Summer 7-20-2023

Interposition based container optimization for data intensive applications

Rohan Tikmany
DePaul University, rtikmany@depaul.edu

INTERPOSITION BASED CONTAINER OPTIMIZATION FOR DATA INTENSIVE APPLICATIONS

BY

ROHAN TIKMANY

A THESIS SUBMITTED TO THE SCHOOL OF COMPUTING, COLLEGE OF COMPUTING

AND DIGITAL MEDIA OF DEPAUL UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE

DEPAUL UNIVERSITY

CHICAGO, ILLINOIS

2023

DePaul University
College of Computing and Digital Media

MS Thesis Verification

This thesis has been read and approved by the thesis committee below according to the requirements of the School of Computing graduate program and DePaul University.

Name: Rohan Tikmany

Title of dissertation: Interposition Based Container Optimization for Data Intensive Applications

Date of Dissertation Defense: 20th of July 2023

Prof. Tanu Malik

Advisor*

Dr. Ashish Gehani

1st Reader

Prof. Alexander Rasin

2nd Reader

** A copy of this form has been signed, but may only be viewed after submission and approval of FERPA request letter.*

Acknowledgements

I would like to extend my sincerest appreciation to those who have made this thesis possible through their direct and indirect support.

Firstly, profound gratitude is due to my advisor, Prof. Tanu Malik. Your unwavering guidance, insightful critiques, and relentless pursuit of excellence have fundamentally shaped this work. Your expertise and commitment to scholarly rigor have been invaluable to my research experience and personal growth.

I would also like to thank the members of my Thesis Committee, Dr. Ashish Gehani and Prof. Alexander Rasin. Your constructive feedback and knowledgeable perspectives have enriched the quality of this research. Your time and expertise were critical in bringing this work to fruition.

Special thanks are extended to my colleagues—Yuta Nakamura, Aniket Modi, and Dr. Moaz Reyad. Your collaboration and companionship throughout this process have been enlightening and motivating. Your contributions, both academically and personally, have significantly impacted the successful completion of this research.

I cannot conclude without expressing my heartfelt gratitude to my family and friends. Your unconditional love, unwavering support, and endless encouragement have been the backbone of my academic journey. It is your belief in me that has made this milestone achievable.

Thank you to all who have contributed to this endeavor. Your roles, whether large or small, have left an indelible mark on this work and on me.

List of Tables

List of Figures

Abstract:

Reproducibility of applications is paramount in several scenarios such as collaborative work and software testing. Containers provide an easy way of addressing reproducibility by packaging the application's software and data dependencies into one executable unit, which can be executed multiple times in different environments. With the increased use of containers in industry as well as academia, current research has examined the provisioning and storage cost of containers and has shown that container deployments often include unnecessary software packages. Current methods to optimize the container size prune unnecessary data at the granularity of files and thus make binary decisions. We show that such methods do not translate efficiently to scientific data files, where only a subset of data may be accessed across several files. In this thesis, we propose a method of looking at this problem at the granularity of bytes. Instead of keeping track of which files are accessed, we keep track of the portions of files accessed in the form of file offsets. This I/O lineage allows us to package only relevant parts of data files, significantly reducing the storage and sharing cost of containers. Results show the generality of our method across different data formats and reduction approximately equal to the amount of data fetched into memory.

Contents

Chapter 1

Introduction

1.1 Motivation

In the intricate landscape of data-intensive scientific research, collaboration and sharing of information have become fundamental yet challenging. Consider the case of a scientist named Alice, who works at NASA on the Global Precipitation Measurement (GPM) project [2]. Alice's work involves vast amounts of data, and she wants to share her findings with peers across the globe. Like many of her colleagues, she faces a common problem: how to share complex research work efficiently.

Containerization has emerged as a significant technological advancement to tackle this issue. A container can be likened to a virtual box, where everything needed to run a piece of software, including the code, system tools, libraries, and runtime, is packaged together. This ensures that the software will run the same way, regardless of where the container is deployed. Tools like Docker have made this concept a reality, enabling scientists like Alice to share their work effortlessly with minimal change in workflow.

However, Alice soon discovers a glaring problem with this approach. The size of these containers, especially when dealing with extensive datasets as in the GPM project, can be enormous. These unwieldy containers create a significant barrier to sharing and collaboration, turning what was initially a seamless solution into a complicated challenge.

