into their regular expressions. This allows the user to create VDF definitions for specific types of files, such as a definition for converting PNG source files to PDF format (Figure 18). Multiple VDF definitions may have regular expressions that match the same file name, and therefore a single source file may have multiple VDFs associated with it. For instance, a JPEG source file may have two definitions: one for converting to GIF format and another for converting to PNG format (Figure 18). Each VDF definition also contains the Linux commands needed to perform the operation; these definitions are stored within the configuration system.

Associating VDFs with source files also provides a method for organizing the location of VDFs on the file system. For every file whose name matches the regular expression within a VDF definition, a hidden directory in the same directory as the source file, known as the "VDF Directory", is recognized as a valid path by RepeatFS. The VDF Directory has the same name as the source file, appended with a preselected special character (Figure 18: "file1.fa+"). RepeatFS does not create this directory on disk, nor does it appear in directory listings unless configured to do so by the user. However, since RepeatFS considers this path valid, the user may select it as their working directory or direct any applications to make use of the path. RepeatFS lists all VDFs within the VDF directory whose definitions match the source file. All VDFs have the same name as the source file, appended with a suffix configured by the user in the VDF definition. Since VDFs are treated as normal files, they may also be source files for other VDFs. In this way, multiple VDFs may be chained together by accessing the appropriate path (Figure 18: "file2.jpg.png.pdf").

Efficiency and performance are also practical considerations. Unlike normal files whose contents already exist on disk or in memory, VDFs must execute software to generate the output

that is returned to the system call accessing the VDF. Depending on the particular code being executed, the time involved in this request is often many orders of magnitude greater than simply retrieving data from storage. Repeatedly accessing the same file, which is often required during the analysis process, magnifies this problem, as the same set of commands would be executed each time the VDF is accessed. Since each subsequent run provides identical output to the first, only the initial execution is required. Additionally, if multiple processes accessed the same file concurrently, each would attempt to write its output to the VDF, creating the potential for conflicting or corrupted data.

To address these problems, RepeatFS provides a caching system to store VDF contents, called the Block Caching System (BCS, Section 4.2). When the commands associated with a VDF run and return output, this data is saved into the RepeatFS BCS. This ensures that repeated or concurrent system calls reading from the same VDF access the BCS, and RepeatFS will not execute the VDF commands a second time. Concurrent access to the BCS is also prioritized for typical sequential access, ensuring high performance for multithreaded applications (Section 4.4). Although the Linux page cache could provide some of these features, as a generic service it is not able to provide the tight coupling RepeatFS requires to interface with the Process Output Handler (POH, Section 4.5). Therefore, the page cache is disabled for any system calls made to a VDF, allowing RepeatFS full control over initial and repeated IO requests.

Not only do VDFs provide an implementation of the RepeatFS proactive model and a method for users to easily execute commonly performed tasks, they also serve as a versatile replacement for POSIX unnamed and named pipes (FIFOs). Pipes are a commonly utilized mechanism for processing large amounts of data. Pipes allow the output from one program to be streamed into the next program. Any waiting output from the first program is immediately

available for reading by the second program, and should a second program read when no data is available, the pipe causes the second program to block until more data arrives. This allows two or more programs to concurrently process different portions of the data from the same input file. Using pipes, the total time to process the entirety of the data is approximately the runtime of the longest running process. This is more efficient than saving each program's output to a file in its entirety before sequentially executing the next application in the pipeline. Without pipes, the total time to process the entirety of the data is the sum of each process's runtime (Figure 19).

Though pipes are advantageous for streaming data from one process to another, they have limitations. Pipes redirect the stdout device of one process into the stdin device of a second process. This restricts usage to applications that read/write input/output to the standard stream devices: stdin, stdout, and stderr. This precludes many applications that directly access files on disk. Though FIFOs overcome this by providing a named pipe at a file system location, they require the user to manually create each instance to be used in an operation, which is time consuming and litters the file system with temporary FIFOs. Modern shells and the Linux procfs also provide special mechanisms and paths for treating unnamed pipes as normal files, but these are often unique to each shell and syntactically unwieldy.

Another significant drawback of the pipe design is that only a single endpoint may receive the output data from a pipe. Software workflows involving two or more programs reading the same input data concurrently cannot utilize a pipe, since only one of the programs reading the input can be connected to the receiving end of the pipe at any one time. Though endpoints may swap control of the pipe, this is not sufficient, since data is deleted from the pipe once received by the endpoint. Although a second endpoint could take control of the pipe, the data read by the first endpoint would no longer be available to read.

Normal disk files do not suffer from this issue and do allow multiple programs to concurrently read the file. But disk files have the following limitation: if a process tries to read past the end of file (EOF), the calling process is not blocked, and the read request immediately returns no data to the calling process. This is problematic as two scenarios can cause a read request to return no data, and each scenario must be handled differently in a normal pipeline workflow. In the first scenario, a first process with an open file descriptor is actively appending data to a file being read by a second process, but the second process reads the data faster than the first process writes it. In the second scenario, no processes are writing to a file and a process attempts to read more data than is present in the file. In the first scenario, the reading process should continue read attempts since more data will eventually be present within the file, while in the second scenario, the reading process should cease attempting to read from the file. However, since the file itself normally contains no information to signal to the reading process whether any other processes are currently writing to it, the reading process is unable to differentiate between these two scenarios.

We designed VDFs solve the limitations of both pipes and disk files. Since RepeatFS presents VDFs as normal files on the file system, multiple processes may concurrently read from them, solving the limitation of pipes. Unlike files, VDFs are aware of processes that are actively writing to them; the Process Output Handler (POH) causes EOF reads to a VDF to block while a writing process is still populating the contents of a VDF, or the POH returns immediately if all writing processes have completed and the file is finalized. Reads to a VDF that return no data unambiguously indicate to the reading process that no further data is available, solving the limitation of files. VDFs combine the benefits of pipes and files with none of their drawbacks; they not only fulfill the requirements of the proactive model, but also allow for efficient branching pipeline workflows that are otherwise not possible.

VDF Definitions		
Task	Regex	VDF Ext
Filter sequences	\.fa$	filtered
JPEG to GIF	\.jpe?g$	gif
JPEG to PNG	\.jpe?g$	png
PNG to PDF	\.png$	pdf

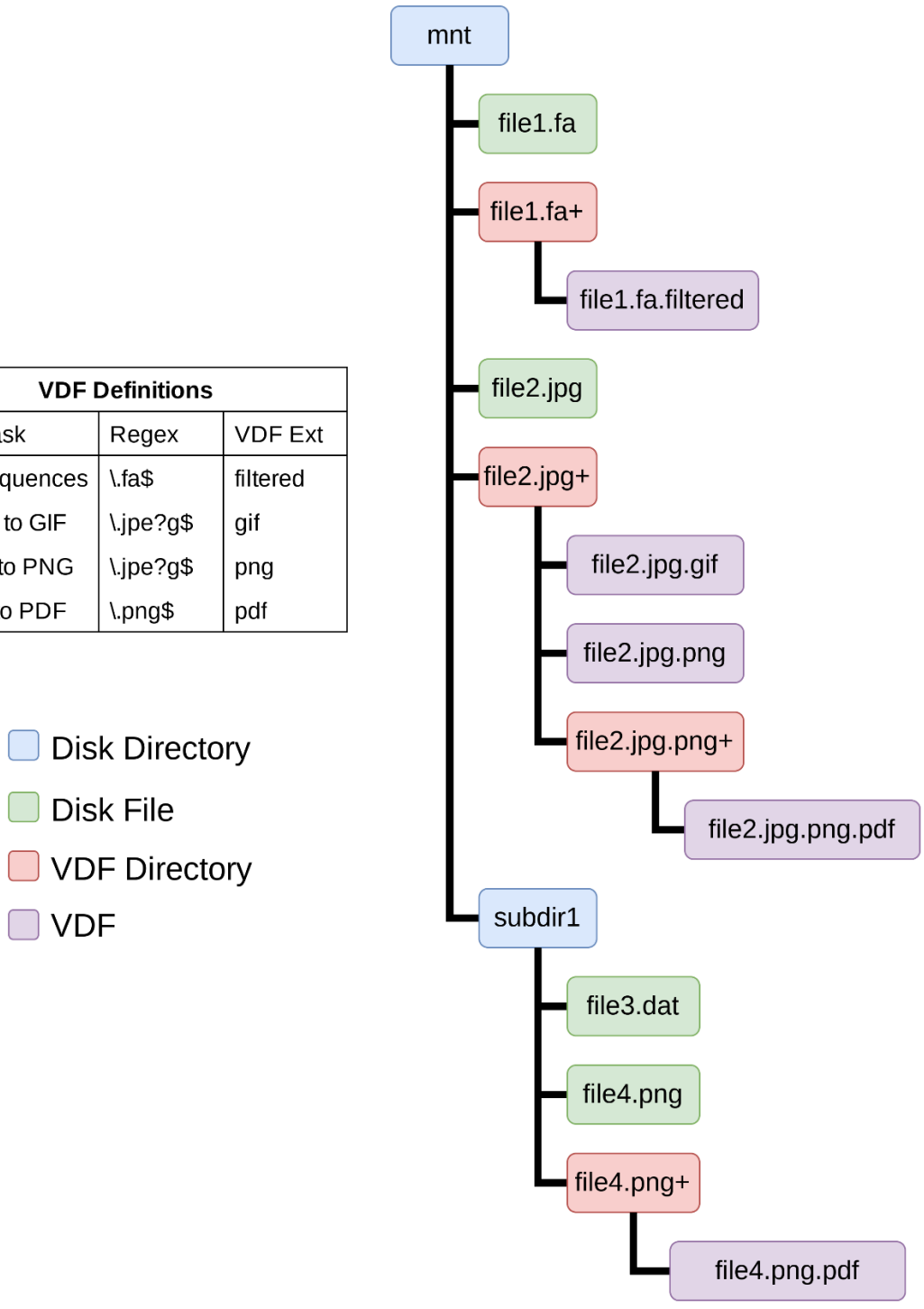


Figure 18: VDF path organization

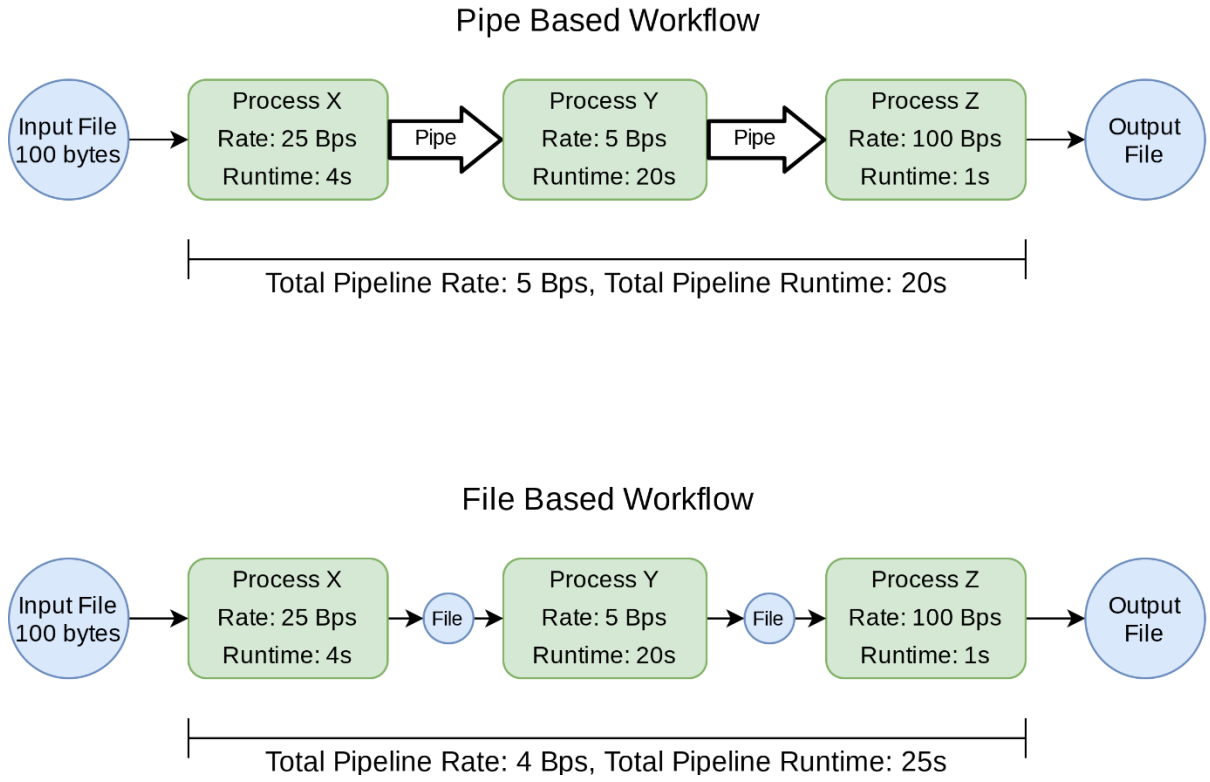


Figure 19: Pipe based vs file based workflows

4.2 Block Caching System

The BCS consists of data structures and routines for saving, retrieving, and maintaining block data and metadata associated with each VDF that has been previously accessed. The system contains both a memory-based and disk-based cache (Section 4.3), and a prioritized locking system for concurrent operations (Section 4.4). For any system calls involving paths that target VDFs, RepeatFS first checks if the operation is allowed. Since VDFs are normally read-only, *rename*, *unlink*, *chmod*, or other modification system requests are denied, though certain *write* and *truncate* requests are allowed. Permitted requests, such as *open* and *read*, are routed to the appropriate method within the BCS. Upon receiving an *open* request, the BCS checks if the target path is

present within a list of VDF entries stored in the cache. If the VDF path is not present, the BCS creates the entry, and generates the new file descriptor, associates the descriptor with the VDF entry, and returns the numeric ID associated with the descriptor to the calling process.

I/O calls involving a file descriptor must already be present within the cache, so the VDF entry table is not checked for *read*, *write*, and *truncate* calls. Upon receiving one of these operations, the BCS signals the POH to begin execution of the commands associated with the VDF, and the POH immediately returns to the BCS. The BCS then calculates the block locations given the requested file position and number of bytes. For *read* calls, the BCS checks if the requested blocks are present within the VDF entry, and if the last requested block has enough bytes to fulfill the entire *read* request. If both requirements are fulfilled, these blocks are retrieved, the requested bytes are extracted from the blocks, and this data is returned to the calling process. If any of the requested blocks are not in the cache, or if the final block does not contain the required number of bytes to fulfill the *read* request, the BCS checks the POH to see if the commands associated with this VDF have completed running and the VDF has been marked as finalized. If the VDF is finalized, the POH provides no further data, and therefore the VDF is fully populated. In this case, the BCS retrieves as many bytes as it can from the block cache to fulfill the read request and returns the data to the calling process. If the VDF is not yet finalized, the BCS blocks the calling process until it can retrieve additional data from the POH, adds these blocks to the cache, and fulfills the read request. If the VDF is finalized before the number of requested bytes are generated as output, only the available data is returned.

For *write* and *truncate* requests, the BCS relays the request to the POH. The POH may process the request or deny the operation (section 4.5). Denied operations are immediately returned to the calling process, and no further processing is performed by the BCS. For operations that are

accepted, the POH processes the entirety or a portion of the request. If the POH only processes a portion of the request, it returns the number of bytes it did not handle. The BCS calculates the block locations given the position and number of bytes of the *write* or *truncate* request and writes the unhandled data directly to the block cache. Finally, the BCS returns the number of bytes handled by the operation to the calling process.

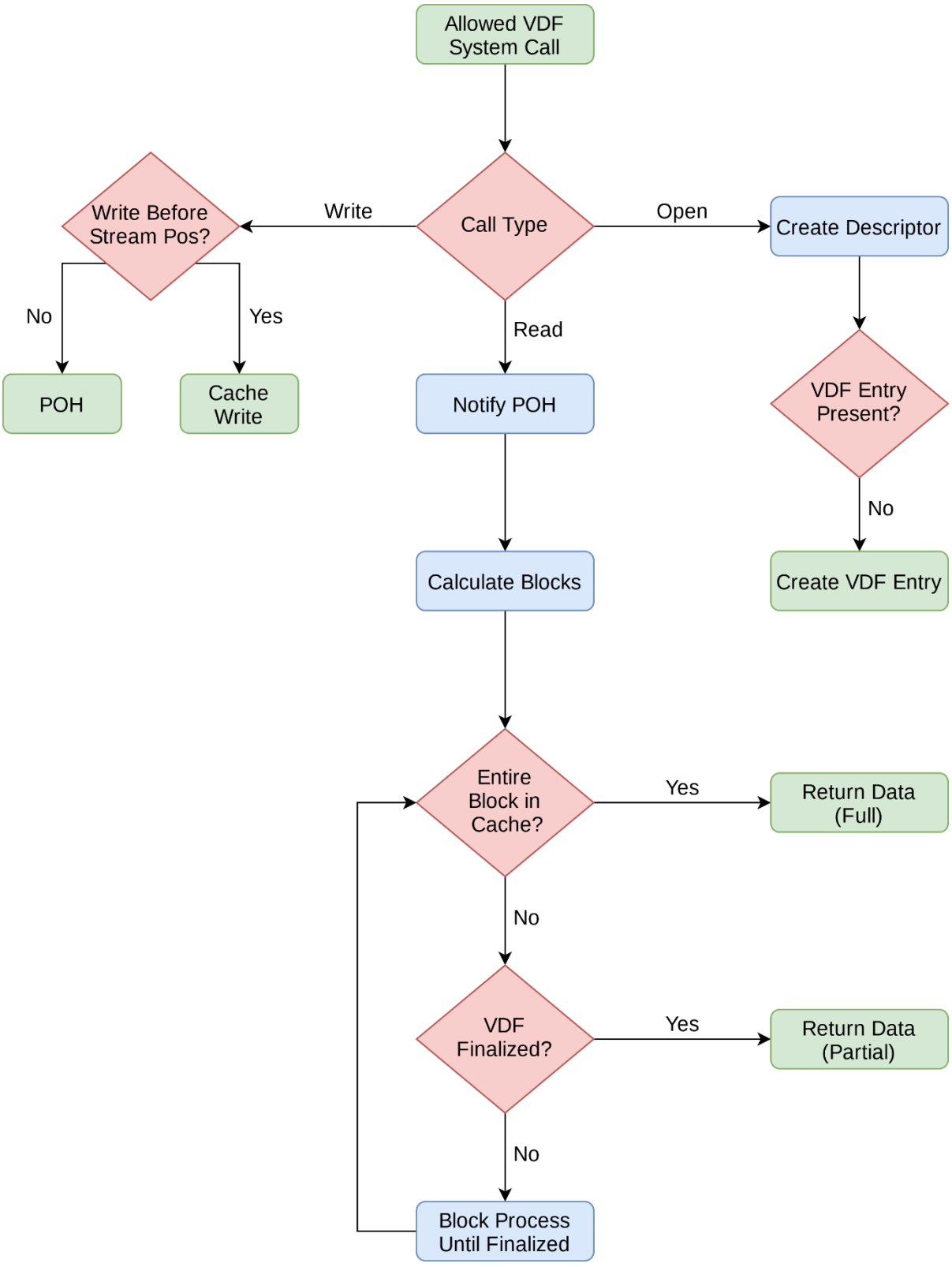


Figure 20: Block Caching System workflow

4.3 BCS Cache Handling

Two sets of caches are maintained for each VDF entry stored in the BCS. As outlined in section 4.2, these caches contain the contents of the VDF; they store the actual data returned by the commands associated with the VDF. Both caches are organized as uniformly sized, sequentially ordered blocks. The default block size is defined by the configuration system and may be overridden by the user in the configuration file. The primary cache is memory-based and directly accessed during BCS operations. Since VDFs may quickly grow dependent on the task being performed, this memory cache may easily consume the available system memory. To remediate this issue, the BCS also maintains a secondary, disk-base cache for each VDF. This cache provides another location for block storage to alleviate memory consumption.

For each write operation processed directly by the BCS, or indirectly by the POH, the BCS first checks if the current total memory cache size is greater than a percentage threshold defined in the configuration system. If the cache size does not exceed this threshold, the data is saved to the block in the memory cache, and the metadata record attached to the block is updated. This metadata contains a dirty flag indicating that the block contains data not yet written to the disk cache; it is set to true during each write operation. The BCS also adds the block index to a history queue, giving an ordered record of when each block of every VDF was updated. This queue allows the least recently used (LRU) blocks to later be flushed if necessary.

If the current memory cache size does exceed the configured threshold, the memory cache is flushed to disk prior to the write operation being performed. The BCS iteratively pops block indices from the history queue and checks the dirty flag of each referenced block. Blocks with unset dirty flags are immediately removed from the memory cache; if the dirty flag has not been set, it indicates no new writes have happened since last loading the block from the disk cache, and

the two caches are coherent for this block. If the dirty flag is set, it indicates the memory cache has newer data than the disk cache, and the block is flushed to the disk. The BCS calculates a hash based on the VDF path and creates and/or opens a file by this name in a preconfigured directory. The BCS then saves the block to the same location within the file relative to the block's location in the VDF. After the block has been flushed to disk, it is then removed from the memory cache.

For each read operation processed by the BCS, it first attempts to retrieve the block from the memory cache. If the block is present, it is immediately returned. If a memory cache miss occurs, the BCS attempts to locate the block in the disk cache. If present, it copies the block from disk to the memory cache, the dirty flag attached to the block is set to false, and the data is retrieved from the memory cache. If a disk cache miss occurs, the block is not yet present in the BCS. In this case, the BCS checks if the VDF is finalized to decide whether to block the calling process in order to wait for further data from the POH.