The motivation to address this problem is clear: facilitating effective collaboration and dissemination of research work is key to scientific innovation and progress. That's where the concept of data debloating comes into the picture. Data debloating is the process of reducing the size of datasets by identifying and eliminating non-essential information. Unlike software debloating, which focuses on trimming unnecessary code from software, data debloating specifically targets the data itself, maintaining only the vital components required for computation.

1.2 Problem Background

The example below is a pseudocode outline of the visual comparison program referenced in chapter X.

```
/*
Function that takes an argument of a list of columns
Extracts those columns from the IMERG dataset and the GEOS5 dataset
and compares them using the CompareData function
*/
func visualComparison(listOfColumns)
    // Open the 2 files
    imergFile = Open("IMERG.HDF5")
    geos5File = Open("GEOS5.HDF5")

    // Read only the columns that have been passed in
    imergCols = ReadColumns(imergFile, listOfColumns)
    geos5Cols = ReadColumns(geos5File, listOfColumns)

    // Compare the columns from the 2 files
    CompareData(imergCols, geos5Cols)
```

This program addresses the challenge of handling large datasets when working with data-intensive applications. Specifically, it addresses the issue of sharing and disseminating research work which is often impeded by the size of the data involved. As an example, scientists working with satellite data from the NASA Global Precipitation Measurement (GPM) mission might want to share their work with their peers, but the datasets involved can be extremely large and unwieldy.

In the pseudocode we propose, the program takes as input a list of columns specified by the user, and only reads those columns from two HDF5 files - "IMERG.HDF5" and "GEOS5.HDF5". HDF5 is a high-performance data storage format optimized for storing large arrays of numerical data. In this case, each column represents a different measurement or parameter, and the user is interested only in a subset of them.

By focusing only on the columns of interest, the program avoids reading in large amounts of extraneous data. This selective reading significantly reduces the amount of data that needs to be loaded into memory, which can be beneficial for performance and can lower the system's memory requirements.

Once the specified columns have been read from both files, the program compares the corresponding rows visually by plotting graphs. This visualization allows users to quickly and easily compare the data in the selected columns from the two datasets.

The implications of this approach for data sharing and dissemination are significant. If scientists were to share the full datasets, they would be wasting substantial storage space, especially if the number of columns of interest is small. Moreover, the size of the data that needs to be shared would increase, making it more cumbersome to distribute. This is a problem that the concept of data debloating seeks to solve.

Data debloating refers to the process of reducing the size of a dataset by removing redundant or irrelevant data, while still maintaining the integrity of the data necessary for a particular analysis. In this context, debloating can be achieved by sharing only the columns that are relevant to the analysis. This would make it much more efficient to share these programs with other researchers.

There is a clear need for solutions that enable more efficient sharing of relevant data in data-intensive applications. As demonstrated by the pseudocode, selective reading of only the necessary data from large datasets is an effective approach to data debloating. This approach not only facilitates easier data sharing but also improves performance by reducing the amount of data that needs to be processed.

The example below shows the visualComparison program with writing to the datasets included

```
/*
Function that takes an argument of a list of columns
Extracts those columns from the IMERG dataset and the GEOS5 dataset
and compares them using the CompareData function
*/
func visualComparison(listOfColumns, columnA, columnB)
    // Open the 2 files
    imergFile = Open("IMERG.HDF5")
    geos5File = Open("GEOS5.HDF5")

    // Write a new column based on existing columns
    WriteNewColumn(imergFile, columnA, columnB)
    WriteNewColumn(geos5File, columnA, columnB)

    // Add newColumns to list of columns
    listOfColumns.Add(newColumn)

    // Read only the columns that have been passed in
    imergCols = ReadColumns(imergFile, listOfColumns)
    geos5Cols = ReadColumns(geos5File, listOfColumns)

    // Compare the columns from the 2 files
    CompareData(imergCols, geos5Cols)
```

Our enhanced program reads columns A and B from the datasets ”IMERG.HDF5” and ”GEOS5.HDF5”, processes the data, and writes the results back into the dataset as a new column, C. Subsequently, the program reads columns B and

C for further processing. This means that the program is not only extracting information from the datasets but is also updating them with new data.

In this scenario, it may initially appear that columns A, B, and C are all necessary for correct execution. However, upon closer examination, we realize that column C can be recalculated from columns A and B. Therefore, sharing column C becomes redundant. In essence, only columns A and B are required to recreate the program, as column C can be derived from them.

The task of data debloating becomes much more complicated when the program modifies existing data, as opposed to merely appending new data. For example, if column B were to be overwritten instead of creating a new column C, the original data in column B would be lost unless a backup of the original data is created before the write operation.