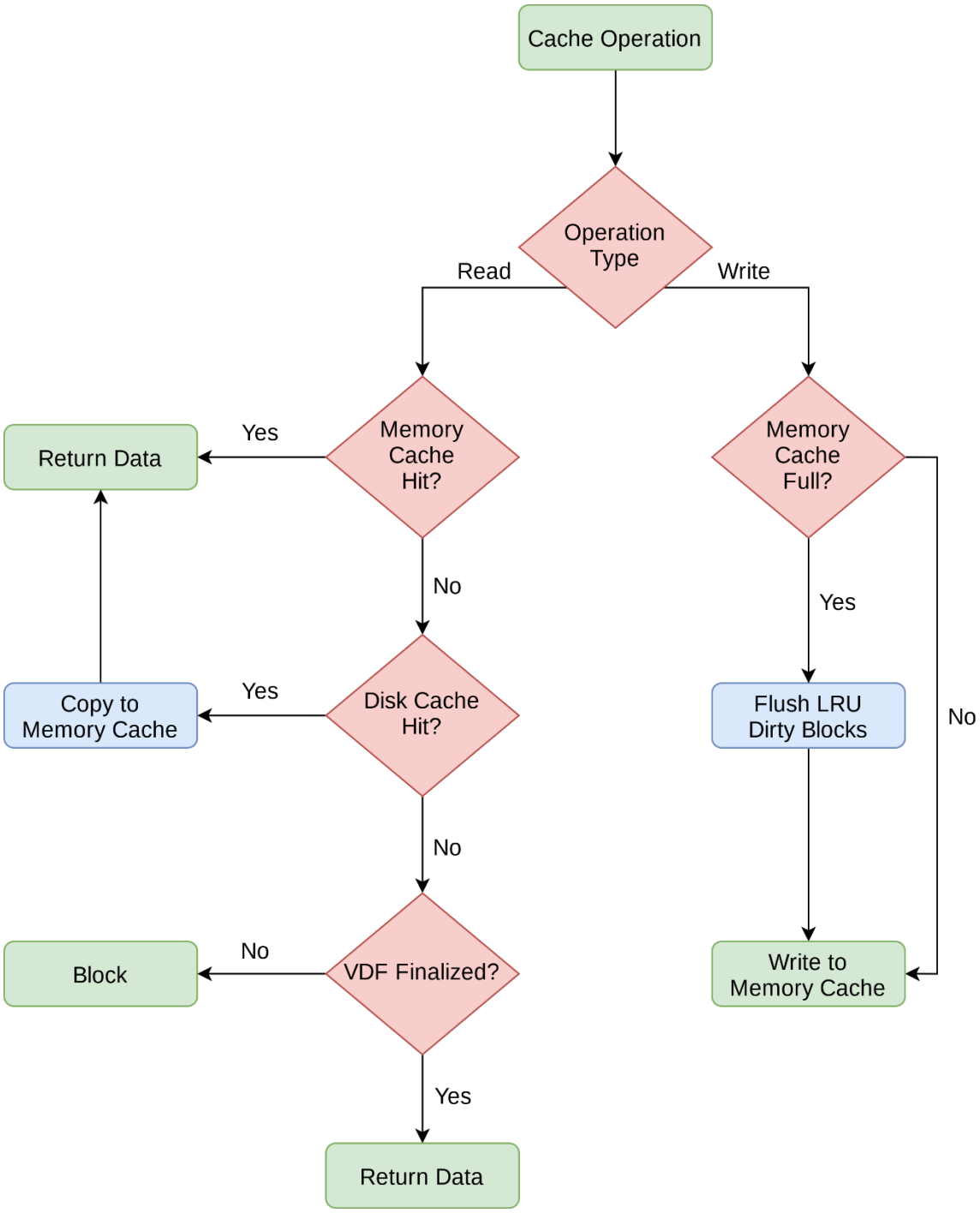


Figure 21: BCS cache handling workflow

4.4 BCS Prioritization

Prior to processing a *read*, *write*, or *truncate* system call performed on a VDF, the BCS will send the request into a prioritized waiting queue. The purpose of this queue is to improve performance for multiple operations accessing the same VDF concurrently. Failure to prioritize operations can lead to thrashing between the memory and disk cache during concurrent read operations. For example, if a VDF contains more data than can fit into memory, a portion of it must be stored in the disk cache at any one time. Given two concurrent processes reading from the same VDF, the first process may read from blocks at the beginning of the file, and the second process may read from blocks at the end. As the blocks at the end of the file are likely newer, the blocks from the beginning of the file have likely already been flushed from the memory cache and are only available on disk. When the BCS services the first process, it will flush the blocks at the end of the VDF to disk in order to load the blocks from the start of the VDF from disk. Then when the BCS services the second process, it will perform the inverse operation, flushing the blocks from the start of the VDF to disk, and loading the blocks from the end of the VDF from disk. This repeated process of flushing and loading, or thrashing, will continue until either process has finished, thereby incurring significant performance penalties. The situation is further complicated if a process is currently writing to the VDF, or additional processes are reading from other locations in the VDF.

To avoid this, concurrent read operations are prioritized by the index of the block being read. This strategy assumes that most reads are being performed in sequential order and will eventually access the entire file. While this may not be the case, it is a common scenario in informatics processing. Requests to read blocks earlier in the VDF will be serviced before requests to read blocks later in the VDF. While this may cause an initial flush of the memory cache if

another process is reading from later portions of the VDF, eventually the process reading from earlier in the VDF will catch up to the same block index being read by the second process (Figure 22). Once the two processes are actively reading from the same blocks concurrently, the risk of one process causing a cache flush that affects the other process is eliminated. Additionally, prioritization is subjected to a timeout threshold. Processes reading earlier portions of the file will only maintain priority as long as they are actively performing read operations, regardless of if they hold an open file descriptor to the VDF. Should a process not perform a subsequent read within the timeout threshold, priority will return to the next earliest read in the VDF.

Write operations are always given the highest priority to avoid delays associated with temporary starvation in typical informatics processing. For example, given a read and write process concurrently accessing a VDF, a process performing reads often does so at a faster rate than the process performing writes, since each write operation is often associated with a time-consuming computational task. In this situation, there is a higher probability a read will be active when the write call is issued. If writes were given a lower probability than reads, this would often cause the write to be blocked. Then the read would eventually reach the end of available data, would block, and the write process would unblock and write the next block of data. This pattern of both processes blocking would repeat until either process has finished, incurring performance penalties due to heavy context switching. Instead, by giving writes the highest priority, they are never blocked. This strategy reduces the number of context switches since only reads are blocked. It also increases the availability of data for the read process, reducing the overall number of times the read process will be blocked.

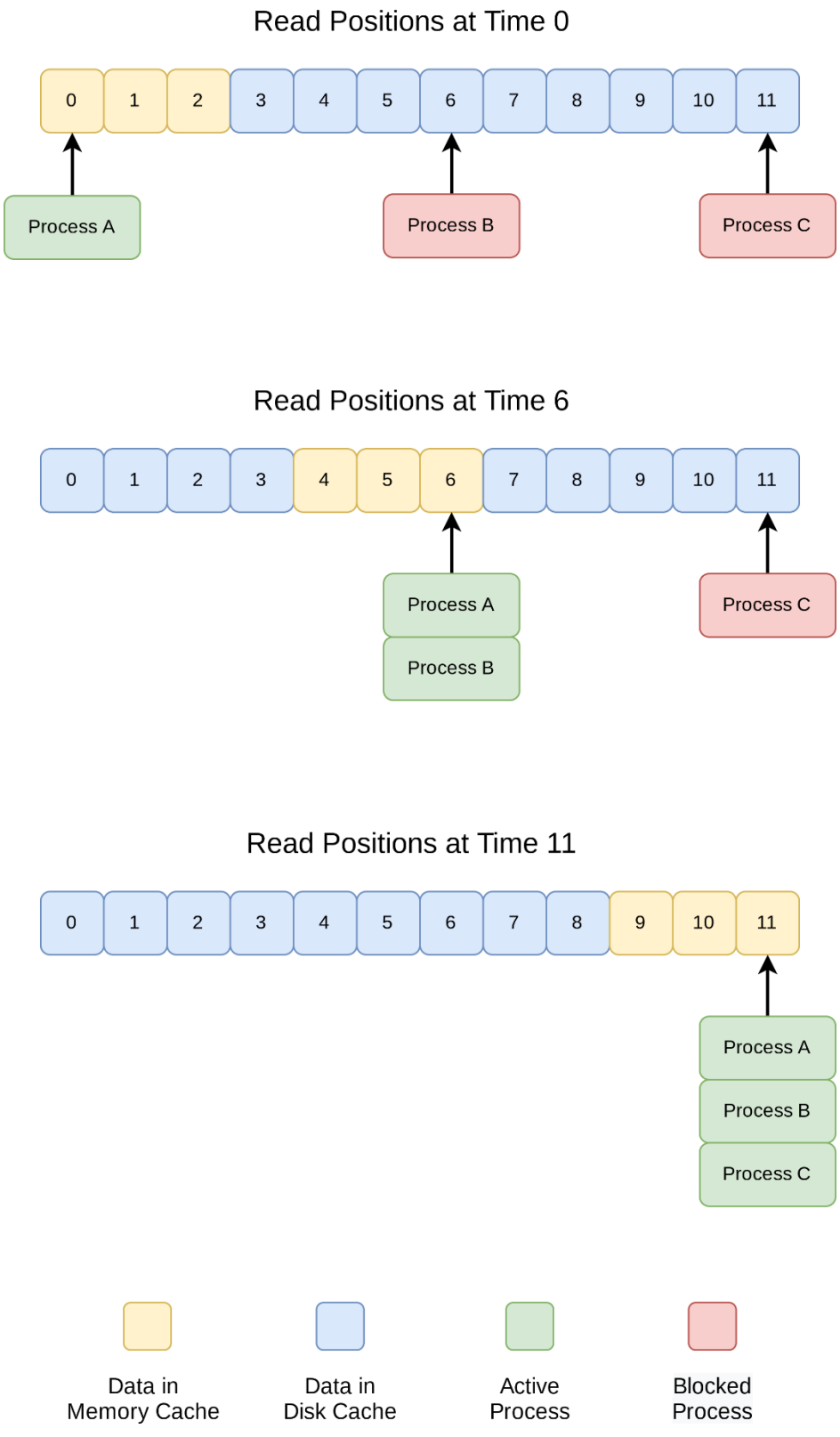


Figure 22: Concurrent VDF sequential read prioritization

4.5 Process Output Handler

The POH is responsible for initiating, monitoring, and potentially terminating the processes associated with a VDF. It also asynchronously receives the output of these processes and makes them available to the BCS. The POH contains two buffering systems for receiving output: a File I/O Buffer (FIOB) for processes that provide output by writing to disk files, and a Stream I/O Buffer (SIOB) for processes that provide output by writing to stdout or stderr. These buffers are unified into a single, generic interface the BCS uses to request data from the POH. Each buffering system contains functionality for blocking process writes or BCS reads when necessary.

The BCS requests execution of the commands associated with a VDF through the POH. Upon receiving an execution initiation request, the POH first verifies the commands have not been run previously by checking if the commands are actively running, or if the VDF is finalized. If either of these conditions are true, the POH ignores the initiation request and returns control to the BCS. If neither condition is true, the POH reads information related to the VDF from the configuration system, including the source file, the string of the commands to execute, and whether the output is provided via file I/O or standard streams. The execution commands string contains placeholders specified by the user that indicate where the name of the input file and/or output file should be filled in. The POH replaces these placeholders with the path to the source file and/or VDF, respectively. It then calls functions in the subprocess Python module to spawn these concurrent processes in their own shell session and immediately returns control to the POH, noting the parent shell PIDs of the spawned processes. For applications that provide output via stdout or stderr streams, the POH redirects these two devices to the POH stream buffering system. Finally, it returns control to the BCS.

4.6 File I/O Buffer

The POH FIOB provides a mechanism for asynchronously receiving data from processes that provide their output by writing to disk files. All writes received by RepeatFS that target VDFs are sent to this buffering system. Before processing the write request, the FIOB first verifies the *write/truncate* call was issued by a process spawned by the POH by confirming the PIDs match. This prevents other applications from erroneously or maliciously writing data to the VDF and corrupting its contents. Since a parent process may share open file descriptors with child processes, the FIOB will check the entire PID lineage for matches. In the event the PIDs do not match, RepeatFS returns the appropriate FUSE exception that indicates an access-denied error. If the PIDs do match, the FIOB also verifies the VDF is not finalized. This prevents processes with PIDs reused by the operating system from writing to the VDF. If the VDF is finalized, this indicates the processes spawned by the POH have already terminated, and RepeatFS returns the access-denied FUSE exception.

Once the calling process has been fully verified and the *write/truncate* call is accepted, the FIOB calculates the portion of the write request's data that should be stored in the FIOB's buffer, and which portion should be written immediately to the BCS; this demarcation is determined by the write position. Whenever a BCS cache miss occurs (Figure 23: reads targeting yellow/green blocks in BCS), it retrieves waiting data from the FIOB. Since the BCS fetches data sequentially, it cannot make a second request for positions it has already retrieved. The last position fetched by the BCS is saved as the "BCS write position" (Figure 23: "BCS write pos"). Write calls to positions before the BCS write position must be sent directly to the BCS since the BCS will only request data from the FIOB for positions after the BCS write position (Figure 23: "Immediate Write to BCS").

Any portion of the data being written at or after the BCS write position is handled by the FIOB (Figure 23: “Write Buffered in FIOB”). If enough free buffer space remains, it will write the entirety of the request to the buffer and return to the calling process. However, if the *write/truncate* call requests writing more data than the buffer can accommodate, the FIOB will write until the buffer is filled, but will block the calling process until the buffer has been retrieved and emptied by the BCS via the POH unified buffer interface. Each time data is read from the FIOB, all waiting write processes are awoken, allowing them to write more data. Related, each time data is written to the FIOB, all waiting read processes are awoken, allowing them to read more data. This mutual process continues until all data has been written and read, and control is returned to the calling process.

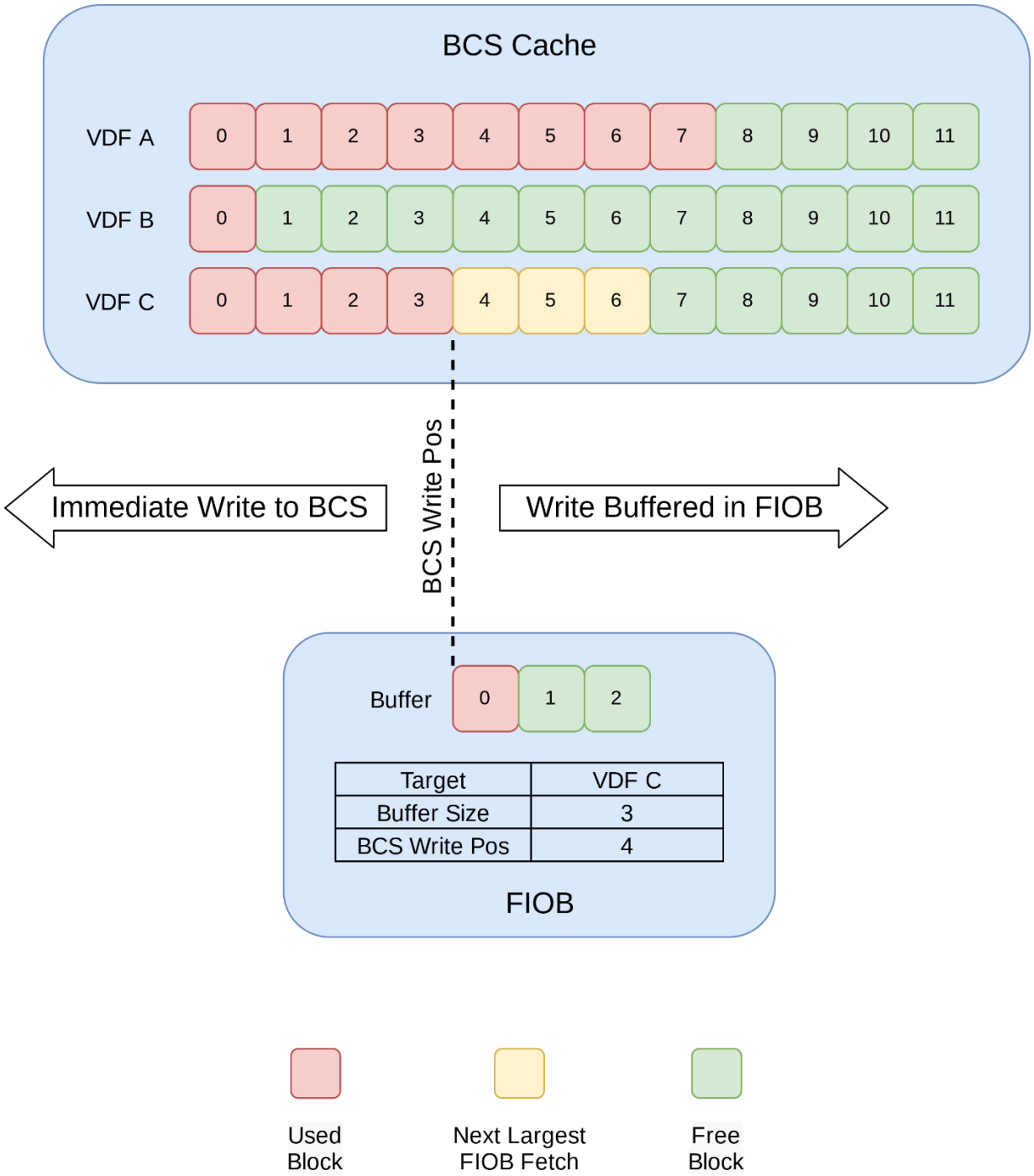


Figure 23: VDF write destination for file-based output

4.7 Stream I/O Buffer

The POH SIOB provides a mechanism for asynchronously receiving data from processes that provide their output by writing to stdout or stderr. When these types of processes are spawned by the subprocess module, the stdout and stderr devices are disconnected from the terminal and redirected to an instance of Python's standard BufferedReader class. Since writes to standard streams do not involve accessing normal files on the file system, it is not possible for other processes to unexpectedly issue writes. For this reason, the SIOB does not perform a PID check to confirm the calling process. Additionally, if the standard stream is still active, this indicates the process is still running and the VDF is not finalized, so the SIOB skips this confirmation step.

Like the FIOB's internal buffer, the SIOB's BufferedReader implements similar blocking behavior. If the BCS attempts to read from an empty buffer, the SIOB's Buffered Reader will cause the calling process to block until more data is available in the buffer. If the process associated with the VDF attempts to write data to a standard stream, the BufferedReader will verify enough free buffer remains to fulfill the entire write request. If the space is available, the data is written to the buffer, and the SIOB returns control to the calling process. However, if the request can only be partially fulfilled, the BufferedReader will write data until the buffer is filled, and then block the writing process until the buffer is emptied. As with the FIOB, the SIOB's BufferedReader will wake any blocked reading processes when a write is made and will wake any blocked writing processes when a read is made.

Unlike the FIOB which accepts write requests that may target any position within a file, including positions before the BCS write position, the SIOB only accepts sequential data from standard streams. There is no way for processes to write to earlier positions, they can only append data (Figure 24: "Streams Buffered in SIOB"). For this reason, the SIOB contains no functionality

for directly sending data to the BCS (Figure 24: “Streams Only Append”). All data received by the SIOB is buffered using its BufferedReader instance.

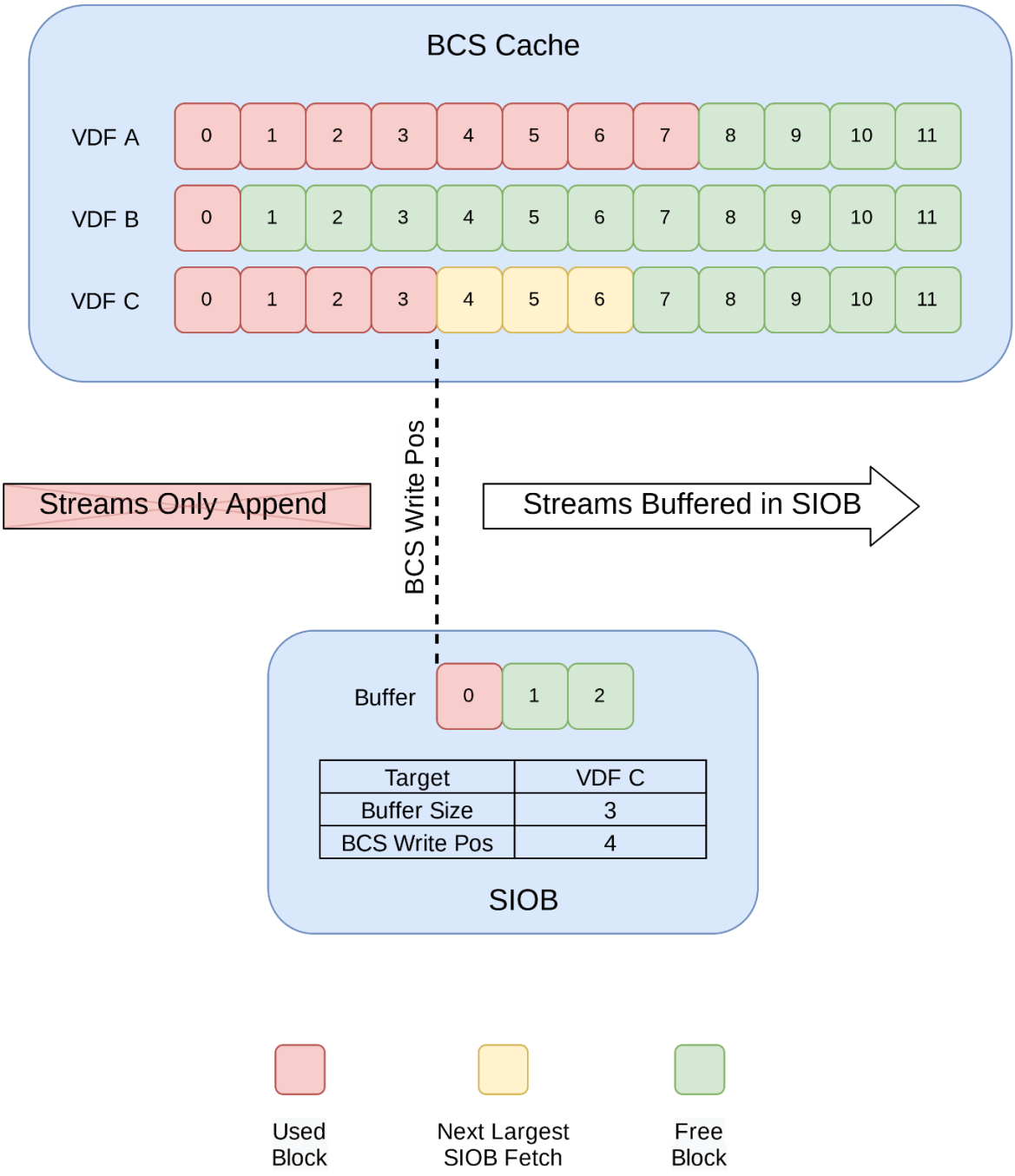


Figure 24: VDF write destination for stream-based output

4.8 System Call Behavior

For system calls that target normal disk files or directories, after recording provenance, RepeatFS relays the operation to the underlying file system (Section 2.4). Since VDFs are handled exclusively by RepeatFS and are operationally unique compared to normal disk files, each system call targeting a VDF must be handled individually by RepeatFS. As previously mentioned, with certain exceptions, system calls attempting to modify the file system are denied. These include: *chmod, chown, mknod, rmdir, mkdir, unlink, symlink, link, rename, utimens,* and *create.* I/O operations that are not denied are forwarded to the POH (Section 4.5). Since VDFs are never symbolic links, the *readlink* system call always throws the appropriate FUSE exception. The remaining system calls each have unique behavior.

4.8.1 System Call: Access

The *access* system call allows the current process to determine if it has read, write, or execute permissions on the file. Although always read-only, VDFs inherit the remaining permissions of their source files. Therefore, *access* calls are relayed to the source file. This ensures a VDF will indicate it has the same read or execute permissions as its source file when queried. If access calls were not relayed, when a process accessed a VDF associated with a source file the process did not have access to, the VDF operation would erroneously succeed. Since the operation was successful, the POH would attempt and fail to read the source file due to denied access. It would then return no data, which is an ambiguous result; an empty VDF could represent the operating system denying access to the source file, or it could indicate the output of the VDF

process simply returned no data. Relaying access calls avoids this and ensures the operating system reports permission issues when accessing the VDF.

4.8.2 System Call: Getattr

Like the access mode, VDFs inherit many of the file attributes of their source files. However, some of these attributes are overridden to reflect information that differs between the VDF and the source file, such as file size and modified date. *Getattr* system calls are first relayed to the source file, but the retrieved data structure is modified before it is returned to the calling process. These attributes often change during the lifetime of the VDF and are easily calculated, so this process is performed for each *getattr* system call.

The modification process begins by changing the access mode bits to reflect the VDF's permissions (Section 4.8.1). RepeatFS then sets the file size to the number of bytes stored in the BCS for the VDF. If the VDF is not finalized and no bytes are currently stored in the BCS, this indicates the VDF process has not written any data and has not finished executing. In this case, RepeatFS sets the file size to a preconfigured initial size; this is 0 by default but may be overridden by the user. Some applications check the file size before performing a read operation, and files with size 0 cause the application to abort the read. By overriding the initial size in the configuration file, a user can ensure these applications read the VDF, causing the BCS to retrieve data from the POH. Lastly, for VDFs that have never been accessed, the modified date is set to the UNIX epoch date; otherwise, the last time the BCS was updated for the VDF is used as the modified date. Taken together, the file size and modified date allow the user to ascertain if the VDF has been accessed previously, and if so, how much data is currently available for read operations.

4.8.3 System Call: Readdir

The *readdir* system call is responsible for providing the list of files and subdirectories present within a given directory. Since VDFs are not physically stored on the file system and do not exist within the BCS until accessed, the list of existing VDFs cannot solely be used to decide when to include a VDF in the directory listing; doing so would only show VDFs that had been previously accessed. Instead, when RepeatFS receives a *readdir* call targeting a VDF Directory, it returns a list of all entries corresponding to VDF definitions whose regular expression pattern match the source file (Section 4.1). This ensures that all possible VDFs associated with the source file are listed, including both ones previously accessed and those never accessed. Typical directory listing utilities such as "ls" perform a *getattr* call for each entry retrieved by *readdir*. The user may use information returned by this operation to ascertain if a VDF is new or an existing entry (Section 4.8.2).

Like all system calls targeting normal directories, RepeatFS will relay *readdir* calls to the underlying file system. However, if the user has configured VDF directories to be visible, RepeatFS will intercept the directory listing returned by the underlying file system and will add an entry corresponding to any VDF directories located within the directory. These will appear as normal subdirectories in the directory listing. The user must take care with this functionality, as VDFs may serve as source files for other VDFs, often resulting in infinite length VDF chains. If VDF directories are configured to be shown in the directory listing, and the user executes an application performing a recursive directory search, it will never terminate since there are an infinite number of VDF subdirectories to traverse. An example is the Linux "du" program, which recursively iterates through a directory tree to calculate the total number of bytes used by files in

each subdirectory. The “du” program would never halt if run in a RepeatFS mount with visible VDF directories.

CHAPTER 5

PERFORMANCE EVALUATION

5.1 Introduction

As detailed in previous chapters, RepeatFS provides an implementation of the reactive and proactive models via provenance services and VDFs. These systems are designed with usability in mind and the performance of RepeatFS is important for usability. This is especially relevant for researchers who deal with large amounts of data or I/O intensive computational tasks. For long-running analysis pipelines or very large workloads that take days or weeks to complete, too much of a performance penalty incurred by RepeatFS may result in these analyses potentially taking more than a month to complete. When paired with the cost associated with the infrastructure required to run the analysis, such as HPC and cloud environments in which the user is charged relative to their usage time, any substantial increase in processing time may render the solution financially infeasible. Additionally, the user may have scheduling or deadline requirements that cannot be met with decreased performance. In both cases, the user is forced to choose between reproducibility and practical considerations.

In order to ensure the RepeatFS overhead does not negatively impact I/O times to unreasonable levels, performance evaluation was performed across a variety of metrics. Not only does this process identify significant or unacceptable performance penalties that must be addressed to ensure usability, but it allows researchers with specialized compute needs to make an informed decision regarding whether using RepeatFS is compatible with any financial or scheduling constraints associated with their research. Examples of these needs include analyses that open thousands of individual input files, perform many write operations with small amounts of data, or

repeatedly recursively iterate through the file system. In these cases, particular system calls, such as *open* and *write*, are utilized disproportionally to other file system calls typical in an analysis. Accordingly, if the system call in question incurs a more significant penalty than other typical calls, the researcher should be aware that RepeatFS will have a greater impact on this type of analysis.

5.2 Testing Environment

All tests were performed on a Debian Linux system running kernel version 5.8.0-2 with a 3.6GHz CPU and 16GB of RAM. Ext4 was used as the underlying native file system on a 7200RPM SATA disk drive with 256MB of cache. A 1GB file was read or written during each test; we chose this size to ensure a sufficient number of operations were necessary to process the entire file. This file was removed from the cache prior to each read test by opening it using the "dd" command (Table 6). It was then set to the expected testing state by reading the entire file through the applicable file system. To remove any potential jitter introduced from unrelated system activity, ten iterations of each test were performed and averaged to produce a final result.

5.3 Core Functionality

In order to record the provenance associated with file system calls, RepeatFS acts as an interface between a running process and storage. For disk file operations, this storage is the underlying file system, and for VDF operations this storage is the BCS. For every call RepeatFS receives, it performs all relevant internal actions, relays the call from the process to storage, and relays the response from storage to the process (Chapters 2, 3). Since identical system calls may be made directly to the underlying file system and via RepeatFS, a direct performance comparison

can be made between the two file systems for the same disk file, establishing the penalty incurred due to the overhead of core RepeatFS functionality.

Scientific analysis processing often involves opening a small number of files once, performing many reads and/or writes, and then closing the file descriptors. Since there are significantly more *read* and *write* calls than *open* and *close* calls in this scenario, establishing performance statistics in relation to I/O volume is especially important. It is also necessary to evaluate performance over relevant parameters. The count and size of each read/write operation may vary; a small number of large writes may perform differently than a large number of small writes. Additionally, leveraging the page cache can have a significant impact on performance. Cache hits or misses can affect if data is retrieved from disk or memory, and RepeatFS involves two potential interactions with the cache (Figure 3), leading to complex performance relationships depending on the state of the cache.

In order to address these different conditions, provenance overhead was evaluated with each combination of five parameters (Table 6). The "dd" command was used to read or write the 1GB file with the respective read/write size and operation count. Read tests used the "/dev/null" device as output and write tests used the "/dev/zero" device as input. Throughput, kernel mode CPU time, user mode CPU time, and real time were recorded for each test using the "time" command. All tests were performed using a single thread, allowing off-CPU time (O) to be calculated by subtracting kernel mode (K) and user mode (U) CPU time from the real time (R) (Equation 4). This metric represents the amount of time the CPU spent in a blocked, waiting, or halted state, such as when waiting for an I/O operation to complete or requesting a lock.

$$O = R - (K + U) \tag{4}$$

Table 6. Performance evaluation parameters

Parameter	Values
I/O direction	Read Write
File system	Native (Underlying) RepeatFS
Cache state for reads	File not in either page cache File only present in native page cache File present in RepeatFS page cache Direct I/O
Operation count	2^N (For N = 0..22)
Operation size	2^N (For N = 8..30)

Operation count and size are calculated in relation to each other, where $N_{size} = 30 - N_{count}$. This ensures the total amount of data read/written for each test iteration is uniform at 2^{30} bytes, or 1GB. For removing the file from the page cache, the following command was used: "dd of=ofile oflag=nocache conv=notrunc,fdatasync count=0".

5.3.1 Core Functionality: Uncached Reads

When *read* calls target a disk file that is not present in the native file system's portion of the page cache, nor in RepeatFS's portion, performance is similar across all metrics, with native calls only performing up to 5% better. Throughput and CPU time are both highly dependent on the size of the read (Figures 25-29). Small, suboptimal read sizes of 2KB or smaller are highly variable in their performance, with native calls performing worse than RepeatFS. This relationship is counterintuitive since RepeatFS relays calls to the underlying file system and therefore should always have lower throughput than the direct call. However, since 1GB of data is always read, and the number of reads is inversely proportional to the size of the read, the number of *read* calls is much larger for smaller reads. This increases the user mode CPU time spent preparing and issuing these requests by the application (Figures 26-29), all of which is subject to scheduling by the

kernel. As scheduling is highly dependent on available resources, running processing, and other state dependent factors, it is likely the RepeatFS related processes were scheduled differently in these scenarios. For smaller numbers of read operations for read sizes of 2KB or greater, user mode CPU time was negligible in comparison to off-CPU time (Figure 28, 29), and performance variability decreased.

In all cases, since the file was not present in the page cache, it was retrieved from disk and required I/O, as illustrated by the high off-CPU times. The optimal read size appears to start at 8KB for RepeatFS and 2MB for the native file system, and end at 256MB for both; these ranges maximize throughput and off-CPU time while minimizing real time. Greater read sizes than 256MB increase kernel CPU time, likely due to the size exceeding internal kernel or disk buffer sizes, requiring multiple internal kernel operations to complete a single request. For all read sizes, RepeatFS only performs slightly worse than the underlying file system, and the real time required to read a 1GB file differs in less than one second in all cases. The time required by RepeatFS to process system call requests and record provenance does not have a noticeable impact on overall read performance for uncached files with a difference of less than one second. This is a very common scenario for researchers performing analysis on large files. Since the page cache is limited in size, a single large file may not fit entirely within the cache, or subsequently accessing other large files may flush the original file from the cache; chances are high that the requested data is not cached and requires disk I/O. Our experiments show that RepeatFS has little negative performance impact for the user in these cases.

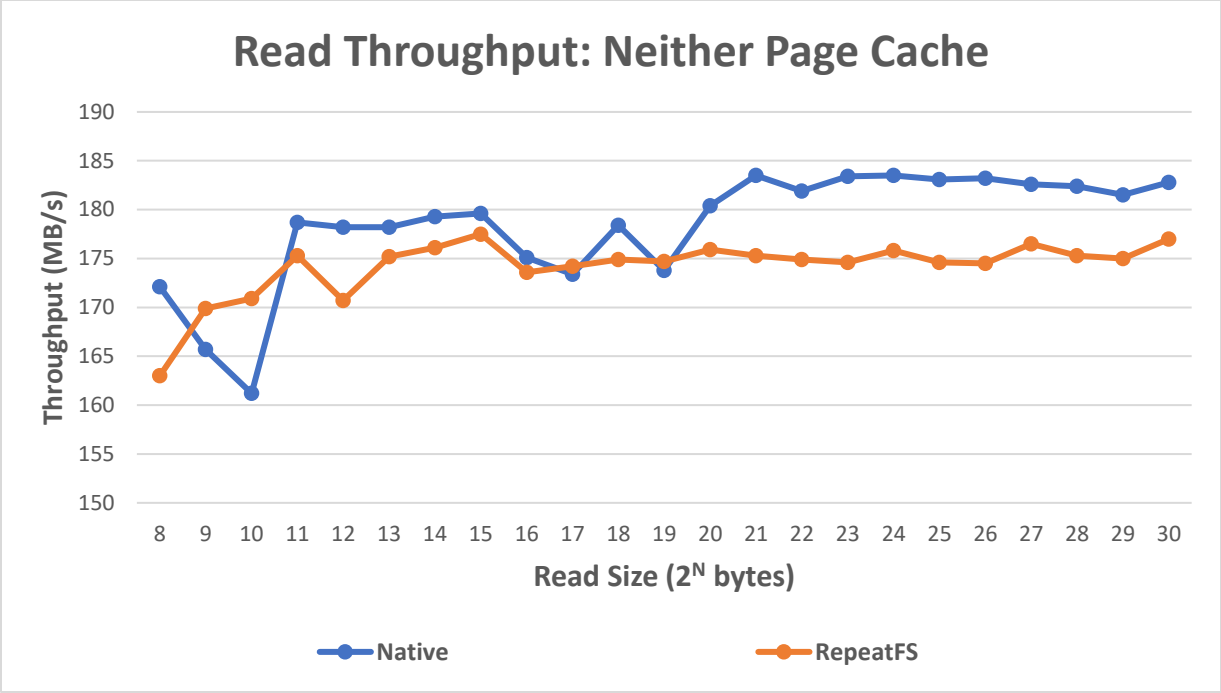


Figure 25: Read throughput (1GB): file in neither page cache

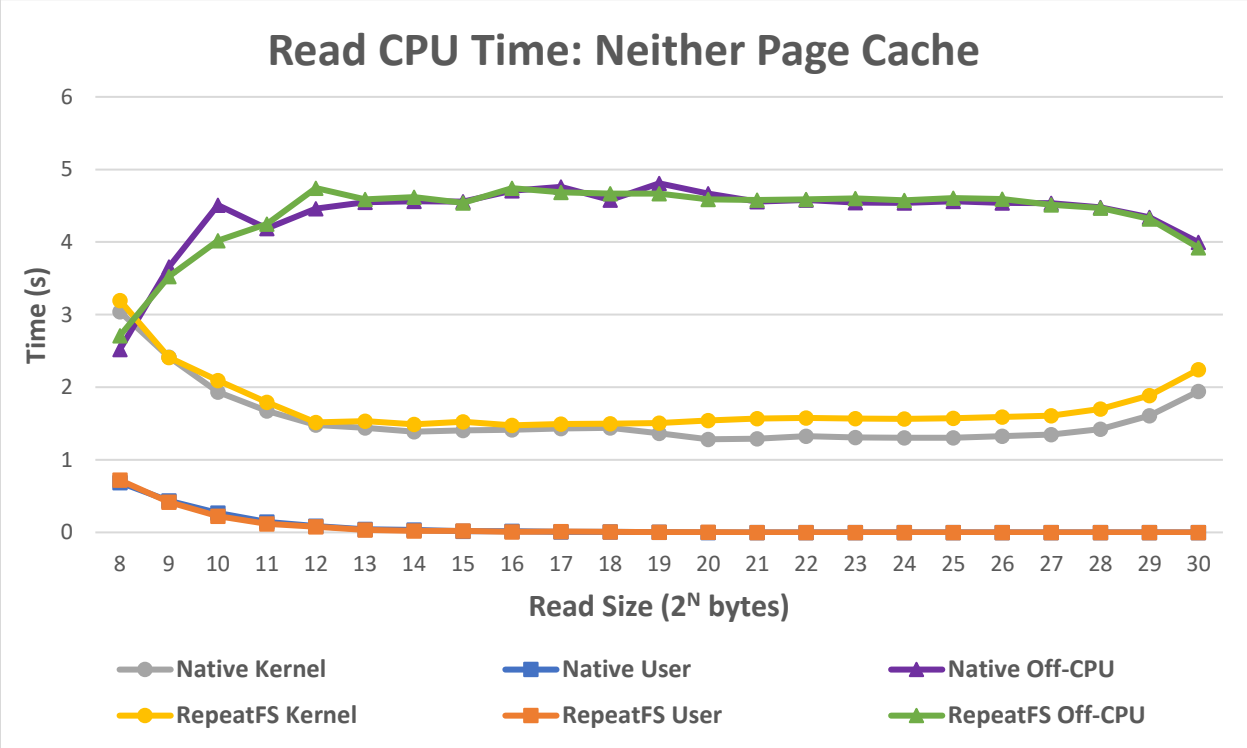


Figure 26: Read CPU time (1GB): file in neither page cache

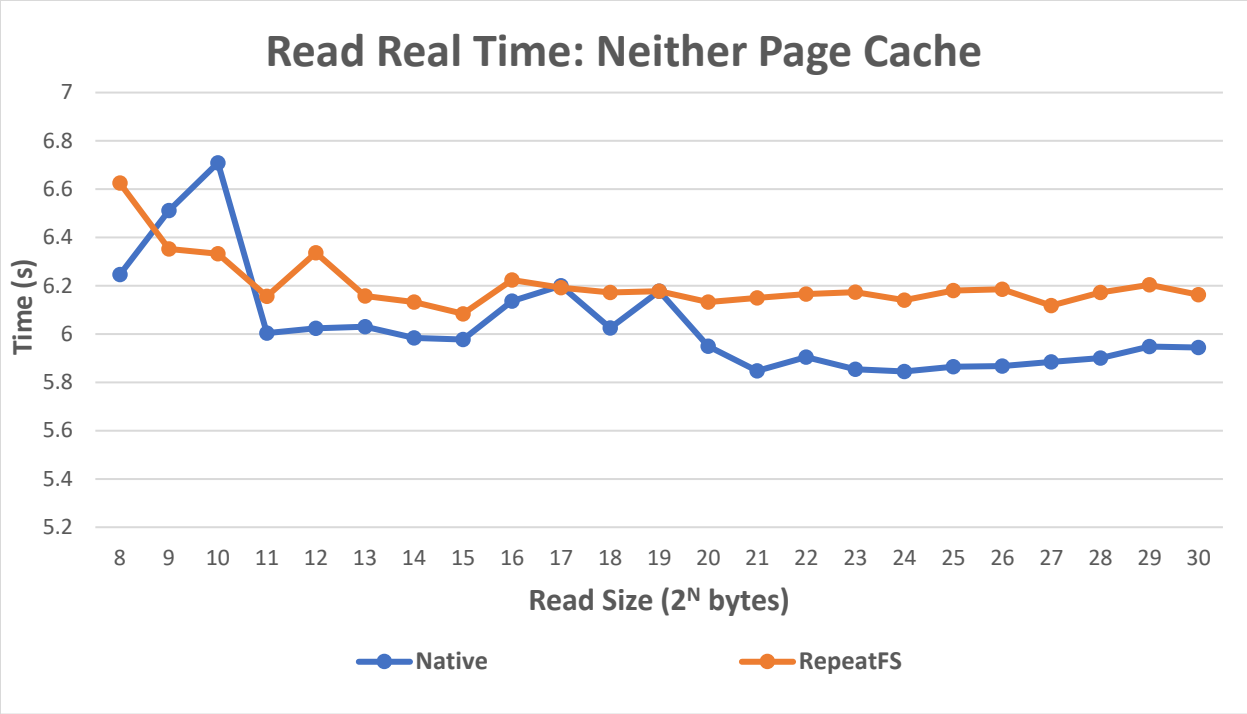


Figure 27: Read real time (1GB): file in neither page cache

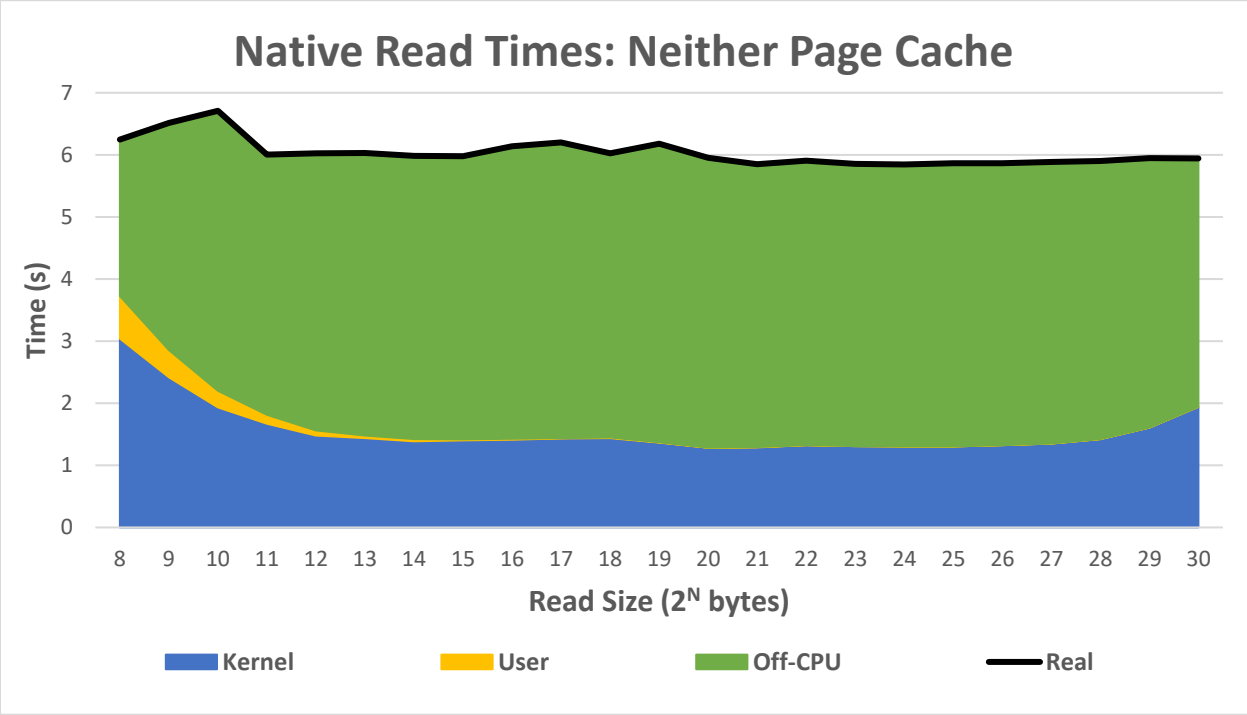


Figure 28: Native read times (1GB): file in neither page cache

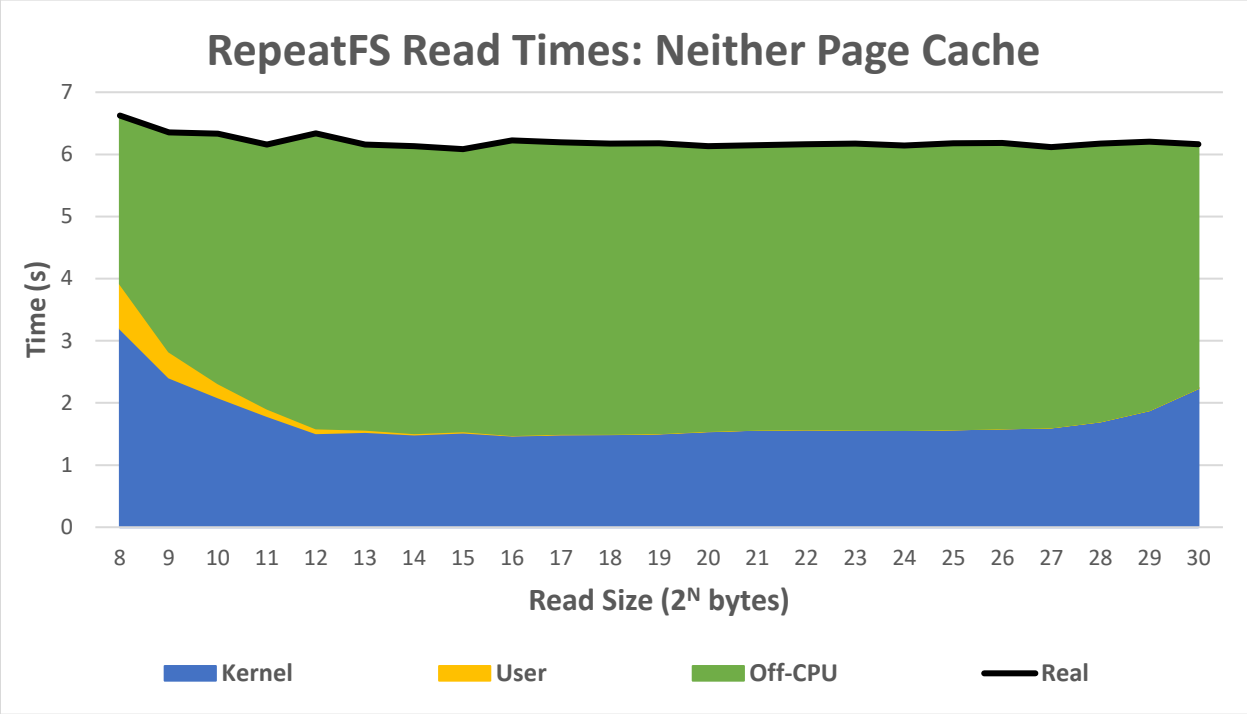


Figure 29: RepeatFS read times (1GB): file in neither page cache

5.3.2 Core Functionality: Cached Reads

When *read* calls target a disk file that is present in both the native file system's portion of the page cache and RepeatFS's portion, performance is virtually identical across all metrics. This is expected behavior, as the kernel is responsible for managing the page cache, and neither file system is involved in servicing these system calls. The CPU times reflect this, with kernel mode activity representing almost the entirety of the request (Figures 31-34). Similar to when the file is uncached, there is still an increase in user mode and off-CPU activity for small read sizes due to the application preparing and issuing the requests. Additionally, for all read sizes, the off-CPU time is slightly increased for RepeatFS in comparison to the native file system due to the additional

waiting and context switching likely associated with a FUSE based file system for the non-*read* calls involved in reading the file, such as *getattr*, *open*, and *release*.