Addressing this challenge requires a system capable of actively tracking instances where data that has previously been read is subsequently modified or overwritten. This system must create real-time backups of the destructively changed data during execution, ensuring that the necessary data is available for recovery post-execution. By doing so, we can rebuild the data in its original form, ensuring identical re-execution.

This process is complex, as it requires intricate tracing of data dependencies and real-time monitoring of data modifications during execution. The data that is written often depends on the data that was previously read, making the identification of the minimal necessary data for sharing intricate. The example with columns A, B, and C illustrates this challenge, as the inclusion of the written data in the shared dataset is unnecessary if it can be recreated from the original data.

In conclusion, effective data debloating is crucial in data-intensive applications, as it facilitates data sharing and optimizes resource utilization. However, the presence of both reading and writing operations, especially when modifying existing data, adds complexity to the identification of the minimum necessary data for sharing. We need a system that tracks destructively changed data and creates real-time backups during execution, ensuring post-execution recovery of the original data for guaranteed re-execution. The intricacies of data debloating in data-intensive scientific applications underscore the importance of developing sophisticated methods for this purpose.

Chapter 2

Problem Statement and Proposed Methodology

2.1 Problem Statement

The primary goal of this thesis is to identify a minimal subset of a given dataset that still retains the capacity to reproduce the results of a specific program execution. Formally, let us consider a program P, a dataset D, parameters T, and results R. Our aim is to find a subset D' of D such that executing P on D' with parameters T yields the same results R as running P on D with parameters T. This problem can be mathematically formulated as finding $D' \subseteq D$ such that:

$$P(T, D') = P(T, D) = R$$

Addressing this problem involves several key elements:

1. **Execution Analysis**: The first step involves analyzing the original dataset D, which could comprise multiple components, including files, datasets, or databases. It is essential to develop techniques that can accurately identify the critical components within the dataset used dur-

ing the program's execution. Such an analysis requires understanding the program's data provenance to assess the relevance of all sections of the dataset D. An integral part of this analysis is recognizing and tracking derived data produced during the program's execution, as well as creating backups of original data that are modified or overwritten. This ensures that the original data can be restored post-execution for accurate reproduction of the program's results.

2. **Data Carving**: Based on the results of the dataset analysis, the next step is to debloat the original dataset D by removing redundant or unnecessary components. This leads to a debloated dataset D' that reduces the overall size of the container image. It is crucial to ensure that the debloated dataset still contains all the necessary components for accurate program execution (P).

3. **Re-execution**: After debloating, the application must be re-executed in a manner that is transparent to the user and requires minimal user intervention. This re-execution should produce results consistent with the original execution. If any divergence is detected, it is vital to inform the user of the discrepancy.

4. **Container Verification**: Lastly, it is important to assess the effectiveness of the data debloating techniques in terms of container completeness and system performance. The reduced container image size should result in improved resource utilization, including reduced disk space consumption, while still producing consistent results. Furthermore, any potential trade-offs in terms of execution time or memory usage must be measured and considered.

2.2 Methodology

2.2.1 Proposed Solution

Our proposed solution to this problem is to capture the I/O lineage of a program, which refers to the detailed execution trace of a program in terms of input/output system calls. We capture this lineage at byte granularity, providing a fine-grained analysis of the data elements used during the program's execution.

To facilitate this, we employ function interposition through the LD_PRELOAD keyword to dynamically insert our library into the user's application. This method necessitates minimal changes to the user's workflow, making it a user-friendly approach.

During the Execution Analysis phase, our solution leverages the captured I/O lineage to identify the data components actively used by the program. We analyze the data provenance to comprehend the relationships between different parts of the dataset and the program's interactions with them. A critical part of this analysis is identifying and tracking derived data created during execution and creating backups for any data that is modified or overwritten. This ensures the original data can be restored post-execution.

In the Data Carving phase, we debloat the original dataset D based on insights from the execution analysis. We remove redundant or unnecessary components, retaining only the necessary data. The resulting dataset D' is a minimized version of the original dataset but still allows for accurate program execution.

During the Re-execution phase, the application is rerun using the debloated dataset. This re-execution should be transparent to the user and produce consistent results. If any divergence occurs, the user is notified about the discrepancy.

Finally, in the Container Verification phase, we assess the effectiveness of our debloating techniques. We measure the reduction in container image size and evaluate the impact on resource utilization. We ensure that the debloated dataset produces consistent results and consider any trade-offs in execution time or memory usage.

Our solution combines the benefits of byte-granularity I/O lineage capture, a lightweight user-friendly approach, and effective debloating of datasets. By providing precise data debloating while maintaining a seamless and efficient workflow, we aim to enhance resource utilization and efficiency in containerized environments.

Chapter 3

Related Work

3.1 Containerization

Containerization is a technology that packages, distributes, and deploys applications within containers. This thesis focuses on data debloating within containers, examining how three major containerization platforms – Docker [7], Podman [10], and SciUnit [3] – operate and the unique features they offer.

1. Docker is a widely adopted containerization platform known for its ease of use. It enables developers to create, deploy, and manage containers quickly and easily. By packaging applications and their dependencies into container images, Docker ensures consistent execution across different environments. Docker's extensive library of pre-built images and its powerful command-line interface make it a popular choice among developers and system administrators.

2. Podman, an open-source containerization tool, offers a unique feature of rootless containers. These containers can be run without requiring root privileges, improving security and ease of use in multi-user and shared environments. Podman is also compatible with Docker's command-line

interface, providing a familiar experience for Docker users.

3. SciUnit is designed for scientific applications and reproducible research. It enables researchers to package and share their experiments in self-contained containers, capturing the entire environment, including code, data, and dependencies. This promotes consistent execution across systems and fosters collaborative research and result verification.

These containerization platforms cater to different needs. While Docker and Podman are suited for general-purpose application deployment, SciUnit addresses the unique requirements of scientific research and analysis. As we explore data debloating for containers, these platforms serve as the foundation for our research into optimizing container performance and resource efficiency.

3.2 Debloating

Containerization enables efficient sharing of applications. However, as applications and datasets become more data-intensive, efficiently sharing containers becomes a challenge. To address this, data debloating focuses on creating smaller datasets by pruning unused data. Additionally, software debloating reduces container sizes by optimizing application code and libraries. This thesis explores data debloating for containers, aiming to identify a subset of the original dataset sufficient for efficient program execution.

Efficiently sharing containers with large datasets is crucial for collaborative research and resource optimization. Data debloating reduces the burden of distribution and storage without sacrificing functionality or accuracy by creating compact subsets of datasets. This thesis examines data debloating for containers, proposing innovative approaches to creating efficient dataset subsets and contributing to streamlined containerization.

Software debloating [5] [11] [13] and data debloating are vital in modern containerization. While software debloating optimizes container sizes by refining application code and libraries, data debloating focuses on efficiently managing and sharing datasets within containers. This thesis explores data debloating for containers, aiming to create compact subsets of datasets to advance efficient container sharing and distribution in data-intensive applications.

Two notable examples of data debloating research are the SlimToolkit [12] for Docker and the IOSPReD framework [8].

3.2.1 SlimToolkit

SlimToolkit, designed for Docker container optimization, offers both software debloating and data debloating capabilities. It improves containerized application efficiency and resource utilization by removing unnecessary components, reducing the container's overall size and memory footprint.

SlimToolkit operates at a file level, analyzing individual files within the container to determine if they have been accessed during runtime. It then decides whether to include or exclude the file from the container. This file-level granularity aims to eliminate unnecessary files and components, streamlining the container image and improving resource efficiency.

However, SlimToolkit does not consider scenarios where large files are partially accessed or where only metadata has been read. This limitation may lead to overprovisioning, with containers including files that are only partially accessed or contain relevant metadata.

3.2.2 IOSPReD

IOSPReD is a data debloating framework that focuses solely on data debloating. It uses the LLVM [6] toolchain to produce, analyze, and transform the bitcode of targeted applications, allowing fine-grained analysis and transformation of data elements.

While IOSPReD offers granular byte-level debloating, it introduces a more involved workflow due to its reliance on the LLVM toolchain, which may require a higher level of expertise. Additionally, IOSPReD is designed for read-only applications, restricting its applicability to a specific subset of use cases.

3.2.3 Comparison of SlimToolkit and IOSPReD

In the context of the first program, where data is partially read from two files, data debloating at a fine granularity offers significant advantages over coarser approaches like SlimToolkit. By focusing on more granular data sharing, we can optimize the amount of data included in the container, reducing its size and the resources required for sharing it.