RepeatFS and native reads both receive a performance improvement when the file is present in the page cache. Under optimal read sizes, throughput increases from approximately 180MB/s to over 3.5GB/s (Figure 30), reflecting the faster service times of main memory over disk I/O. Both file systems service the entire request in approximately 300ms. Unlike uncached reads, the optimal read size for cached data is 64KB, corresponding to the Linux page size of the test system. For RepeatFS, 64KB read sizes also lie within the optimal range of uncached reads, making this size a good choice for users writing pipeline scripts that run on RepeatFS instances. With a 64KB read size, maximum performance is achieved regardless of whether the data is present in the page cache or not.

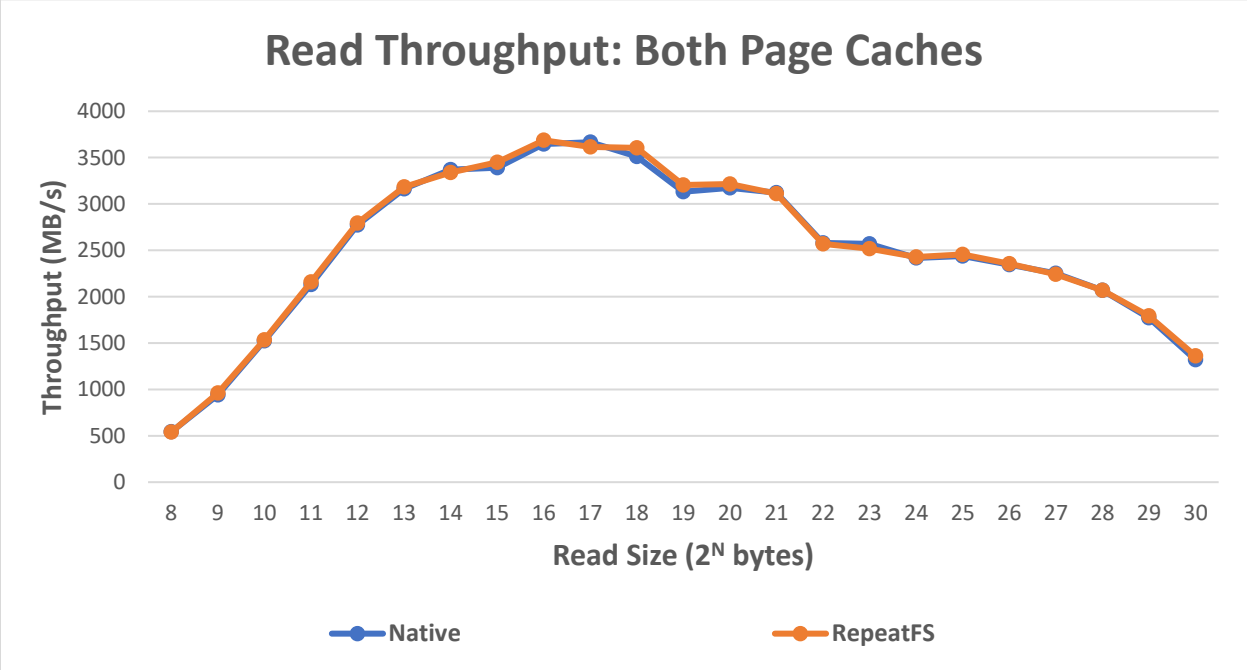


Figure 30: Read throughput (1GB): file in both page caches

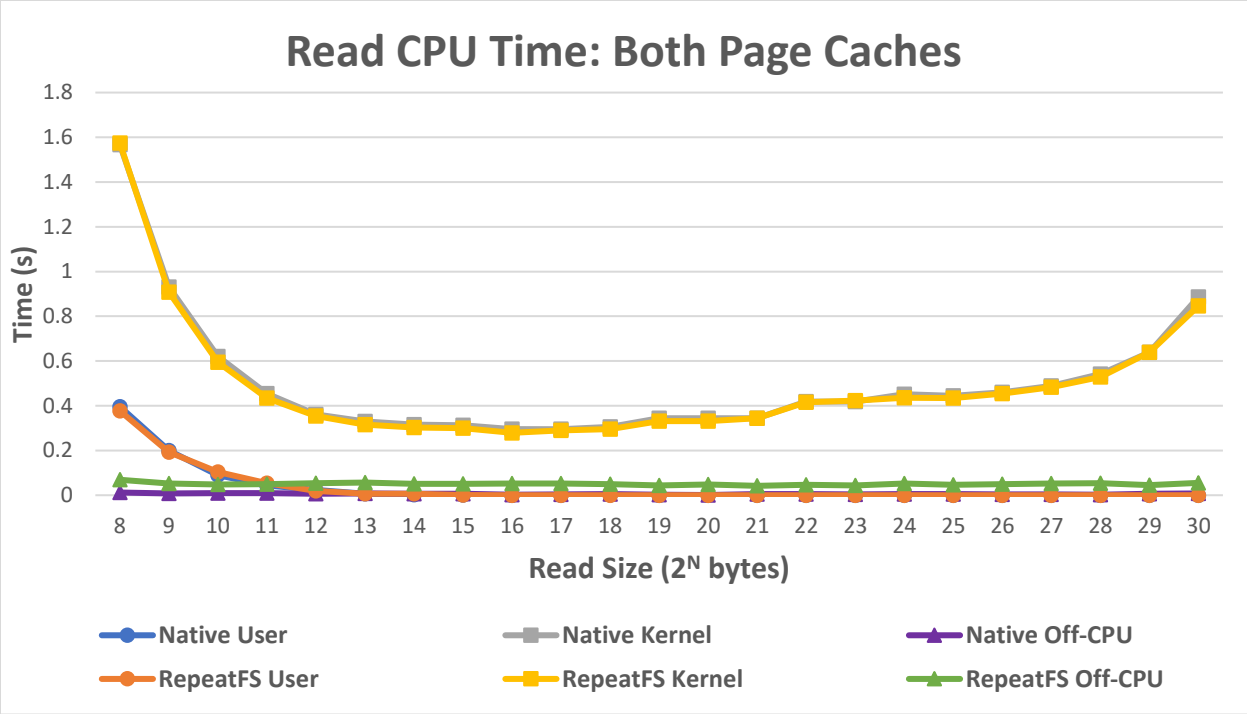


Figure 31: Read CPU time (1GB): file in both page caches

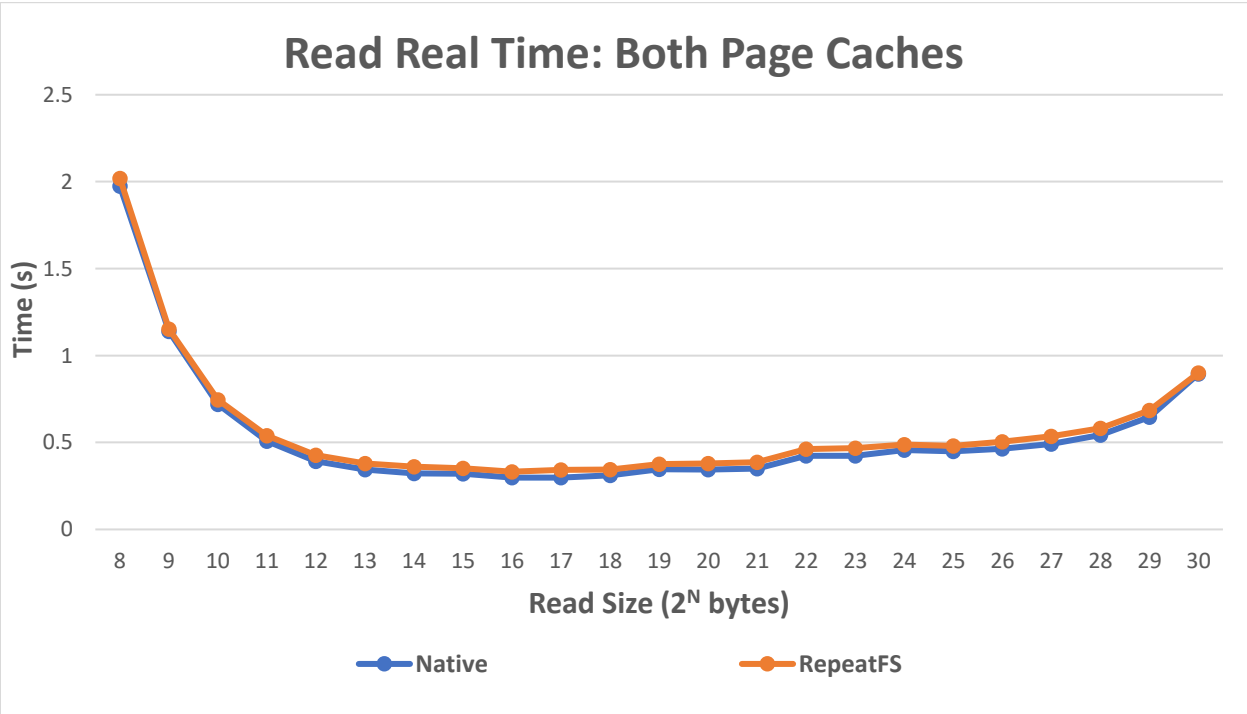


Figure 32: Read real time (1GB): file in both page caches

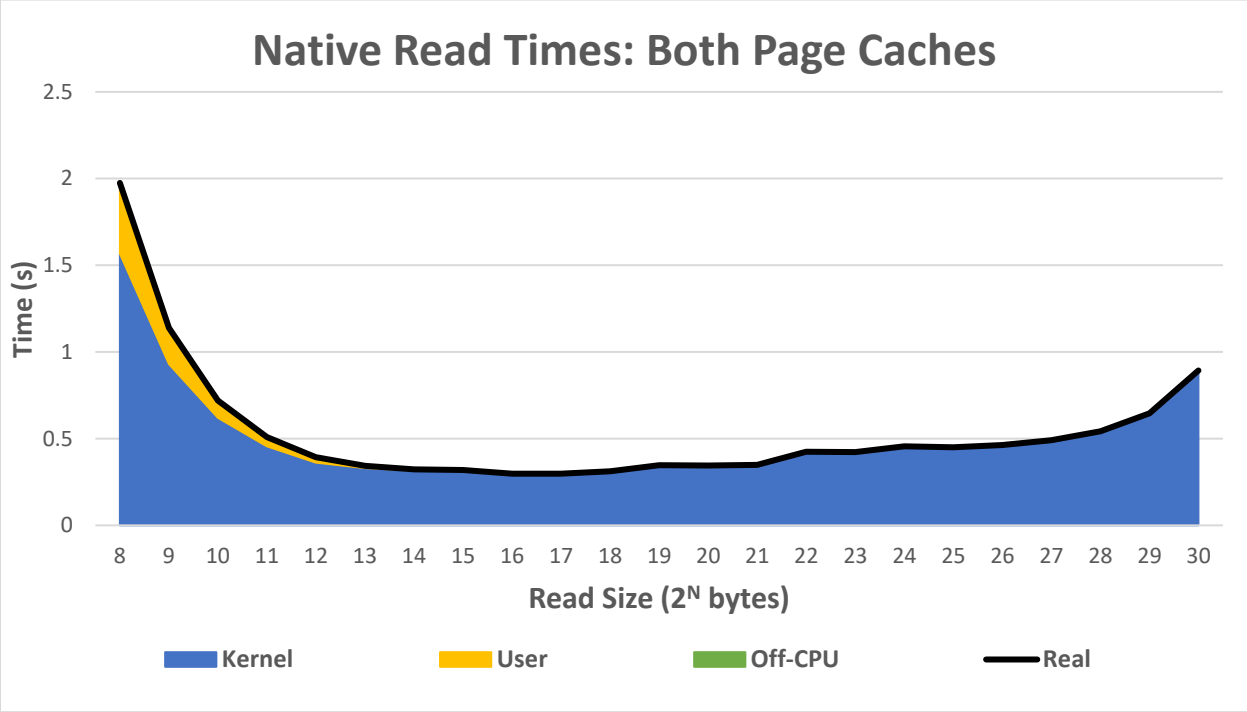


Figure 33: Native read times (1GB): file in both page caches

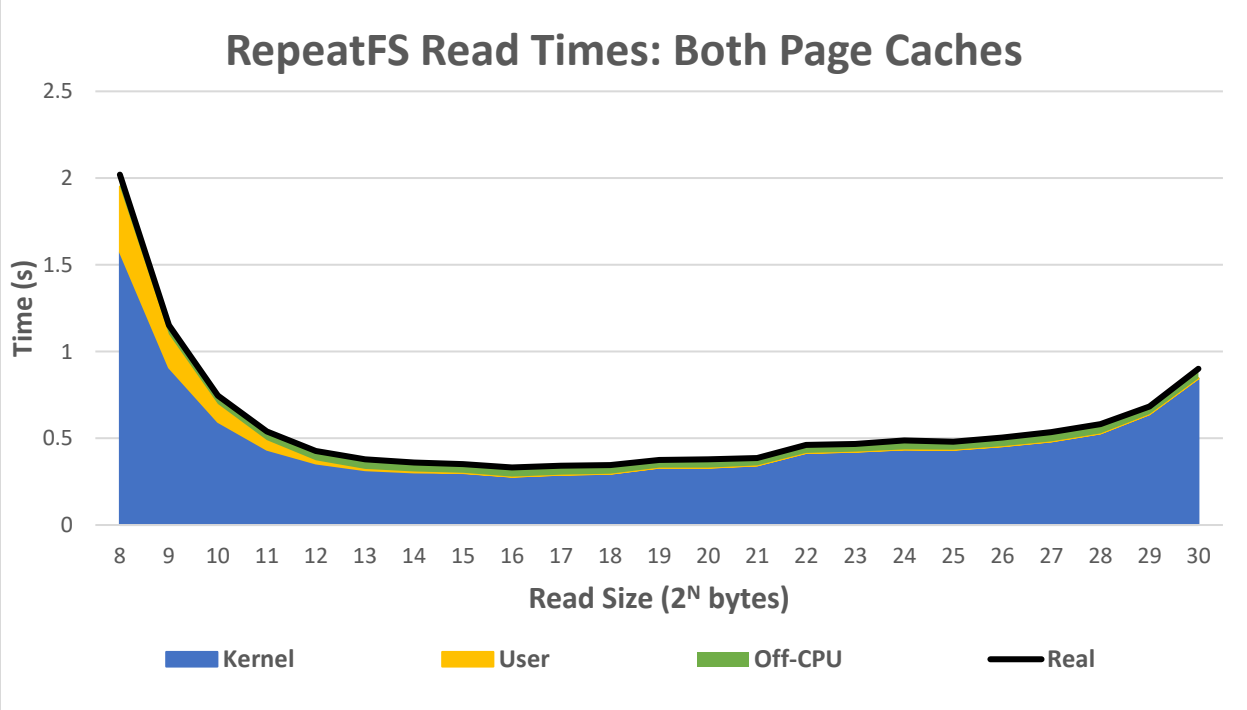


Figure 34: RepeatFS read times (1GB): file in both page caches

5.3.3 Core Functionality: Mixed Cached Reads

There are scenarios in which a file being read is present in the native portion of the page cache but not RepeatFS's portion, even if the file is located within a directory monitored by a RepeatFS instance. This loss in coherence between the two copies of the data in the page cache has multiple causes: processes accessing these files directly without utilizing a RepeatFS instance, or processes reading large files whose contents cannot fit into memory twice. In the first case, the process is accessing the native file system, and therefore the kernel stores the data in the native portion of the page cache only. In the second case, the two copies of the file compete for limited memory resources, and one portion of the data is flushed from the cache to accommodate the other.

Although a direct comparison obviously shows significantly better performance for cached reads over uncached reads (Figures 35-39), evaluation of RepeatFS in this mixed cache scenario is still useful. Since RepeatFS relays *read* calls to the underlying file system for reads to disk files, these calls will return to RepeatFS much more quickly since the data is retrieved from the page cache, and therefore RepeatFS returns to the calling process more quickly as well. When compared for optimal read sizes to uncached reads (Figures 25-29), RepeatFS throughput nearly triples, with rates increasing from approximately 175MB/s to 500MB/s. Similarly, total real time decreases from six seconds to a little over two seconds. This illustrates that RepeatFS still benefits from the page cache, even if the data is only cached for the underlying file system.