In the second program, where data is read, combined, written back into the dataset, and read again, it is crucial to track derived data. IOSPReD falls short in this regard. Our proposed system addresses these challenges by tracking derived data and ensuring relevant data is included in the shared container. It processes data in real-time, identifying destructively changed data and creating backups, enabling recovery of original data for post-execution analysis or replication. Moreover, our system does not rely on the LLVM toolchain, making it more accessible for scientists. By addressing these challenges, our system enables efficient data debloating in scenarios involving reading, writing, and modifying data, facilitating sharing and replication of data-intensive scientific applications.

Chapter 4

Architecture

4.1 Overview

Our proposed tool tackles the challenge of data debloating in containerized environments by breaking the problem into three distinct yet interconnected phases: the Audit Phase, the Datastore Creation Phase, and the Re-Execution Phase. Each phase serves a vital function in the data debloating process, contributing to an overall system that is both systematic and efficient. In this section, we provide an overview of each phase and highlight the key concepts that underpin the entire architecture.

4.1.1 Key Concepts

Before diving into the individual phases, let us first establish some fundamental concepts that will recur throughout the explanation of our system.

Definition 4.1.1. Event. An event is a six-tuple $< id, t, c, l, sz, h >$ consisting of the following elements:

- id identifies the subject that generated the event and the object or subject it affects,

- t represents the logical timestamp of affecting o,
- c signifies the type of system call,
- l marks the start offset location in object o which the event affects,
- sz denotes the size of the affected object o starting from l, and
- h represents the SHA-256 hash of the buffer contents during read and write events.

Definition 4.1.2. Audit Trace. An audit trace, denoted as A, is an ordered set of events that fully captures all events during an execution of a given program P. Formally, $A_P = \{e_0, e_1, e_2, ..., e_n\}$.

Definition 4.1.3. Merge. Two events, e_1 and e_2, with the same id, type, and dependency, are considered merged if:

- $e_2(t) < e_1(t)$, and
- $e_2(l_1) \leq e_1(l_1) < e_2(l_1) + e_2(sz)$.

Definition 4.1.4. Program State. The program state, within the context of execution and I/O, refers to the instantaneous condition and data of a program at a given moment during its execution. This encompasses its memory, variables, I/O streams, file descriptors, and control flow.

Definition 4.1.5. Complete Audit Trace. An Audit Trace, represented by A, is deemed complete for a program P if it captures all Read/Write I/O calls for the specified program.

4.1.2 Phase 1: Audit Phase

The initial phase, known as the Audit Phase, involves monitoring the execution of the application and capturing the I/O lineage at a byte-level granularity, forming the Audit Trace. By employing function interposition (via LD_PRELOAD), our tool dynamically inserts a library into the application. This allows us to meticulously track and record the details of input/output system calls, giving us a comprehensive audit trail that details the application's data interactions.

4.1.3 Phase 2: Datastore Creation Phase

Following the Audit Phase, the tool uses the recorded I/O lineage data to create a dedicated datastore. In this phase, the tool analyzes the I/O lineage, identifies the relevant data, and constructs a new dataset (D') that contains only the necessary data elements while excluding those that are redundant or unused. The datastore is a compact representation of the data elements required for the accurate execution of the application. Alongside the new datastore, this phase also generates a mapping structure to map the original dataset (D) to the newly created datastore (D').

4.1.4 Phase 3: Re-Execution Phase

The final phase, referred to as the Re-Execution Phase, leverages the created datastore (D') to re-execute the application. In this phase, the containerized application is launched using the newly constructed datastore, which contains the essential data components. Using function interposition (via LD_PRELOAD), the tool dynamically inserts a library into the application to intercept all I/O calls and redirect them to the optimized datastore (D'). This ensures the accurate and efficient execution of the application, resulting in improved resource utilization and performance.

By breaking down the process into these three well-defined phases, our tool adopts a systematic approach to data debloating. The Audit Phase captures fine-grained data usage, the Datastore Creation Phase constructs an optimized datastore, and the Re-Execution Phase guarantees the accurate execution of the containerized application. This structured workflow leads to precise data debloating, reduced resource consumption, and enhanced efficiency in containerized environments.

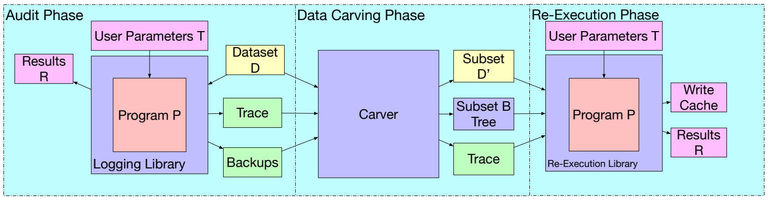


Figure 4.1: Full System Architecture

4.2 Alternate Structure

In the alternative approach being considered, the system undergoes a simplification by consolidating the first two phases—Audit and Datastore Creation—into a single phase. Although this streamlined process may have some advantages, it is essential to weigh the pros and cons before deciding which approach is more appropriate.