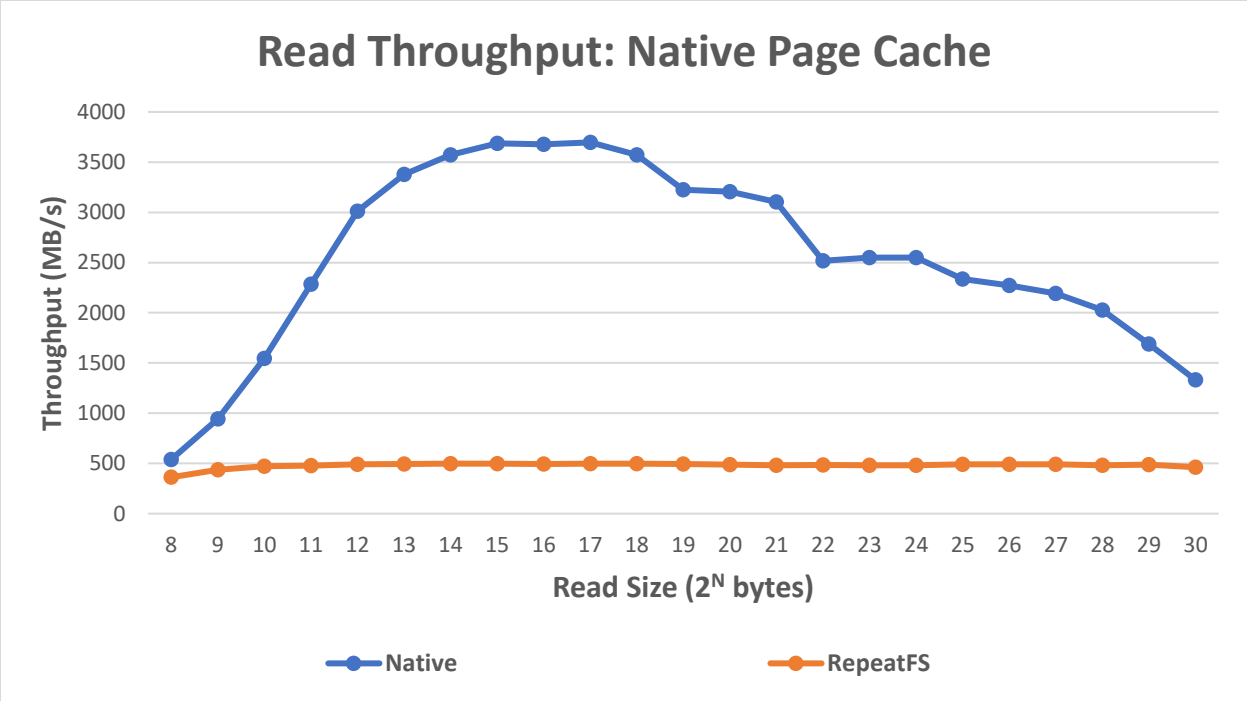


Figure 35: Read throughput (1GB): file in native portion of page cache

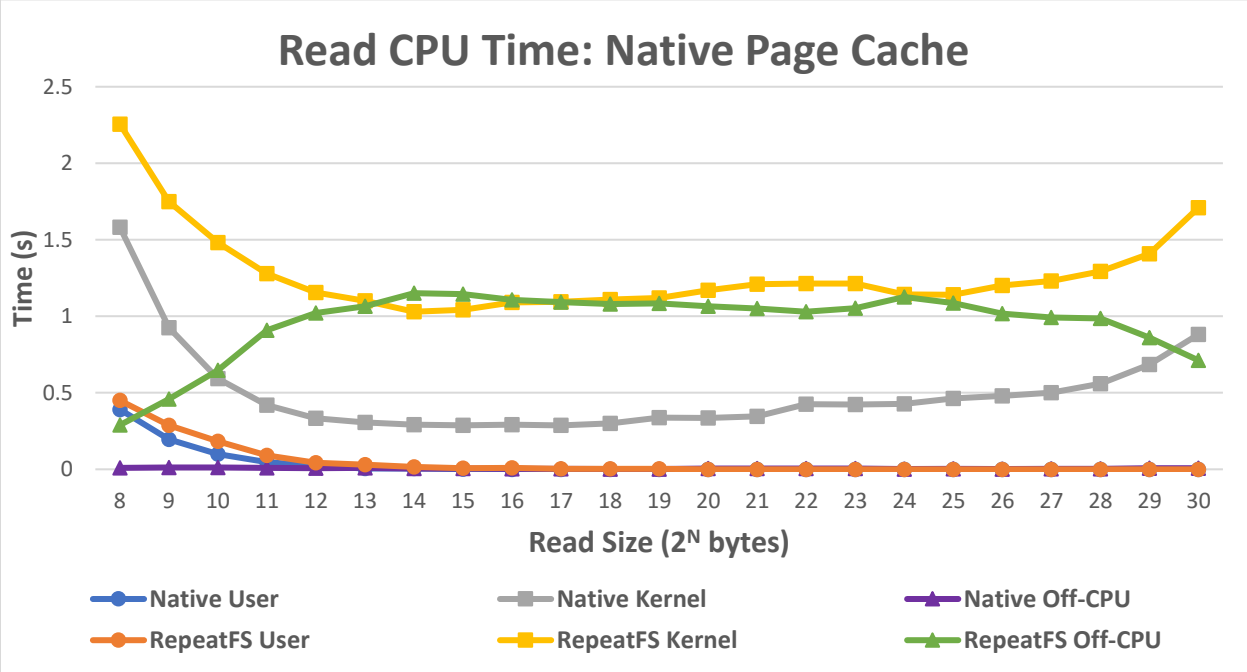


Figure 36: Read CPU time (1GB): file in native portion of page cache

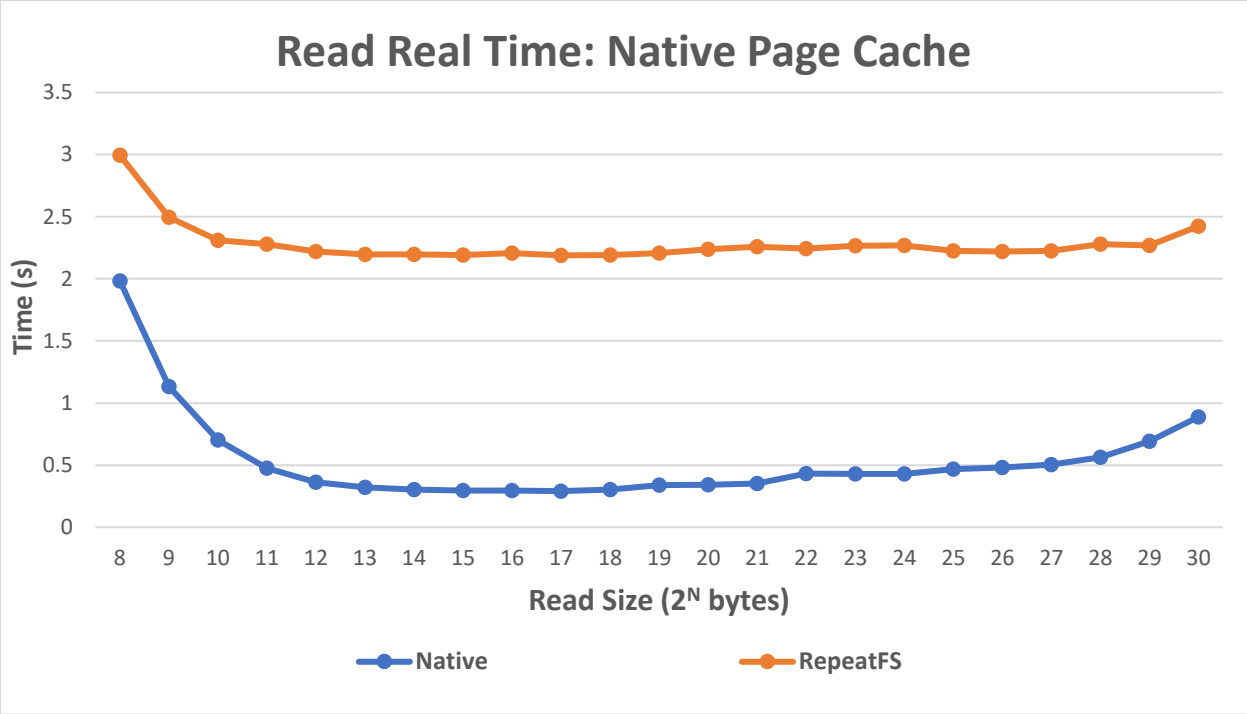


Figure 37: Read real time (1GB): file in native portion of page cache

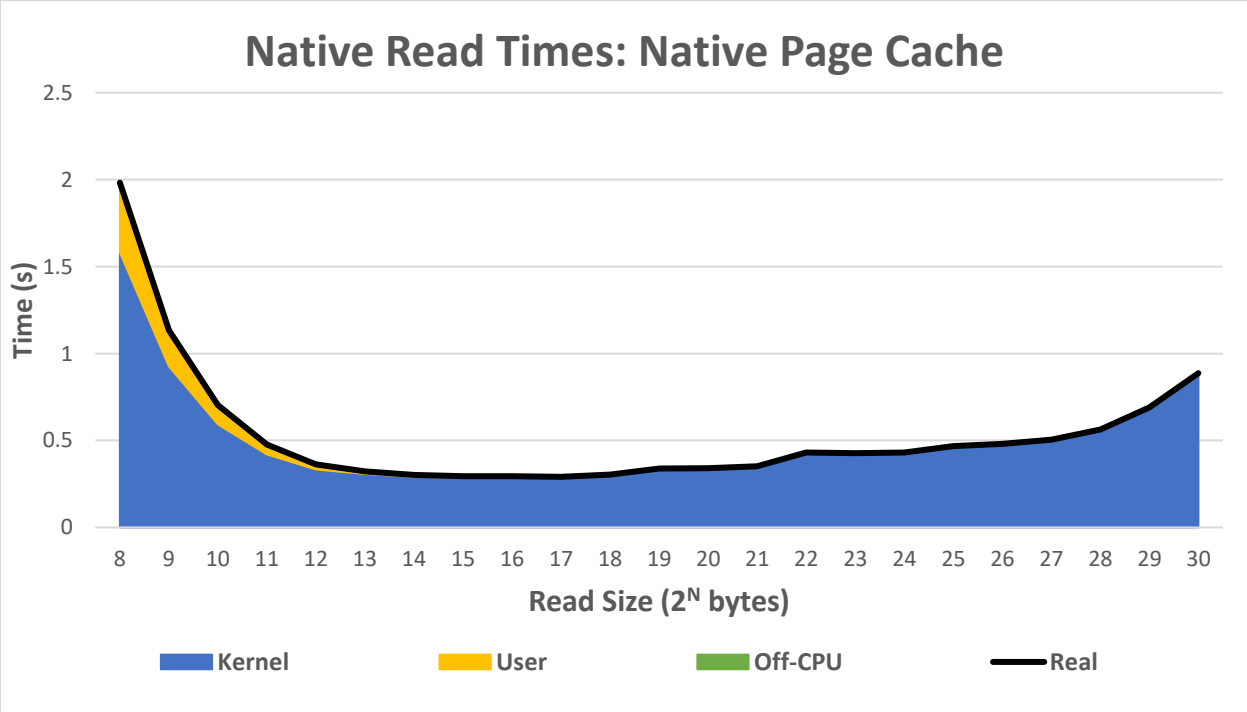


Figure 38: Native read times (1GB): file in native portion of page cache

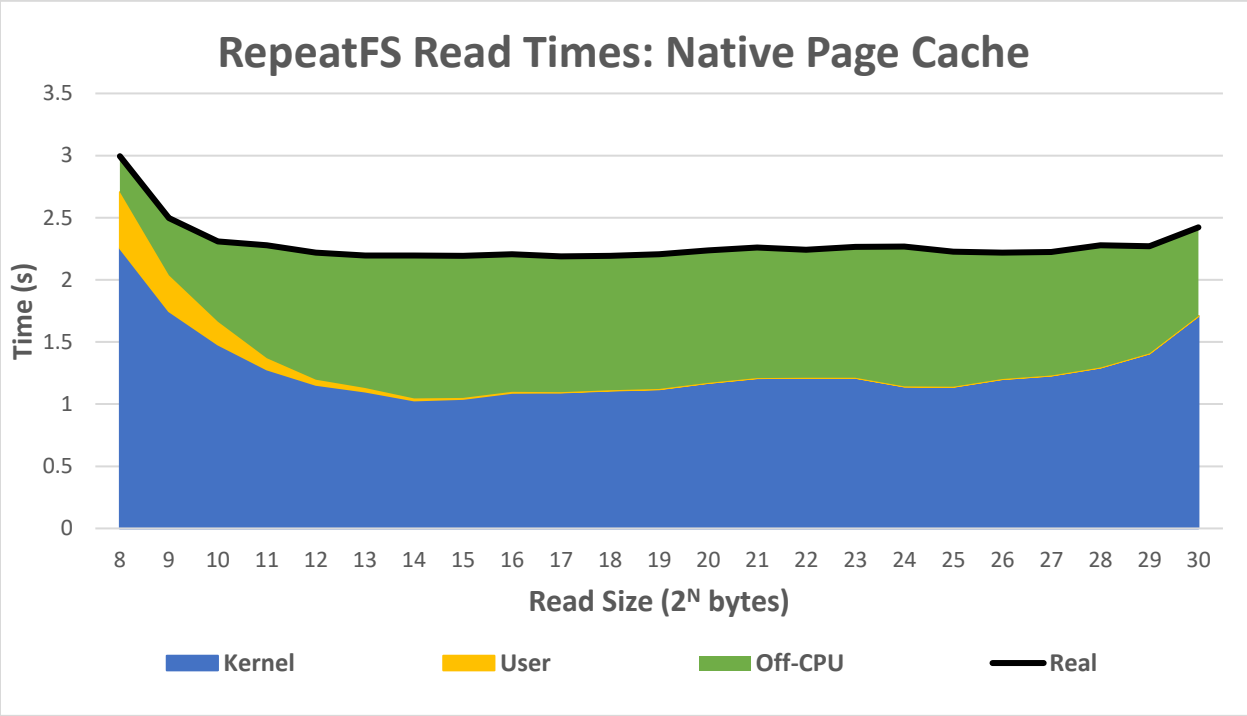


Figure 39: RepeatFS read times (1GB): file in native portion of page cache

5.3.4 Core Functionality: Direct I/O Reads

Due to limitations in Python, a file cannot be opened using standard file handling functions in direct I/O mode in order to bypass the page cache. This limitation is due to a requirement of the underlying *open* call, which restricts files opened with the *O_DIRECT* flag to use a buffer aligned on a 512-byte boundary. Python does not expose control over memory locations to the user, and the standard memory allocation routines do not align to this boundary. Attempts to use the standard file handling functions in Python with the *O_DIRECT* flag result in the underlying *open* call returning an error indicating this issue. Although Python offers a module that supports memory-aligned buffers, RepeatFS does not utilize it. This module carries its own restrictions and would require redesigning many parts of RepeatFS. Since most researchers performing informatics analysis rarely have the need to bypass the page cache via direct I/O and are instead focused on

the highest possible performance, RepeatFS purposely does not address this use case. Although it accepts direct I/O calls for compatibility, these requests still make use of the page cache. When RepeatFS receives an *open* call with the *O_DIRECT* flag specified that targets a disk file located on the underlying file system, it performs a bitwise operation to mask this flag before relaying the call to the underlying file system. This results in a partially direct I/O call, in which the RepeatFS portion of the page cache is bypassed, but the native portion is still used.

A cache miss and explicitly requesting to bypass the cache both result in the cache not being used. Since RepeatFS direct I/O bypasses its portion of the page cache but uses the native portion when relaying operations, the performance mirrors that seen in section 5.2.3 on mixed cached reads, in which the target file was only present in the native page cache (Figures 35 vs 40). Direct I/O performance does slightly worse than this previous test, as Off-CPU throughput has decreased by 50 MB/s, perhaps due to the different routines utilized in direct I/O mode.

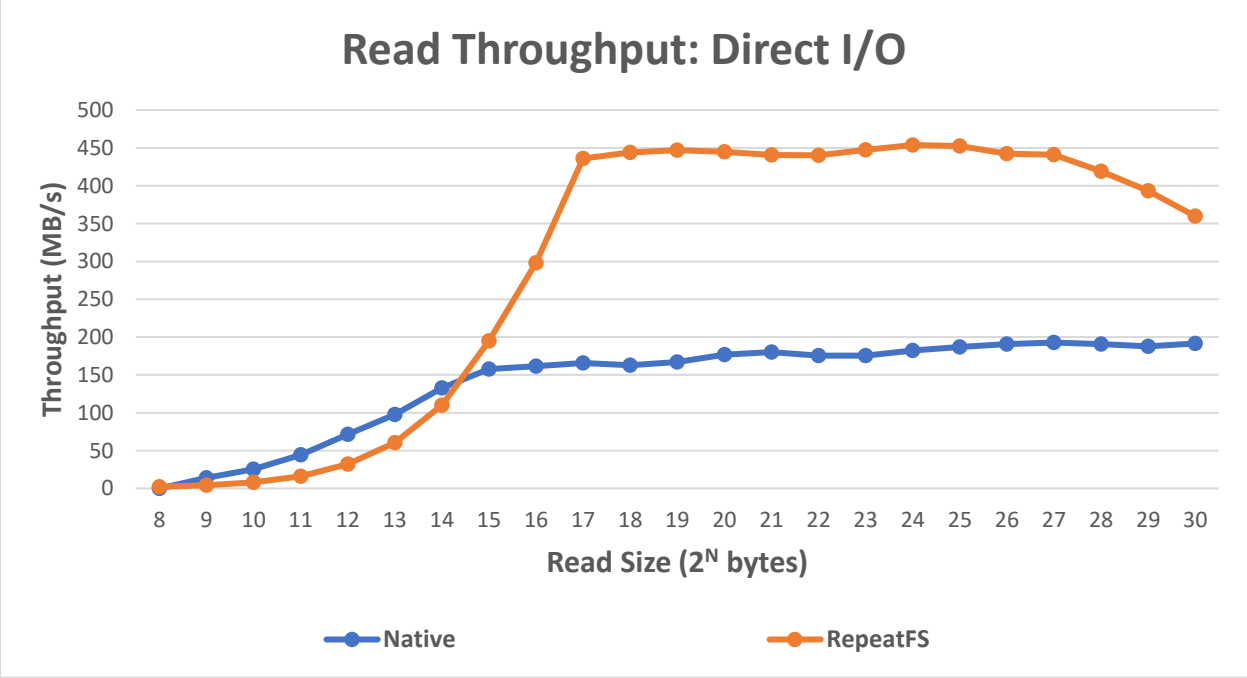


Figure 40: Read throughput (1GB): direct I/O

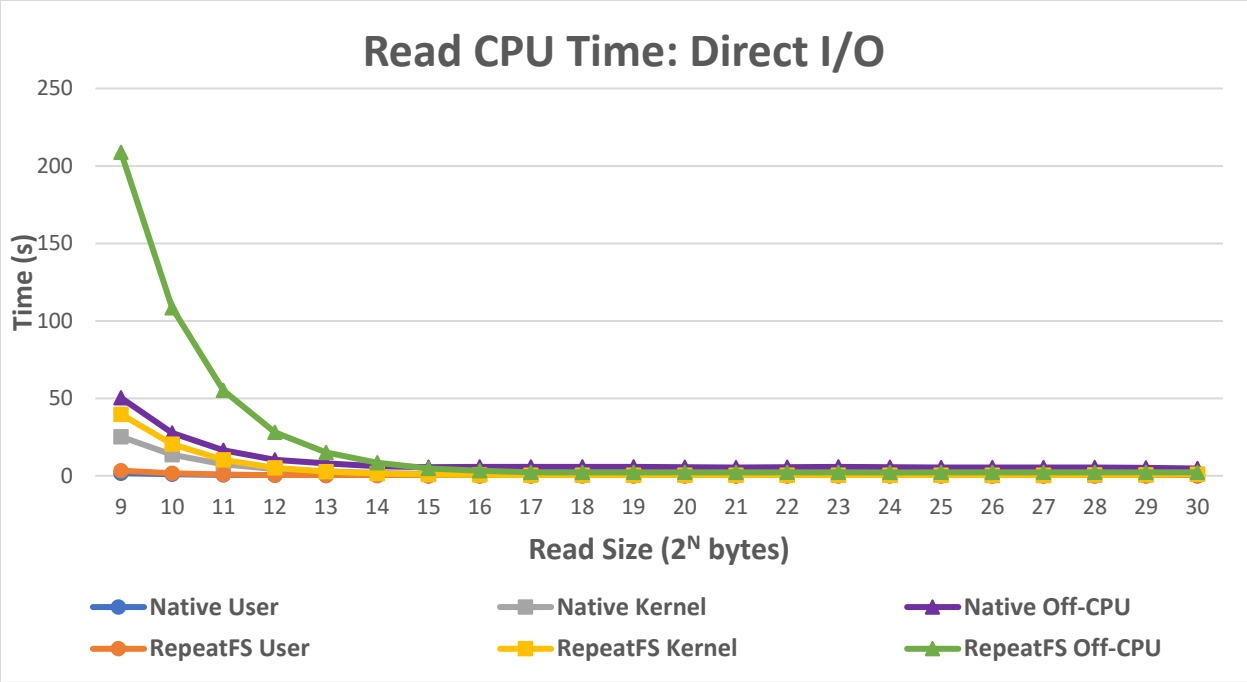


Figure 41: Read CPU time (1GB): direct I/O

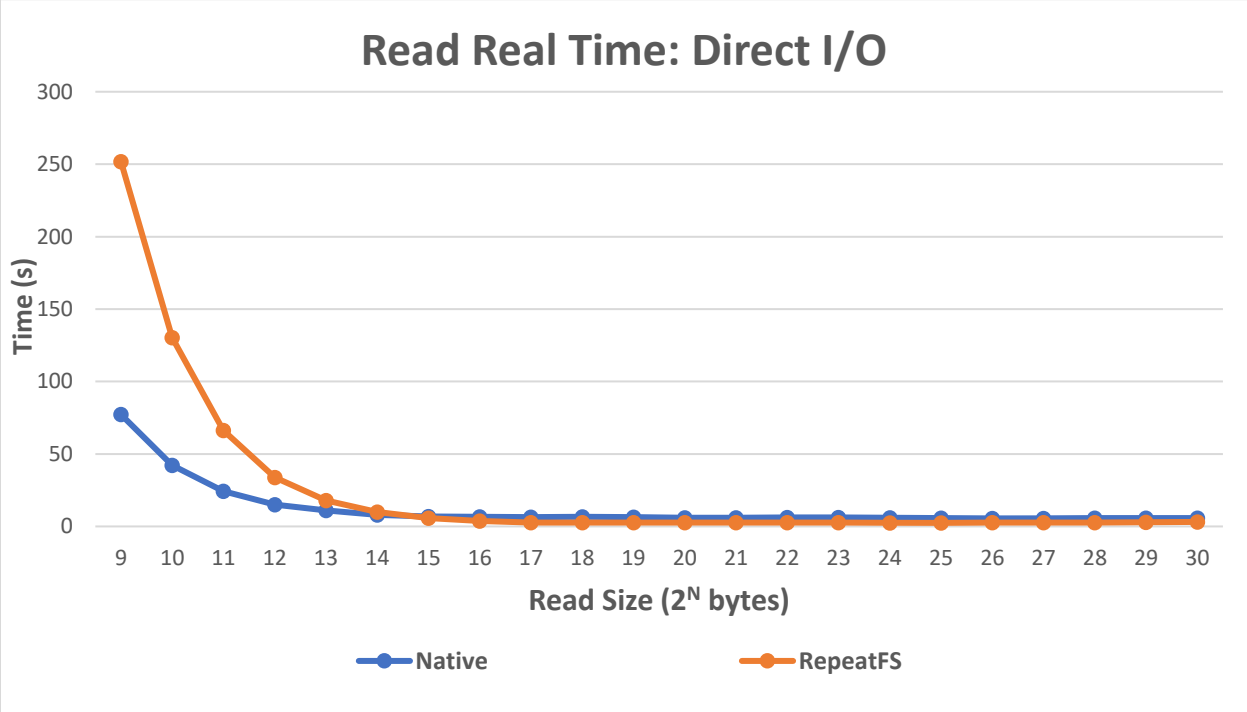


Figure 42: Read real time (1GB): direct I/O

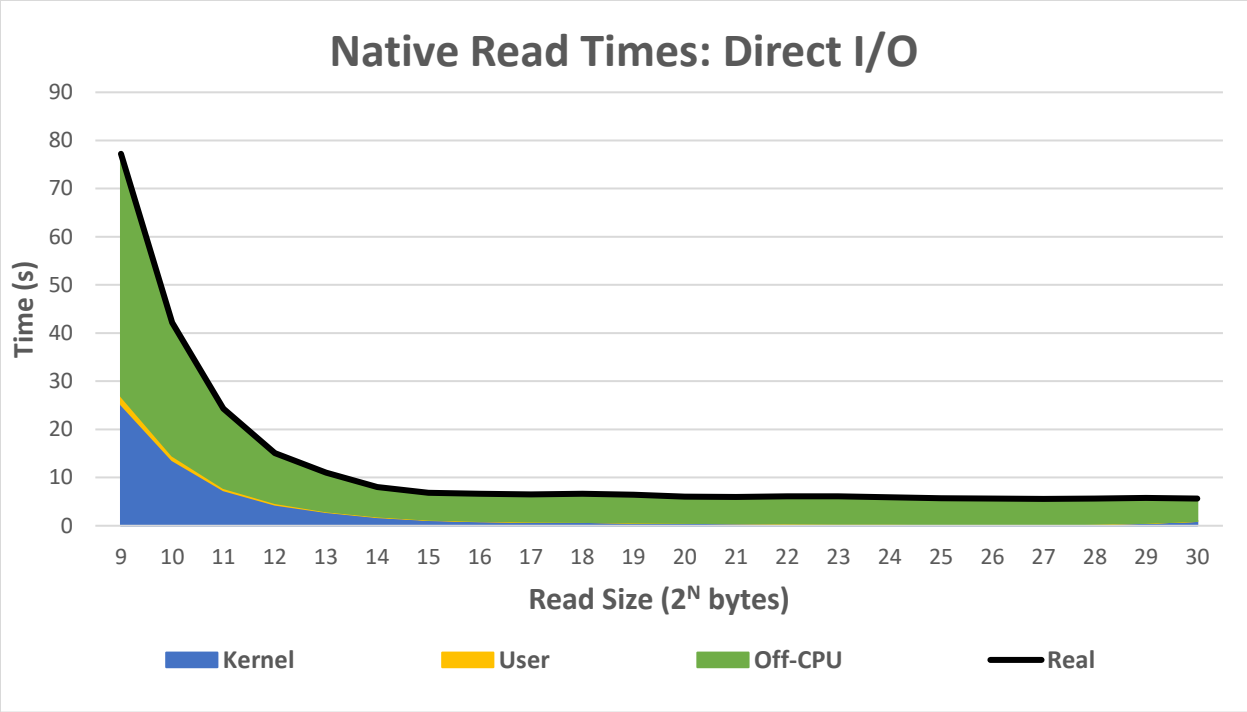


Figure 43: Native read times (1GB): direct I/O

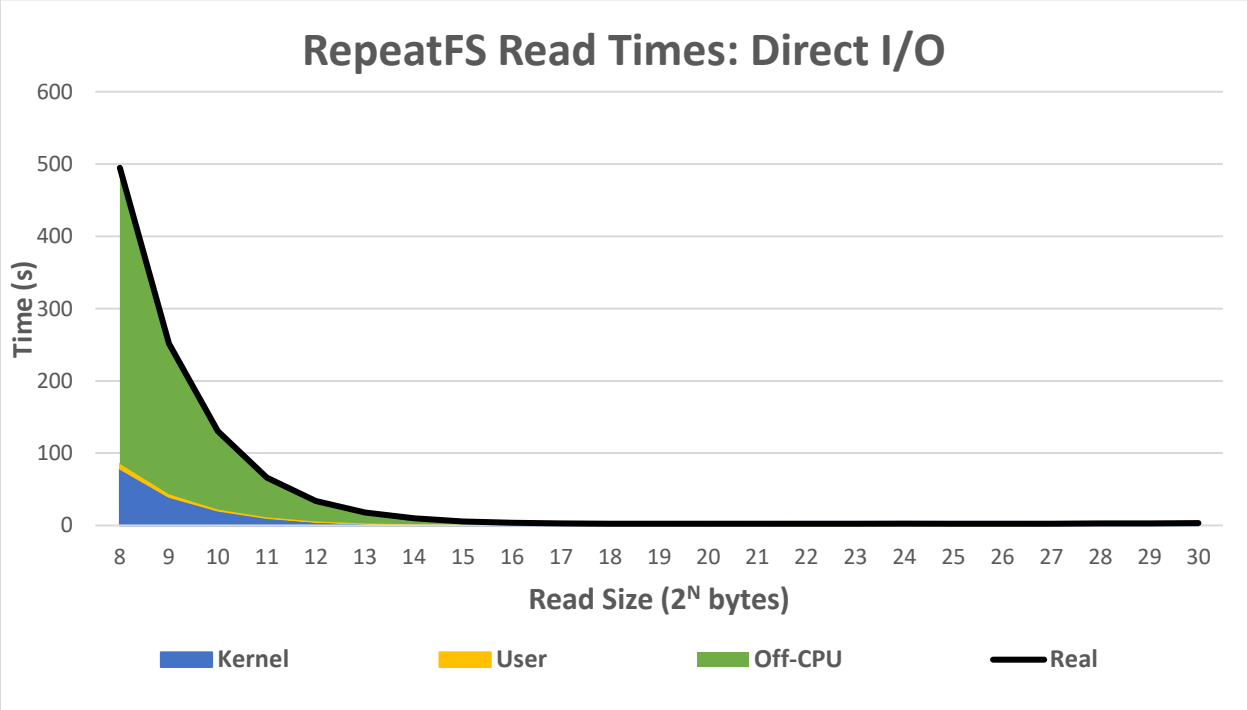


Figure 44: RepeatFS read times (1GB): direct I/O

5.3.5 Core Functionality: Writes

Data written to Linux file systems with page cache support are normally stored in the cache, marked as dirty, and written to the underlying file system in batch at a later time. This increases performance for the writing process since no direct disk I/O is involved. However, the current version of the fusepy Python module does not support asynchronous page cache writeback (Section 2.3). Instead, all writes made by a process to RepeatFS are immediately received as direct I/O. This known limitation does impact performance when compared to the native file system. At optimal write sizes, native throughput is approximately 185MB/s, while RepeatFS throughput is approximately 25MB/s, around seven times slower (Figure 45). The real times maintain this ratio as well, with the native file system taking approximately 6 seconds to complete, and RepeatFS taking 41 seconds (Figure 46). The off-CPU times of 4 seconds and 35 seconds for the native file system and RepeatFS, respectively, confirms that write activity is being handled differently between the two (Figures 48, 49). RepeatFS's high off-CPU times indicate the call is being handled synchronously in the same process call stack; off-CPU delays due to FUSE related context switches, locks, and other mechanisms each contribute to the overall off-CPU time of the calling process. The native file system instead directly writes to the page cache, and a separate kernel process later writes these dirty pages to disk, resulting in a reduced off-CPU time for the calling process.