Advantages of combining the Audit Phase and Datastore Creation Phase into a single phase:

1. *Efficiency*: Combining the Audit and Datastore Creation Phases eliminates the need for separate iterations for capturing I/O lineage and creating the subsetted dataset, resulting in a more efficient and less time-consuming process.

2. *Simplicity*: The consolidated approach reduces the number of steps involved in the workflow, making it less complex and easier to understand

and implement.

Disadvantages of combining the Audit Phase and Datastore Creation Phase:

1. *Storage Requirements*: With this combined approach, users must store each I/O lineage alongside the preserved datasets, regardless of whether they plan to create the subsetted dataset immediately. This requirement can result in increased storage needs.

2. *Increased Execution Overhead*: The process of combining the phases might introduce computational overhead for applications with extensive I/O lineages, as it entails processing and analyzing the entire I/O lineage, including unused data, which can extend the overall execution time.

While merging the Audit Phase and Datastore Creation Phase into a single phase could offer benefits in terms of efficiency and simplicity, it also brings up considerations related to storage requirements and potential execution overhead. Therefore, the choice between the two-phase or three-phase approach should take into account the specific needs of the application and weigh the trade-offs associated with storage usage and computational overhead.

Our decision to utilize a three-phase approach for our data debloating system was based on a careful assessment of multiple factors and requirements. Here are the reasons behind our choice:

1. *Flexibility and Selective Data Subset Creation*: The three-phase approach enables users to capture the I/O lineage in the Audit Phase without immediate pressure to create the subsetted dataset. This flexibility allows for the capture of detailed data usage information without imposing the obligation of creating an optimized dataset. Users can generate the subsetted dataset when sharing their work becomes necessary or when optimizing resources becomes a priority.

2. *Efficient Storage Management*: The three-phase approach separates the creation of the I/O lineage from the subsetted dataset in the Datastore Creation Phase, providing a lazy creation scheme. Users can maintain multiple I/O lineages without needing to store each subsetted dataset separately. This approach efficiently manages storage resources by eliminating the need for potentially redundant or unused datasets, leading to optimized storage utilization.

3. *Workflow Modularity and Reusability*: The three-phase approach offers modular components that can be reused independently. The Audit Phase captures the I/O lineage, providing insights into data usage patterns that can be stored and used for various purposes beyond data debloating, such as performance analysis or debugging. The Datastore Creation Phase leverages the captured I/O lineage to construct optimized datasets. This modularity allows for flexibility in utilizing the captured lineage and encourages reusability in different contexts.

In conclusion, we selected the three-phase approach for its flexibility, efficient storage management, modular workflow, and preservation of execution state. It offers users the ability to capture the I/O lineage without immediately creating the subsetted dataset, optimizes storage usage, fosters modularity and reusability, and ensures the data debloating process's accuracy.

4.3 Proposed Workflow

The proposed three-phase approach for data debloating is a comprehensive process that builds upon the advantages and considerations previously discussed. The approach comprises the following steps:

1. **Step 0: Whitelisting Data Files** Before initiating the debloating process, the paths of the data files utilized by the application need to

be whitelisted. This crucial preparatory step ensures the inclusion of the data files in the subsequent debloating phases.

2. **Step 1: Capturing I/O Lineage (Audit Phase)** In this phase, the audit library shared object is loaded into the application's execution environment using the LD_PRELOAD directive. For instance:

   ```
   LD_PRELOAD=../Audit/auditLib.so␣python3␣global_prec.py
   ```

   By setting the LD_PRELOAD environment variable to the audit library, the library is loaded before the program's execution. This step captures the program's I/O lineage and records data usage patterns during execution, fulfilling the objectives of the Audit Phase.

3. **Step 2: Creating Optimized Dataset (Datastore Creation Phase)** The captured I/O lineage from Step 1 is utilized to generate the optimized subsetted dataset. This is achieved by running the `createDatastore.py` Python script, as shown below:

   ```
   python3␣createDatastore.py
   ```

   This script analyzes the I/O lineage and identifies the essential data components required by the application. Based on this analysis, the optimized subsetted dataset is constructed, in line with the objectives of the Datastore Creation Phase.