Although RepeatFS's write performance is lower than the native file system's, typical informatics workflows are heavier in reads than writes, as they read in a large amount of data, perform analysis methods, and then write smaller result files. However, for tasks that generate significant amounts of data from de novo processes or by converting input data, users should be aware of this current limitation in RepeatFS and plan their computational analyses accordingly.

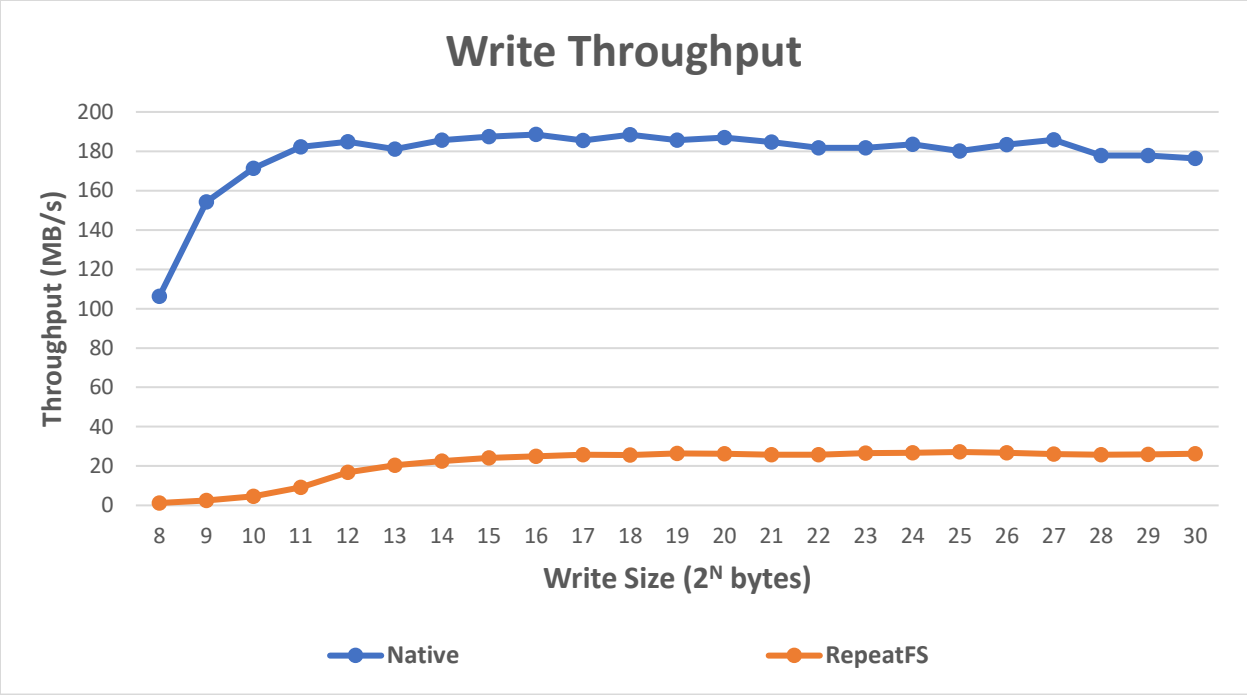


Figure 45: Write throughput (1GB)

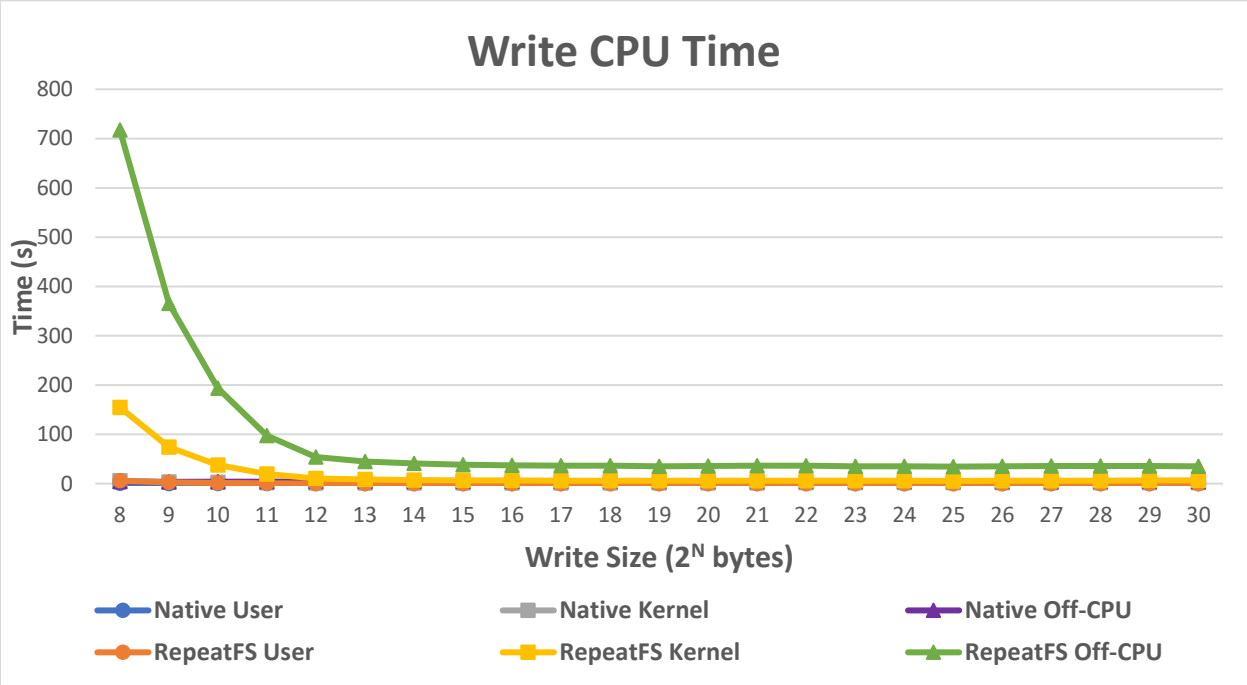


Figure 46 (1GB): Write CPU time (1GB)

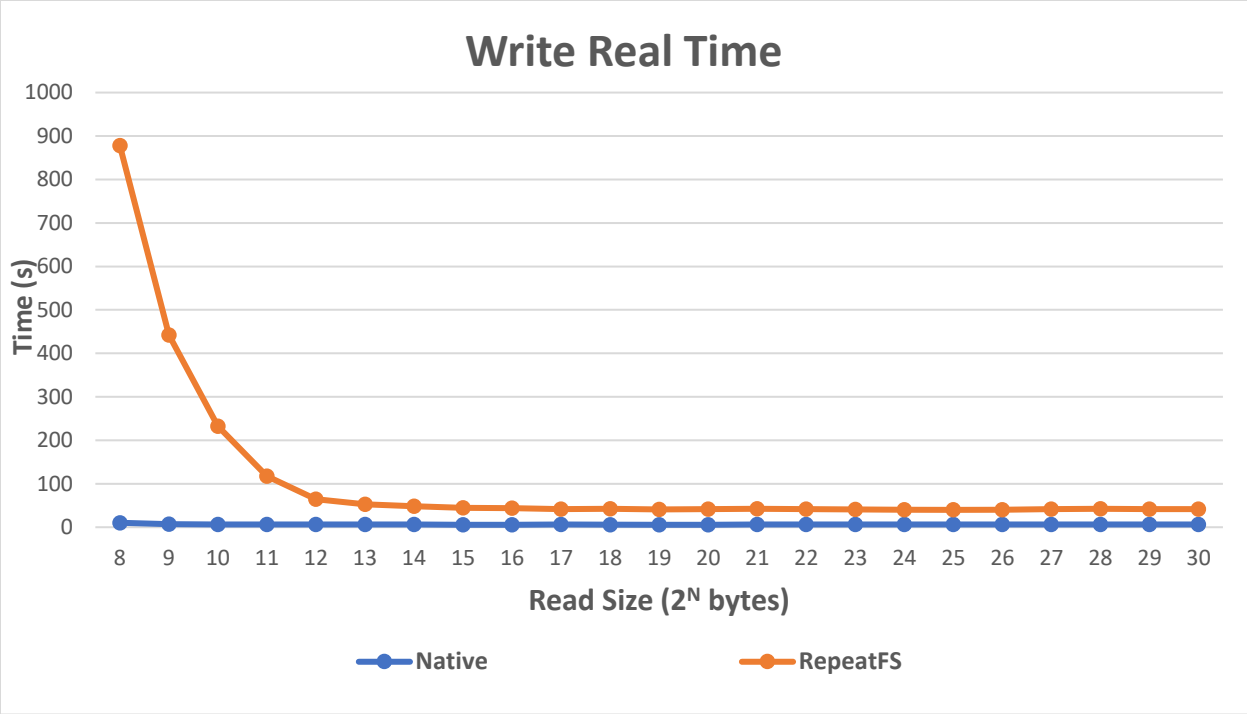


Figure 47: Write real time (1GB)

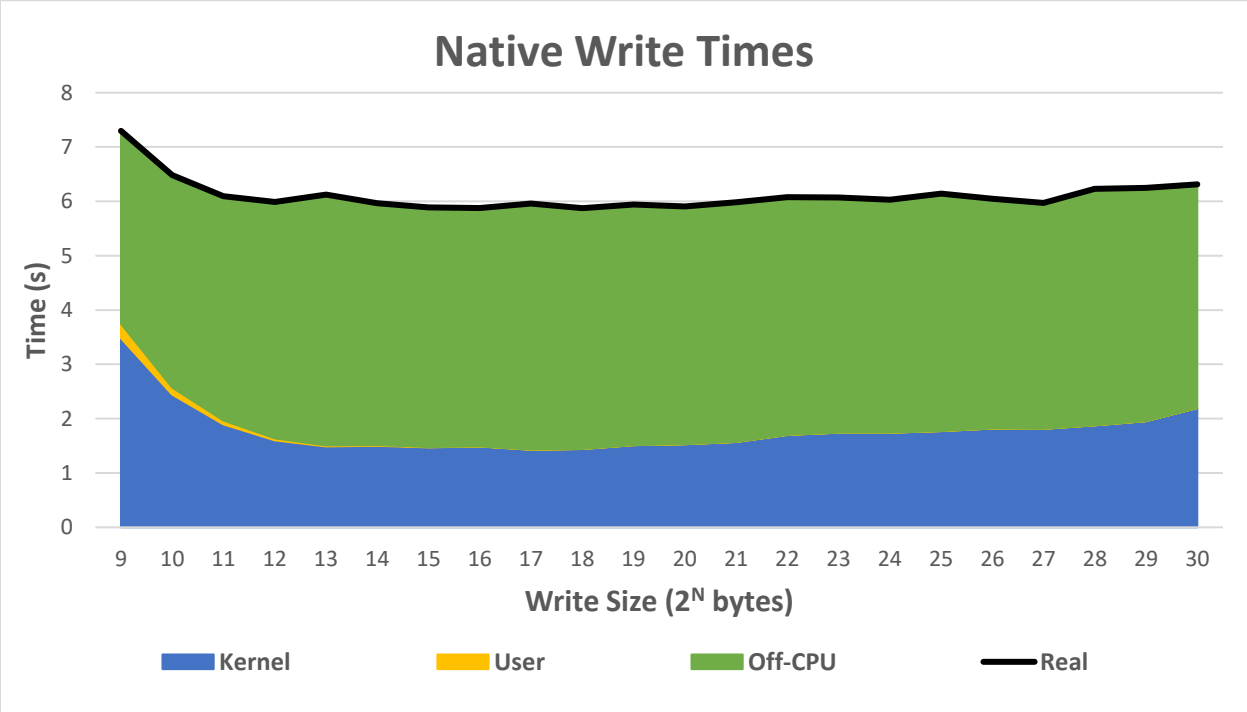


Figure 48: Native write times (1GB)

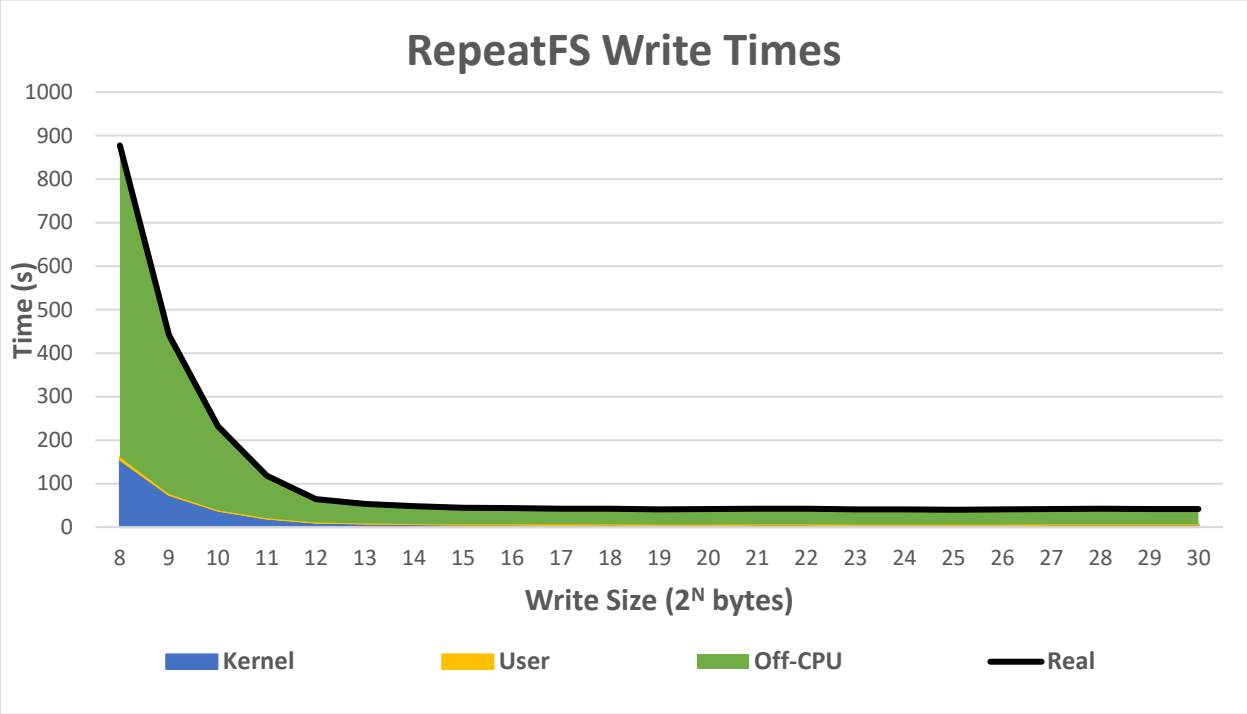


Figure 49: RepeatFS write times (1GB)

5.4 Virtual Dynamic Files

Unlike disk files, read operations targeting VDFs are not relayed to the underlying file system; instead, data for these calls is retrieved from the BCS (Section 4.2). If the BCS has not yet been populated with the data from the associated VDF processes, the data is retrieved from the POH (Section 4.5). Since these execution paths differ significantly in comparison to disk files, performance was evaluated independently for operations involving either of these subsystems. The BCS tests focus on the performance associated with reading data from the BCS memory cache in comparison to reading the same data from the underlying file system. The POH tests focus on the performance of transporting data from VDF processes to the BCS in different scenarios, such as VDF chaining, different output types, and concurrent access.

5.4.1 VDFs: BCS Memory Cache

For this series of tests, a VDF definition was created with the associated action of dumping the contents of the input file to the standard output stream via the “dd” command. To ensure only the BCS was evaluated and not the POH, the VDF was read in its entirety prior to each read test, resulting in the VDF data being loaded into the BCS memory cache. Like the previous tests, the read size was varied, although the range was modified to start at 4KB. Read sizes smaller than this resulted in unreasonably long processing time.

Like direct I/O disk reads (Figure 40), cached VDF reads have suboptimal throughput for read sizes less than 128KB (Figure 50). A comparison of their respective CPU times (Figures 44, 51) shows a similar relationship, with off-CPU times decreasing as read sizes near 128KB, and kernel times representing a relatively smaller portion of the overall real time. This is again likely due to FUSE related activity, as these patterns are not seen in read calls that only target the page cache or do not involve FUSE. For optimal read sizes, throughput remains steady at nearly 250MB/s. Although this is a lower rate than uncached disk file throughput for a single finalized VDF, this is mitigated for VDFs that are still receiving data from their processes. Since the POH provides pipe-like functionality, a series of VDFs may be chained together in a pipeline configuration that dramatically improves performance for multiple processes (Section 5.3.2). Additionally, once the VDF is finalized, it is eligible to be stored within the page cache, significantly improving performance.

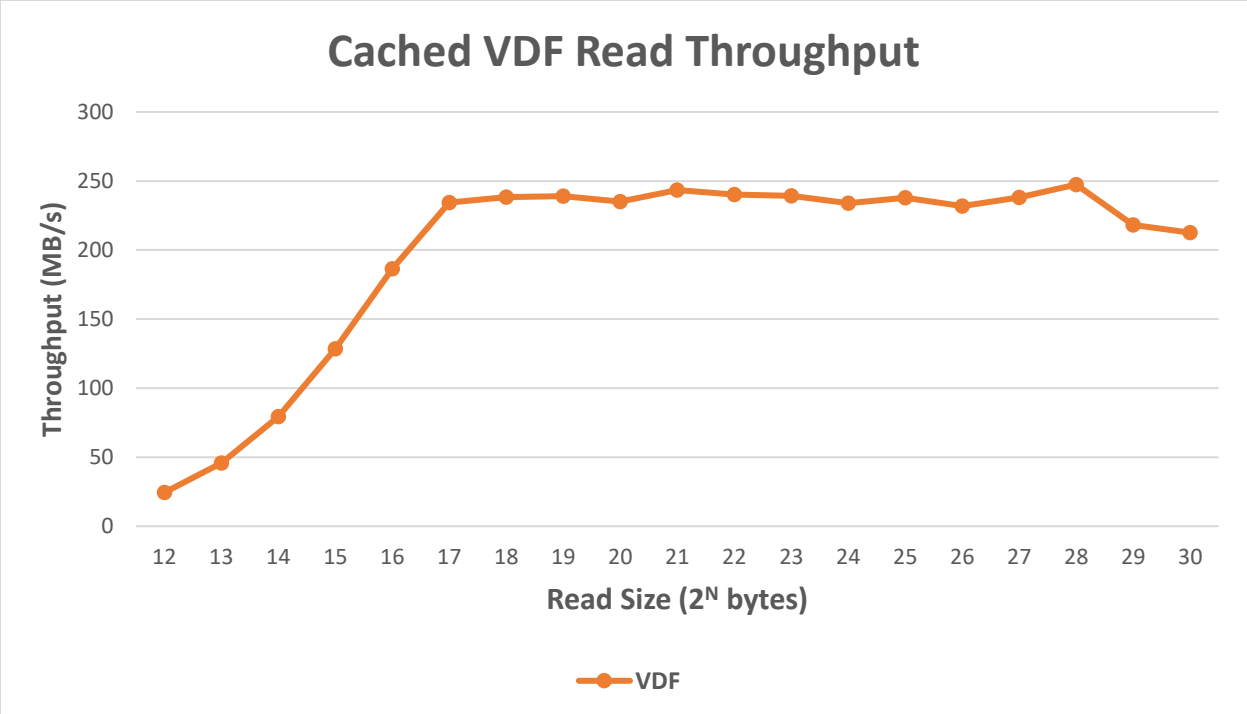


Figure 50: Cached VDF read throughput (1GB)

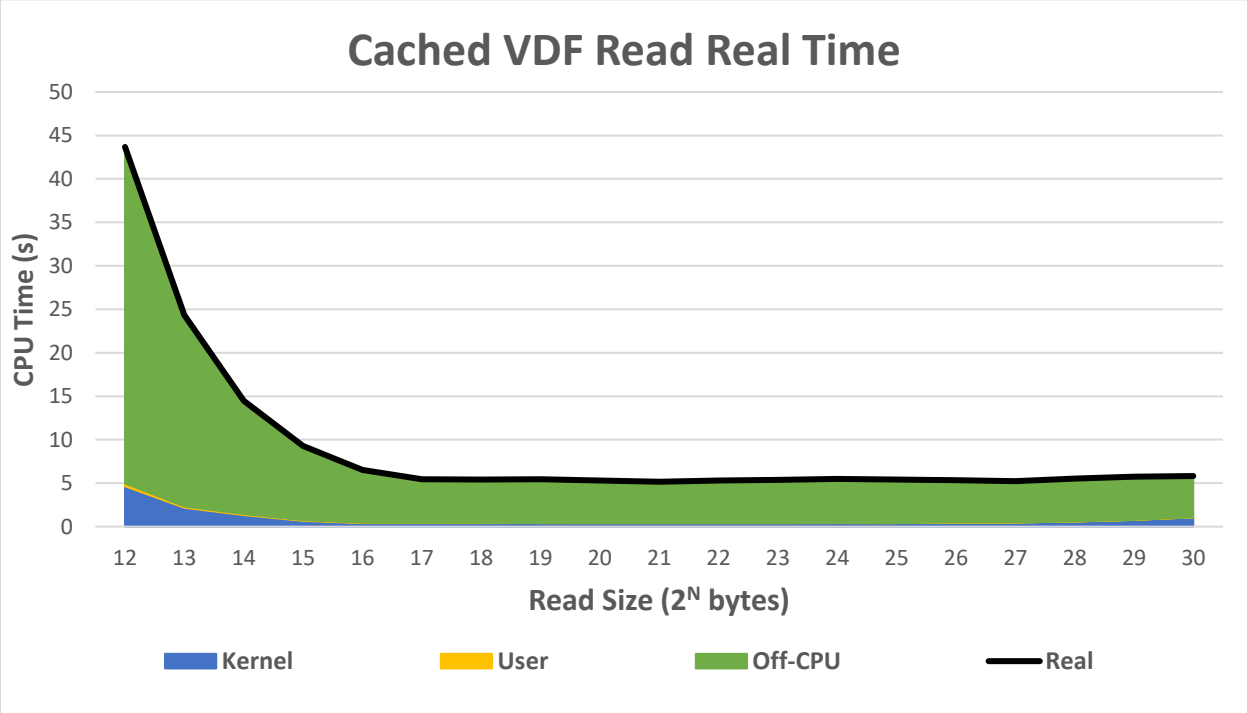


Figure 51: Cached VDF read real time (1GB)

5.4.2 VDFs: POH Pipeline Configurations

To examine the POH performance associated with VDFs in a chained, pipeline configuration, in comparison to the same workflow using temporary disk files (Figure 19), a simple informatics simulator was created to mimic processing data. The simulator reads an input file, introduces a configurable amount of delay per megabyte read, and writes the same number of bytes to an output file. A VDF definition was created to use this informatics simulator; the processing delay was set to ten seconds for every one megabyte read. The test iterated through an increasing number of these VDFs included in the chain, from a single VDF to ten chained VDFs. For each VDF in the chain, the comparison test executed the informatics simulator using temporary disk files on the native file system. A 10MB input file was used in all tests.

Although RepeatFS and the native file system both show a linear relationship between the number of processes executed and the real time to complete the entire workflow, RepeatFS significantly outperforms the native file system, with increasing reductions in real time based on the number of VDFs in the chain. These are expected results, as chained VDFs process data concurrently, while disk files process data sequentially (Figure 19). For disk-based workflows, the real time of each additional process writing to temporary files is added to the total execution time. For RepeatFS, most of the total execution time is based on the slowest process, with a small propagation delay introduced for each additional VDF in the chain.

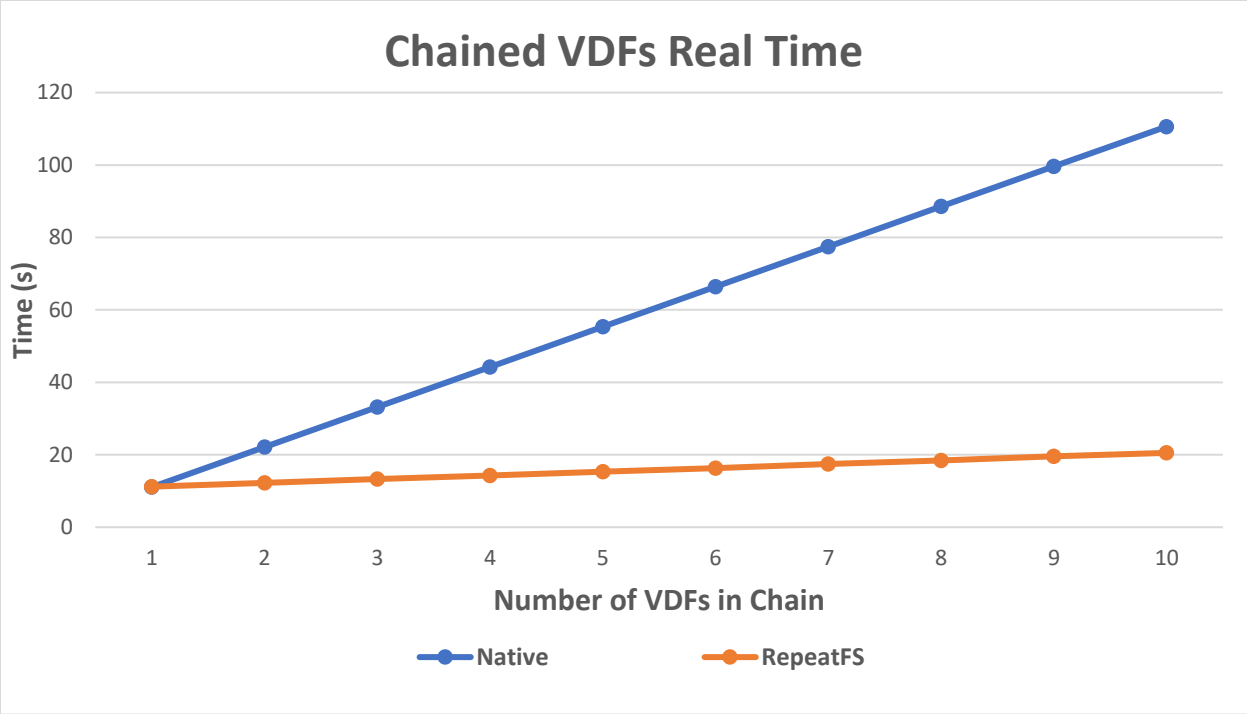


Figure 52: Chained VDFs real time

5.4.3 VDFs: POH Output Types

The POH supports receiving process output from file writes via the FIOB (Section 4.6) and from standard stream writes via the SIOB (Section 4.7). To evaluate the performance of the FIOB and SIOB separately, two VDF definitions were created. The first definition's action was configured to dump the contents of the input file to a disk file via the "dd" command. The second action was configured to dump the contents of the input file to the standard output stream. The "dd" commands were configured to use read sizes of 64KB and 1MB, respectively, to ensure optimal performance for each write method. Like the BCS test (Section 5.3.1), the test's read size was incrementally increased, starting at 4KB. Prior to the start of each test, the RepeatFS API was instructed to remove the VDF's contents from both BCS caches, ensuring the data was only retrieved from the POH.

For all read sizes, standard stream output throughput was always higher than disk file output, with a difference of approximately 60MB/s for optimal read sizes (Figure 48). This difference is further illustrated in the off-CPU times, in which disk file output was consistently twice that of standard stream time (Figures 54-56). This doubling in time can likely be attributed to the second FUSE call involved in disk file output. While both output types involve a first FUSE call made by the user process performing a read on the VDF, standard stream output is directly fetched by the SIOB from the VDF process. VDF processes writing to disk files instead issue a *write* call that is sent to the VFS and directed to FUSE and RepeatFS. As previously discussed (Sections 5.2.5, 5.3.1), each call to FUSE impacts the off-CPU time, and this pattern holds in this scenario as well. Given a command that supports writing output to either a file or the standard output stream, users creating VDF definitions using these commands should configure the program to write to streams for optimal performance.

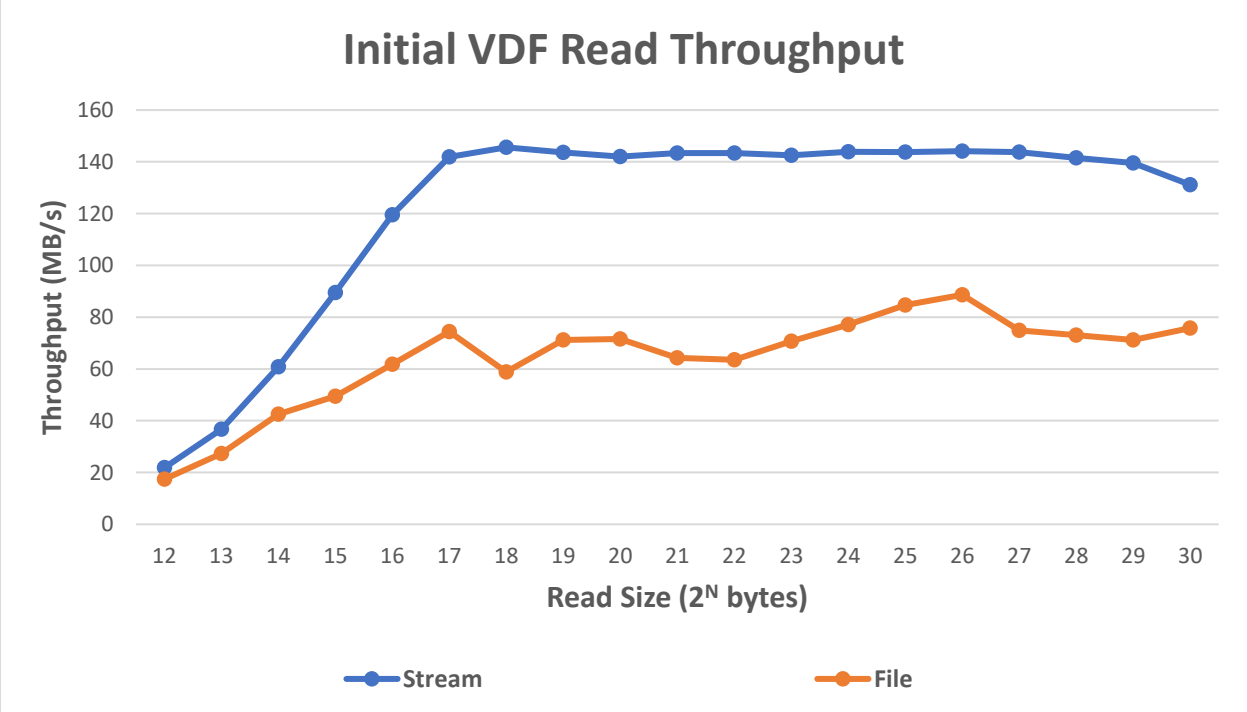


Figure 53: Initial VDF read throughput (1GB)

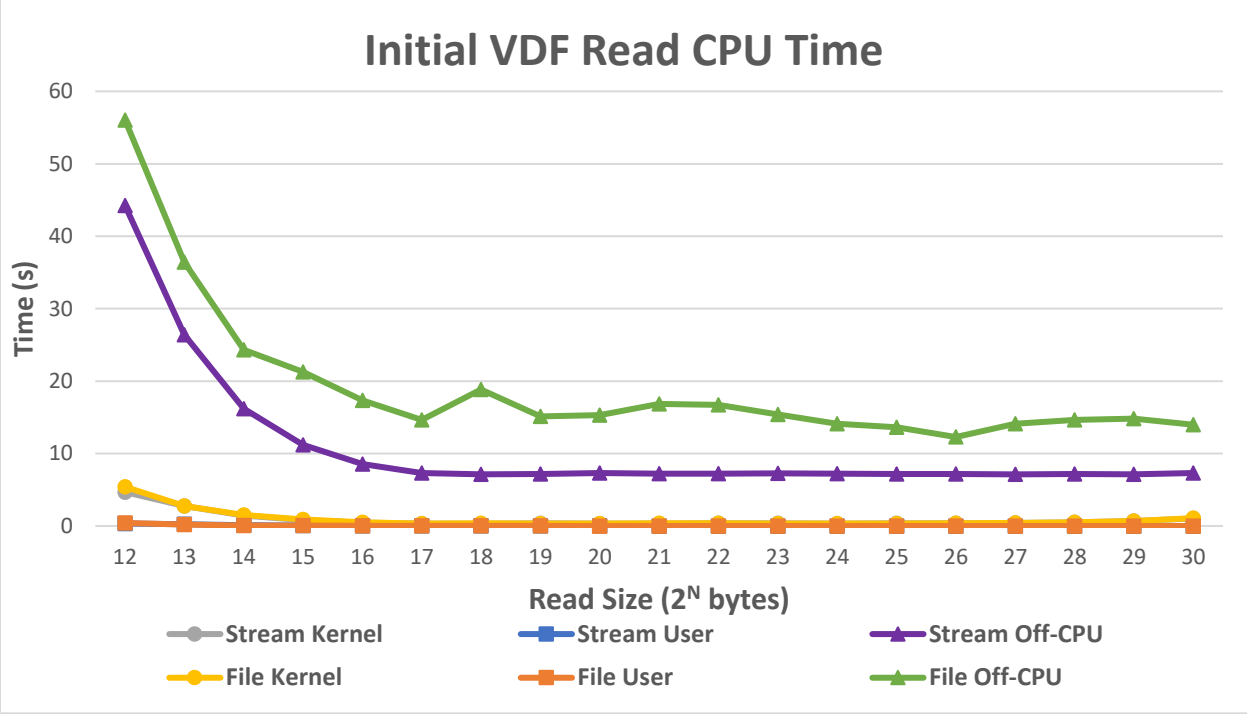


Figure 54: Initial VDF read CPU time (1GB)

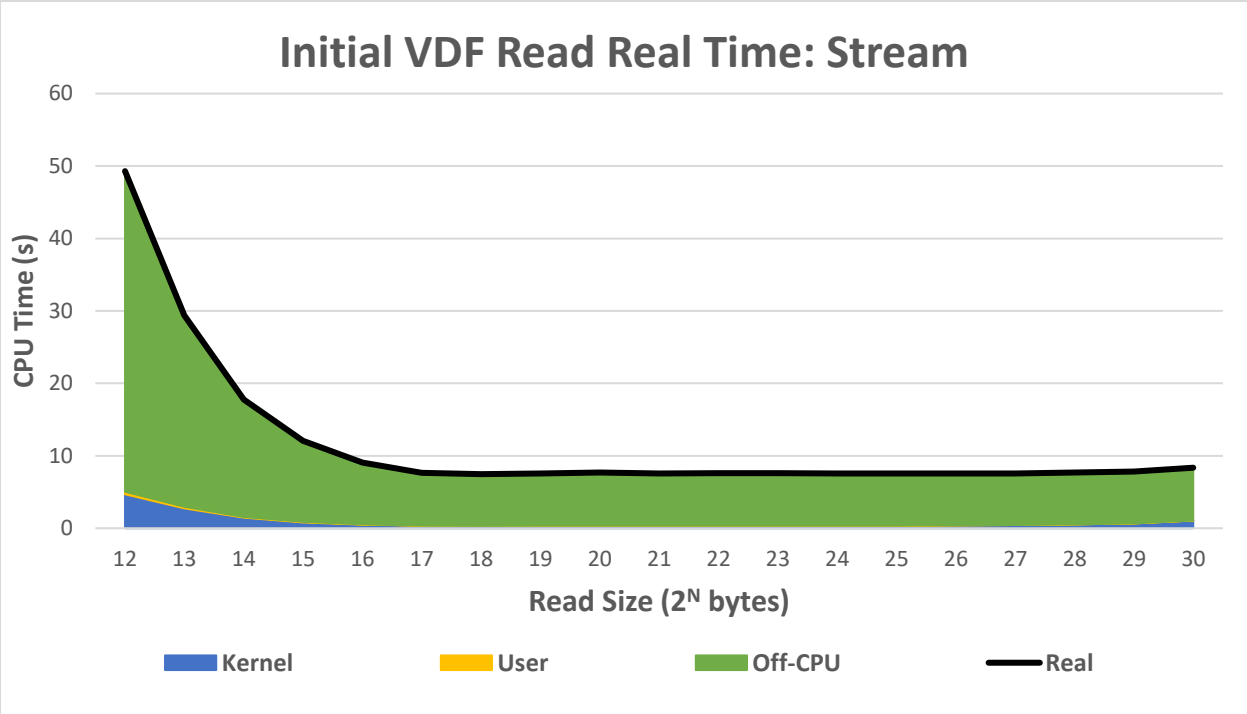


Figure 55: Initial VDF read real time (1GB): stream

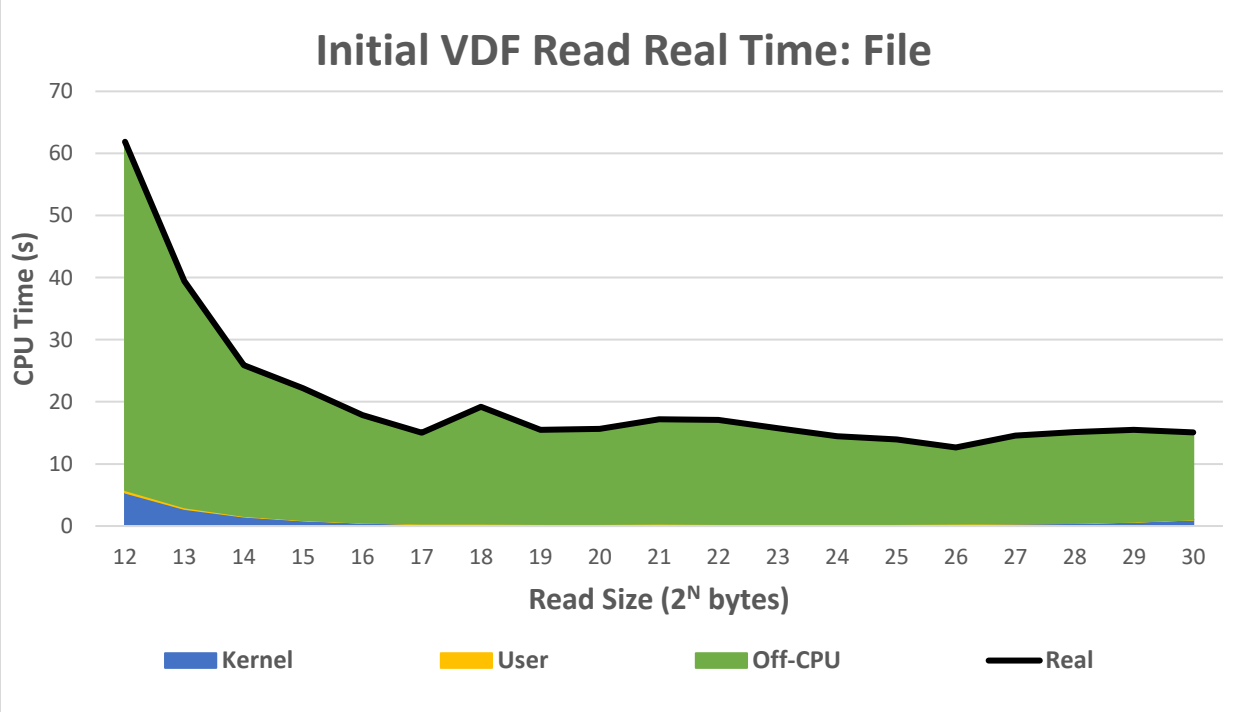


Figure 56: Initial VDF read real time (1GB): file

5.4.4 VDFs: BCS/POH Concurrent Reads

The BCS provides a prioritized waiting queue for concurrent read operations (Section 4.4), in which reads targeting blocks earlier in the VDF are serviced prior to reads targeting blocks later in the VDF. To test concurrent read performance, four different scenarios were employed in order to establish trends associated with the starting read position. Each scenario involved two or more of the following read positions: an "early stage" that starts at the first quarter of the file, a "mid stage" that starts in the middle of the file, and a "late stage" that starts at the third quarter of the file. Regardless of the starting position, all stages read through the remainder of the VDF. Each concurrent combination of two stages was evaluated, as was a final scenario involving all three stages. For every scenario, all reads were initiated simultaneously in parallel. Every scenario was performed twice, first on an uncached VDF not present in the BCS, then on a finalized VDF with

all blocks present in the BCS memory cache. While the first test also evaluats the performance of the POH fetching data from the VDF process, the second only tests the BCS. Like previous tests, each scenario was repeated ten times to remove jitter.

For every scenario tested, although performance of uncached VDFs is lower than cached VDFs, the trends between read stages for each scenario mostly match, with only small differences in the "mid stage" throughput in the mid/late scenario (Figures 57, 58). This illustrates that though throughput decreases while the BCS waits for additional data from the POH, the duration of this wait itself does not vary in response to small numbers of concurrent reads. Therefore, no significant differentiation need be made during analysis between cached and uncached reads.

Since each stage reads through the remainder of the file regardless of starting position, it is expected that all stages in a particular scenario will take an identical amount of real time to perform all required reads. All concurrent reads are initiated at the same time, and mid/late stage reads block until early stage reads have arrived at their position (Figure 22). In this way, they are inactive until synchronized at the same position and they all complete at the same time. Therefore, the total real time is simply the real time of the earliest operation. This is confirmed in all scenarios (Figures 59, 60); the real times of all stages in a scenario always match, and reads starting at positions closer to the beginning of the VDF increase overall real time.

Since all stages read until the end of the file, the early stage reads more total data than the mid stage, which reads more total data than the late stage. Given that the total real time involved per scenario is the same for all stages, it is expected that throughput will be highest for earlier stages and lowest for later stages. This is confirmed in all scenarios (Figures 57, 58); the throughput of a later stage is always lower than the earlier stage. Additionally, when comparing the same stage across different scenarios, throughput is affected by the amount of time two stages actively perform

reads concurrently. For example, early stage throughput for “Early, Mid” is approximately 130 MB/s, while early stage throughput for “Early, Late” is approximately 160 MB/s. Since the early stage begins at the first quarter of the file, and the mid stage begins in the middle of the file, 50% of the file is read concurrently in the “Early, Mid” scenario, while only 25% of the file is read concurrently in the “Early, Late” scenario. For the “Early, Late” scenario, 75% of the file is read by the early stage with no resource contention, improving overall throughput. This same pattern is seen for all other scenarios and stages.