4. **Step 3: Re-executing with Optimized Dataset (Re-execution Phase)** In this final phase, the application is re-executed using the optimized dataset created in Step 2. The `repeatLib.so` shared object is loaded into the application's execution environment using the LD_PRELOAD directive, as illustrated below:

   ```
   LD_PRELOAD=../ReExecute/repeatLib.so␣python3␣global_prec.py
   ```

The application is then launched with the optimized dataset, ensuring accurate execution with improved resource utilization, as per the goals of the Re-execution Phase.

By following these systematic steps, the three-phase approach facilitates efficient data debloating. It captures the I/O lineage, constructs the optimized subsetted dataset, and re-executes the application using the optimized dataset. This process addresses the considerations discussed earlier and optimizes storage utilization, promotes modularity and reusability, and ensures accurate data debloating.

4.4 Step 0

In our data debloating approach, we exploit the `LD_PRELOAD` method to intercept system calls made during the entire execution cycle. This includes both the user application and other components of the system. While `LD_PRELOAD` allows us to effectively intercept syscalls, it can inadvertently intercept syscalls made by unrelated system components. To refine and focus our analysis on the relevant I/O calls for data debloating, we utilize a whitelist and blacklist system.

The whitelist comprises paths of data files or directories used by the application. These paths represent the files and directories we wish to monitor and include in the debloating process. Logging and analyzing I/O calls specifically related to paths in the whitelist allows us to discern the critical data interactions.

Concurrently, we use a blacklist to exclude particular files or directories from being logged and considered for data debloating. This provides the flexibility to unmark all files in a directory, except for specific ones. By prioritizing the blacklist over the whitelist, we can efficiently filter out non-pertinent I/O calls and ensure only the necessary data interactions are considered.

Combining the whitelist and blacklist allows us to selectively intercept and log relevant I/O calls while excluding unrelated system calls. This targeted approach optimizes the data debloating process by focusing on the files and directories of interest. It ensures that the logged I/O calls are in line with the data usage patterns we aim to analyze and optimize.

To summarize, while `LD_PRELOAD` intercepts all system calls made during the execution cycle, our whitelist and blacklist system filters and differentiates the relevant I/O calls. This mechanism ensures that we capture and process only the I/O interactions specific to the data files and directories of interest, thereby enhancing the precision and efficacy of our data debloating approach.

Table 4.1: Sample Execution

Event Number	**Type**	**Offset**
1	R	0–110
2	R	70–100
3	R	130–150
4	W	80–100
5	R	90–120
6	W	70–130

4.5 Step 1: Audit Phase

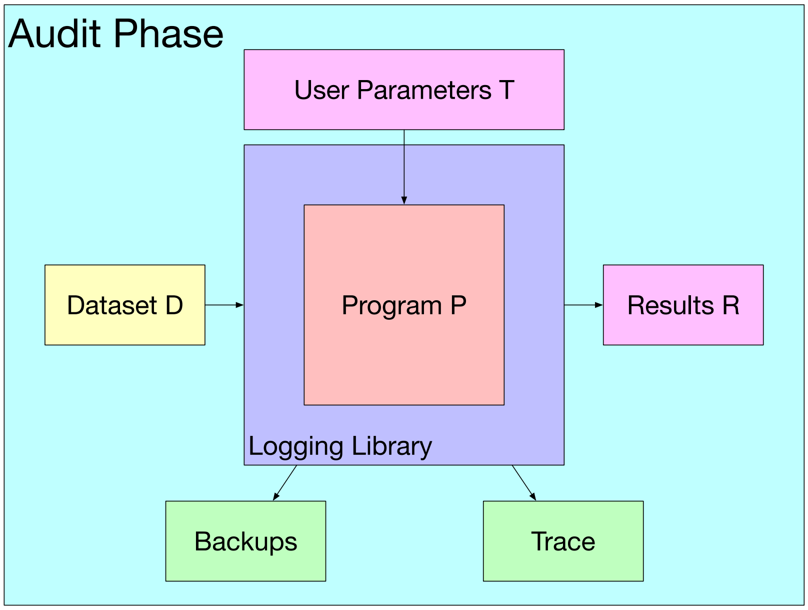


Figure 4.2: Audit Phase Architecture

The goal of this step is to capture the I/O lineage of the application intended for containerization and sharing. The I/O lineage provides a detailed record of the data interactions made by the application during its execution. To achieve this, we utilize the LD_PRELOAD mechanism to intercept all I/O calls made by the application. L_PRELOAD allows us to dynamically load a library that intercepts and modifies system calls, enabling us to capture the I/O lineage at a fine-grained level. Each I/O call made to files is logged individually, creating a lineage that tracks the interactions per file rather than per application.