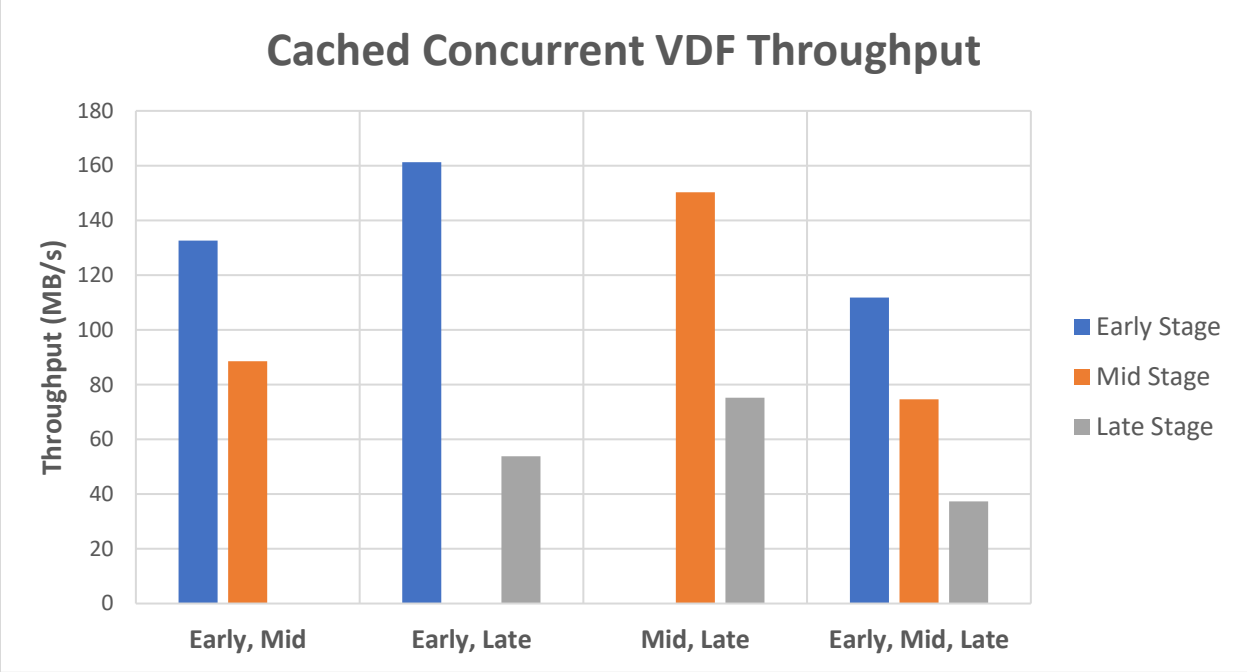


Figure 57: Cached concurrent VDF throughput

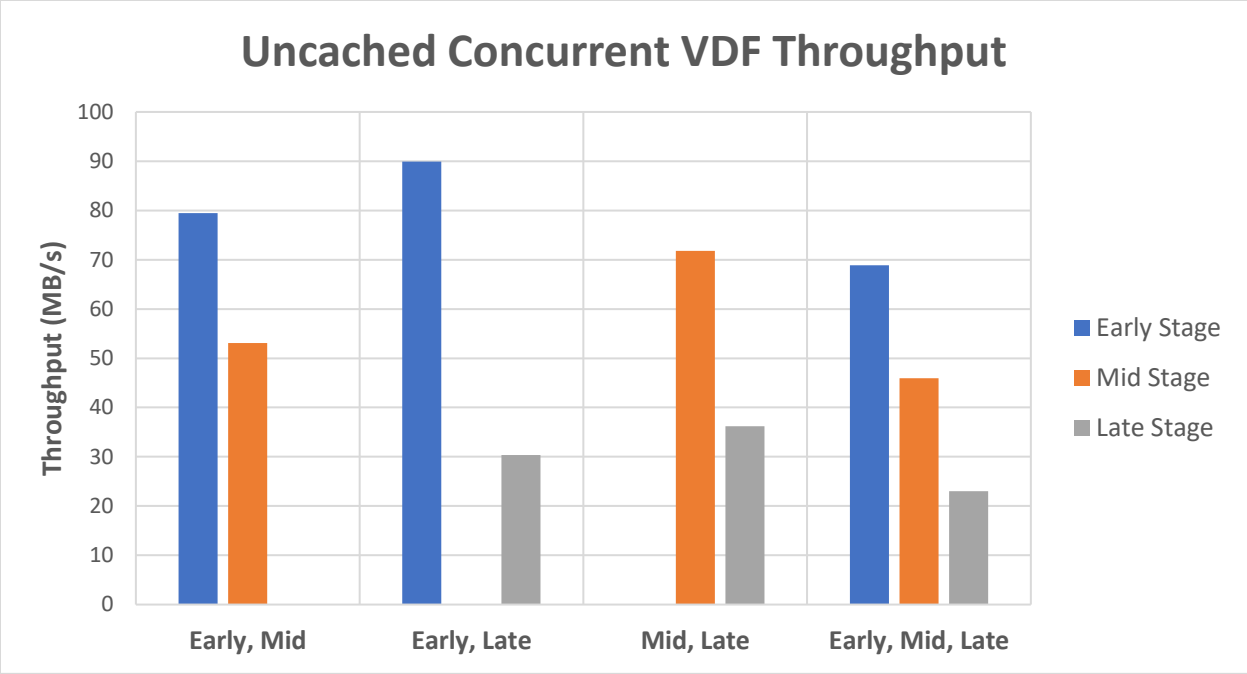


Figure 58: Uncached concurrent VDF throughput

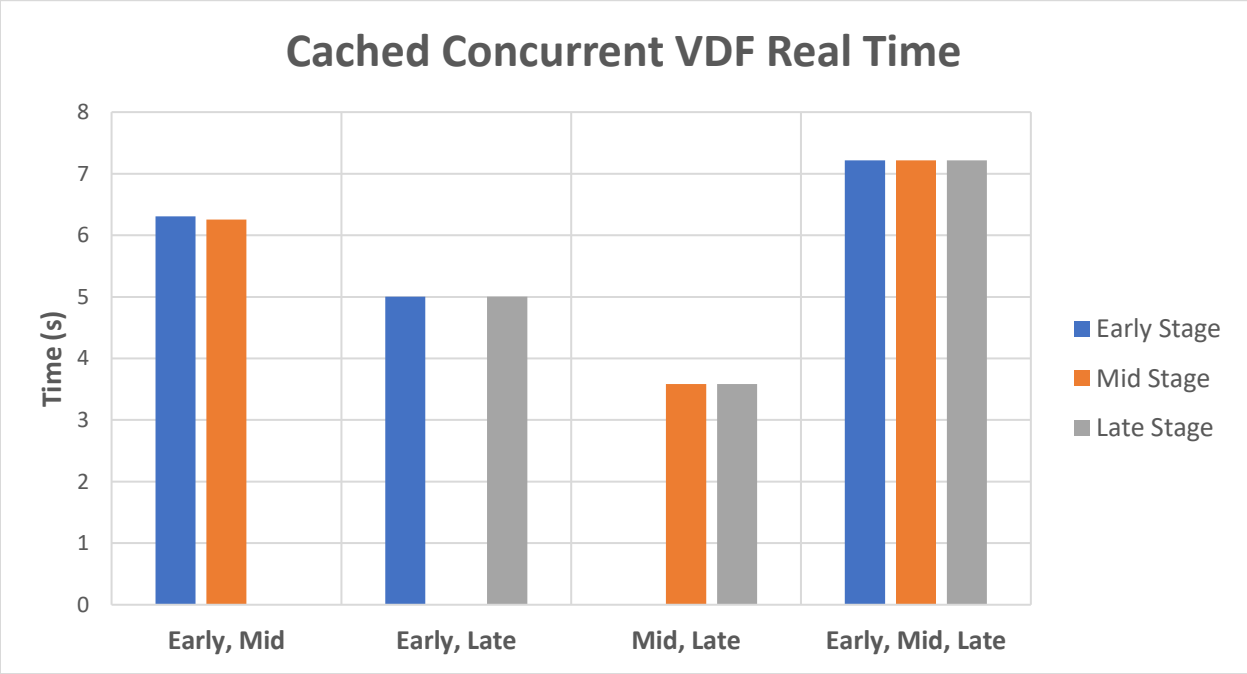


Figure 59: Cached concurrent VDF real time

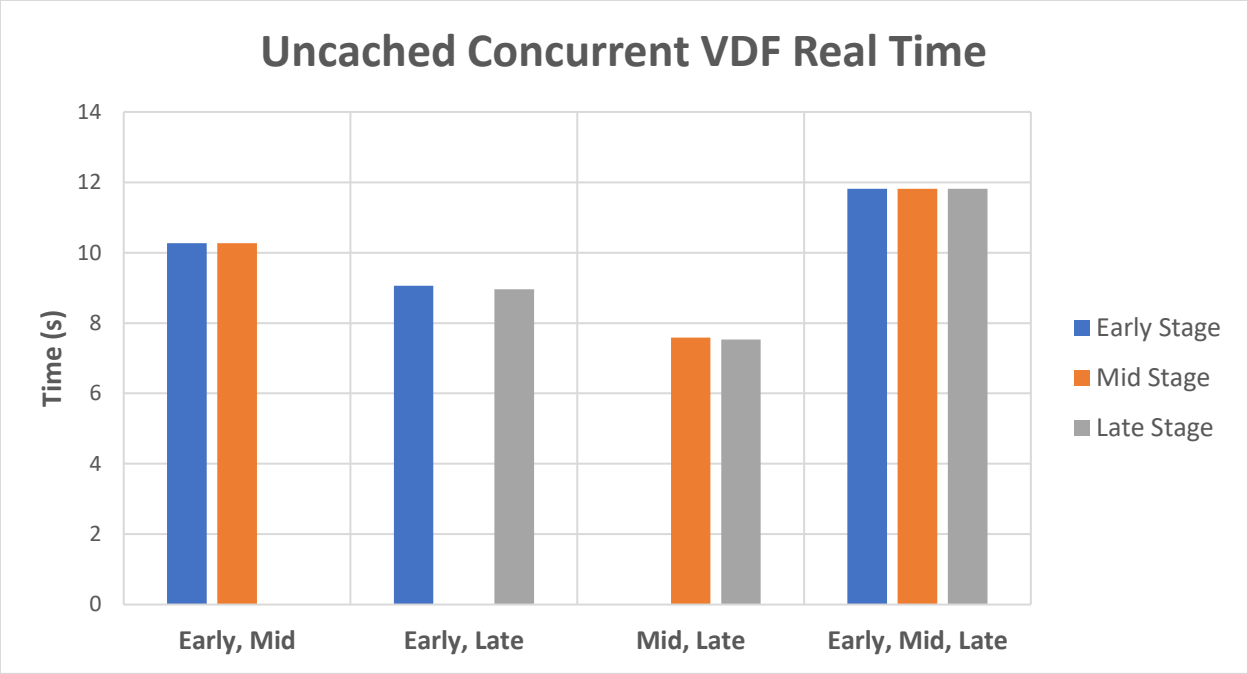


Figure 60: Uncached concurrent VDF real time

5.5 Performance Conclusion

Across all the core functionality read tests, RepeatFS had almost no negative impact on performance, only increasing the total read time by two seconds for a 1GB file. Since the runtime of analysis pipelines is often days, increasing this total time by seconds or minutes is negligible. Although RepeatFS does perform write operations approximately seven times slower than native writes, this is due to Fusepy's lack of page cache writeback support and not a limitation of RepeatFS itself. Analysis pipelines are often heavier on reads than writes as well, ameliorating this performance penalty. Writeback support may also be added to Fusepy by the developers in the future.

Pipeline workflows are significantly improved using VDFs instead of normal disk files. Our test pipeline consisting of identical sequential operations performed with linear time complexity when reading from disk files, while the same pipeline performed with nearly constant

time complexity when reading from VDFs. This difference in the order of complexity, coupled with the effectiveness of concurrent read prioritization, indicates VDFs offer a major performance improvement in larger, parallel, or more complex analysis workflows.

CHAPTER 6

CONCLUSION

6.1 Summary

Reproducibility is a significant struggle across many scientific disciplines. Computational analysis plays a major role in this difficulty, as seemingly identical informatics methods often yield different results. This failure to replicate is often rooted in the vast number of reasons a software package might behave differently from one execution to another, including version mismatches, reference databases used, different parameter values and combinations, environment variables, etc. Some causes, such as library versions or system configurations, may even be hidden to the user, further increasing the risk of irreproducibility. Current software solutions that attempt to address these computational analysis problems are limited and fail to consistently provide workflow transparency or ensure replicability. These issues stem from requiring the user to manually manage the provenance associated with files and processes, either directly or programmatically, and any manually performed task is ultimately subject to failure.

This research addresses problems in informatics reproducibility at both a conceptual and practical level. Chapter 1 introduces the RepeatFS methodology and two models that promote reproducibility by automating the process of managing provenance. The reactive model defines requirements for recording file and process provenance data through a notification mechanism provided by the operating system. Although this model is sufficient for providing transparency, replication, and verification of informatics processes, it does not guarantee uniformity between multiple executions of a non-replicated task. The proactive model provides this feature by defining the additional requirement of a predefined informatics workflow. This increases reproducibility by

ensuring that commonly performed tasks are always executed in an identical manner. These two models are implemented in RepeatFS, a file system that receives notifications of file operations by intercepting system calls. It adheres to the reactive model by recording all operations within a database and adheres to the proactive model through VDFs – virtual files that are created through a predefined workflow.

Chapter 2 documents the core functionality of the RepeatFS implementation. RepeatFS is written in Python and uses the FUSE file system framework. It relays system calls targeting disk files to the underlying file system and internally handles system calls targeting VDFs (Chapter 4). By intercepting system calls, RepeatFS is able to perform actions for each file system operation, including recording provenance. Reads are also compatible with the page cache, greatly increasing performance. RepeatFS management and control are provided to the user via an API system and accompanying client. These allow the user to initiate file replication or shutdown a RepeatFS instance. Lastly, chapter 2 also outlines all available command line arguments and configuration options.

Since an accurate record of provenance is necessary for both the reactive and proactive models, all applicable system calls targeting a disk file or VDF are recorded. Chapter 3 provides details of how the information is stored, as well as methods for reconstructing and utilizing this data. Process and file attributes related to each system call are stored in a relational database. These entries are used by RepeatFS to reconstruct a directed graph representing the read and write operations between processes and files. This graph is then used for multiple practical applications, including visualization via a dynamically generated web application, replicating the original workflow to recreate the result file, and verifying that the replicated workflow does not deviate from the original workflow.

Although a record of provenance is essential for the RepeatFS proactive model, it is not sufficient, and VDFs fulfill the remainder of the requirements. Chapter 4 describes the overall functionality of VDFs, and how they automatically execute a series of actions when accessed. Not only is this predefined workflow necessary for the proactive model, but it also provides an easy way for users to repeatedly perform common tasks in a uniform fashion. Output from VDF processes is saved in the BCS, a system that stores and manages data in memory and disk caches. Concurrent access to the BCS is prioritized for optimal performance given typical use case scenarios. The BCS retrieves waiting data from the POH, which utilizes the FIOB and SIOB to buffer data from running processes. Chapter 4 also outlines each system call that contains additional routines for handling functionality unique to VDFs.

Chapter 5 details a performance evaluation of each portion of RepeatFS. Throughput, user mode time, kernel mode time, off-CPU time, and real time are measured in comparison to read and write size in a number of different scenarios. These include presence or absence of the file in the page cache, and direct I/O. Each subsystem in RepeatFS is isolated and tested, including the provenance system, the BCS, the POH, and the priority queue. For each system and scenario, trends are noted, and optimal read sizes are identified.

6.2 Future Work

Although fully functional and containing many features, RepeatFS has some opportunities for improvement in future versions. Its degraded write performance is due to a lack of writeback support in Fusepy (Section 2.3). Since this is an open-source module, and the underlying libfuse library does support page cache writeback, there is opportunity to develop a Fusepy patch to make use of this functionality. This is also mitigated by informatics pipelines typically performing more

reads than writes. Additionally, due to limitations in Python's standard file handling, RepeatFS does not support direct I/O passthrough to the underlying file system (Section 5.2.4). This could be added by converting all calls to Python's standard file functions to those provided by the mmap module. Since this is a fundamental change in how file operations are initiated and processed, it would require repeating all performance evaluations to ensure this small addition in functionality does not negatively impact overall efficiency and usability. Lastly, RepeatFS is currently designed to be run as a single instance with limited support for shared and distributed storage. Multiple instances may run on different nodes and concurrently save provenance information to the same database. However, RepeatFS uses the local system time and does not employ any type of method for calculating partial ordering of events, such as Lamport or vector clocks. Even with synchronized clocks, users who wish to use RepeatFS across multiple systems do so with an increased provenance uncertainty for I/O events occurring close in time.

6.3 Final Conclusion

RepeatFS provides a multifaceted solution to a significant problem that is present across all scientific disciplines. To address informatics-based reproducibility issues, we designed two conceptual models and implemented multiple novel methods that are not available in any other provenance management system: efficiently recording provenance at the file system level, storing this data in a SQL database, iteratively reconstructing causally related directed graphs using this database, and providing VDFs that simultaneously act as pipes and files. We also created complementary features important for reproducing results, such as visualization, replication, and verification. Most importantly, RepeatFS provides all of this functionality automatically and does not require the user to modify their methods or write additional scripts. This allows researchers to

easily integrate RepeatFS into their analysis workflows without introducing errors or irreproducibility.

We also performed considerable optimization to ensure RepeatFS does not noticeably increase processing time in most pipelines, as demonstrated in the performance evaluation. This was especially challenging given the internal complexity of RepeatFS but an important practical consideration since many analyses process large amounts of data. Furthermore, VDFs increase the efficiency of multistep pipelines by an order of time complexity in comparison to disk files, a significant ability unique to RepeatFS. Although write performance is currently impacted, this may be resolved by a future update to the Fusepy module. This combination of functionality and performance clearly demonstrates RepeatFS is a powerful and novel method of promoting informatics reproducibility across many scientific disciplines.

BIBLIOGRAPHY

Afgan, E., Baker, D., Batut, B., van den Beek, M., Bouvier, D., Cech, M., … Blankenberg, D. (2018). The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Research*, *46*(W1), W537–W544. https://doi.org/10.1093/nar/gky379

Anaconda Software Distribution (Version 4.4.8). (2020). Anaconda Inc. Retrieved from https://anaconda.com

awesomeopensource.com. (2021). The Top 73 Fuse Open Source Projects. Retrieved from The Top 73 Fuse Open Source Projects website: https://awesomeopensource.com/projects/fuse

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature*, *533*(7604), 452–454. https://doi.org/10.1038/533452a

Bolyen, E., Rideout, J. R., Dillon, M. R., Bokulich, N. A., Abnet, C., Al-Ghalith, G. A., … Caporaso, J. G. (2018). *QIIME 2: Reproducible, interactive, scalable, and extensible microbiome data science* (No. e27295v2). PeerJ Inc. https://doi.org/10.7287/peerj.preprints.27295v2

Bray, T. (2017). *The JavaScript Object Notation (JSON) Data Interchange Format* (No. RFC8259; p. RFC8259). RFC Editor. https://doi.org/10.17487/RFC8259

Coiera, E., Ammenwerth, E., Georgiou, A., & Magrabi, F. (2018). Does health informatics have a replication crisis? *Journal of the American Medical Informatics Association : JAMIA*, *25*(8), 963–968. https://doi.org/10.1093/jamia/ocy028

Davis-Turak, J., Courtney, S. M., Hazard, E. S., Glen, W. B., da Silveira, W. A., Wesselman, T., … Hardiman, G. (2017). Genomics pipelines and data integration: Challenges and

opportunities in the research setting. *Expert Review of Molecular Diagnostics*, *17*(3), 225–237. https://doi.org/10.1080/14737159.2017.1282822

Docker (Version 19.03.8). (2020). Docker Inc. Retrieved from https://docker.com

Finney, B. (2021). Python-daemon (Version 2.3.0). Retrieved from https://pagure.io/python-daemon/

Garijo, D., Kinnings, S., Xie, L., Xie, L., Zhang, Y., Bourne, P. E., & Gil, Y. (2013). Quantifying Reproducibility in Computational Biology: The Case of the Tuberculosis Drugome. *PLoS ONE*, *8*(11). https://doi.org/10.1371/journal.pone.0080278

Gooch, R. (2005). Overview of the Linux Virtual File System. Retrieved from https://www.kernel.org/doc/html/latest/filesystems/vfs.html

Goodstadt, L. (2010). Ruffus: A lightweight Python library for computational pipelines. *Bioinformatics*, *26*(21), 2778–2779. https://doi.org/10.1093/bioinformatics/btq524

Graphviz (Version 2.47.1). (2021). AT&T Labs Research. Retrieved from https://graphviz.org/

Hagberg, A., Schult, D., Renieris, M., & Millman, J. (2021). PyGraphviz (Version 1.7). Retrieved from https://pygraphviz.github.io/

Henson, J., Tischler, G., & Ning, Z. (2012). Next-generation sequencing and large genome assemblies. *Pharmacogenomics*, *13*(8), 901–915. https://doi.org/10.2217/pgs.12.72

Hipp, R. D. (2020). SQLite (Version 3.31.1). Retrieved from https://www.sqlite.org/index.html

Hollenbeck, S., Rose, M., & Masinter, L. (2003). *Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols* (No. RFC3470; p. RFC3470). RFC Editor. https://doi.org/10.17487/rfc3470

Honles, T. (2018). Fusepy (Version 3.0.1). Retrieved from https://github.com/fusepy/fusepy

Kanwal, S., Khan, F. Z., Lonie, A., & Sinnott, R. O. (2017). Investigating reproducibility and tracking provenance – A genomic workflow case study. *BMC Bioinformatics*, *18*. https://doi.org/10.1186/s12859-017-1747-0

Kim, Y.-M., Poline, J.-B., & Dumas, G. (2018). Experimenting with reproducibility: A case study of robustness in bioinformatics. *GigaScience*, *7*(7). https://doi.org/10.1093/gigascience/giy077

Köster, J., & Rahmann, S. (2012). Snakemake—A scalable bioinformatics workflow engine. *Bioinformatics*, *28*(19), 2520–2522. https://doi.org/10.1093/bioinformatics/bts480

Lewis, J., Breeze, C. E., Charlesworth, J., Maclaren, O. J., & Cooper, J. (2016). Where next for the reproducibility agenda in computational biology? *BMC Systems Biology*, *10*(1), 52. https://doi.org/10.1186/s12918-016-0288-x

MariaDB Server (Version 10.5.9). (2021). MariaDB Corporation. Retrieved from https://mariadb.org/

Microsoft SQL Server (Version 2019). (2019). Microsoft. Retrieved from https://www.microsoft.com/en-us/sql-server

MySQL Server (Version 8.0.24). (2021). Oracle. Retrieved from https://www.mysql.com/

Oracle Database (Version 19c). (2019). Oracle. Retrieved from https://www.oracle.com/database/

PostgreSQL Server (Version 12.2). (2021). PostgreSQL Global Development Group. Retrieved from https://www.postgresql.org/

Sadedin, S. P., Pope, B., & Oshlack, A. (2012). Bpipe: A tool for running and managing bioinformatics pipelines. *Bioinformatics (Oxford, England)*, *28*(11), 1525–1526. https://doi.org/10.1093/bioinformatics/bts167

SVG Working Group. (2021, April). *Scalable Vector Graphics (SVG) 2*. Retrieved from https://svgwg.org/svg2-draft/

Szeredi, M., & Rath, N. (2019). FUSE (Filesystem in Userspace) (Version 2.9.9). Retrieved from https://github.com/libfuse/libfuse

Wang, G., & Peng, B. (2019). Script of Scripts: A pragmatic workflow system for daily computational research. *PLoS Computational Biology*, *15*(2). https://doi.org/10.1371/journal.pcbi.1006843