Definition 4.5.1. Destructive Change When there is a read-write conflict where:

- $e_1(c)$ = Read and

- $e_2(c)$ = Write and
- $e_2(t) > e_1(t)$ (e_2 happens after e_1)and
- $e_2(l) \leq e_1(l) < e_2(l) + e_2(sz)$ or $e_2(l) < e_1(l) + e_1(sz) \leq e_2(l) + e_2(sz)$

the event e_2 is said to be a destructive change.

During the execution of the application, there may be destructive changes or modifications to the dataset where the dataset in its original form might not be available post-execution. These changes can pose challenges when creating an appropriate subset for re-execution. It is crucial to identify and address such destructive changes to ensure the integrity and accuracy of the subsetted dataset.

Another point of concern is the data structure used to keep track of offsets while minimizing lookup overhead. Efficient management of offsets is important for accurately reconstructing the subsetted dataset during the re-execution phase. Choosing an appropriate data structure that balances lookup efficiency and memory usage is essential for optimizing the debloating process.

By addressing these concerns and implementing an effective mechanism to capture the I/O lineage, we can achieve the goal of logging the data interactions of the application. This lineage serves as a foundation for subsequent phases, enabling the creation of an optimized subsetted dataset and facilitating the accurate re-execution of the application in a containerized environment.

4.5.1 Online and Offline Approaches

Backup Creation: Online and Offline Approaches

In our data debloating process, we consider two approaches for MDR creation based on the availability of the dataset D post execution in the same state as it was pre-execution.

1. Online Backup Creation: When the dataset undergoes destructive changes during execution or the pre-execution state dataset is not available post-execution, we employ the online backup creation approach. This method enables us to dynamically create backups during execution to preserve the necessary data for accurate re-execution. It involves maintaining separate data structures, such as the backup list, read tree, and write tree, to track and capture the modified portions of the dataset. With online backup creation, we can adapt to scenarios where the dataset is modified, ensuring the integrity of the subset data.

2. Offline Creation of Minimum Data Requirement (MDR): In situations where the pre-execution state dataset is available post-execution without destructive changes, we can leverage the offline approach. In this method, we focus solely on the creation of the Minimum Data Requirement (MDR) without the need for backups. By utilizing the available pre-execution dataset, we generate the MDR offline, which represents the essential subset of the original dataset required for accurate re-execution. This approach simplifies the process by eliminating the need for separate backup structures, reducing computational overhead. During the offline creation, we only require the execution trace to identify the accessed offsets and generate the MDR.

The offline approach eliminates the need for the backup list, read tree, and write tree data structures used in the online approach, relying solely on the execution trace to identify accessed offsets. Although the offline method offers simplicity and efficiency when the pre-execution state dataset is available, we chose the online backup creation method for its robustness and ability to handle scenarios involving destructive changes during execution or when the pre-execution dataset is inaccessible.

4.5.2 Dealing with Destructive Changes

In the given scenario, where a portion of a file is read and subsequently undergoes a destructive change, the challenge arises in recovering the original portion of the file for accurate re-execution. To address this problem, we propose a solution that involves keeping track of the offsets of the file that are read and creating backups for those specific portions in case any destructive changes occur.

The solution can be implemented as follows:

1. Tracking Read Offsets: During the execution of the application, we keep a record of the offsets within the file that are read. This can be achieved by monitoring the read syscalls or tracking the read operations performed by the application. By recording the offsets, we have a reference to the specific portions of the file that are accessed and utilized.

2. Creating Backups: Upon identifying the offsets that have been read, we create backups for those portions of the file which are about to be overwritten by a destructive change as defined in definition 4.5.1. These backups serve as snapshots of the original content before any destructive changes are made. By preserving the original data, we ensure that the necessary portions required for accurate re-execution are available even if destructive changes occur during the execution.

3. Re-execution with Original Portions: After the execution is completed, we can utilize the backups created in the previous step to restore the original portions of the file. By replacing the modified sections with the corresponding backups, we can recreate the exact state of the file as it was during the initial read operations. This allows for accurate re-execution of the application, as the required data is restored to its original state.

By implementing this solution, we overcome the problem of destructive changes to file portions. By tracking read offsets and creating backups, we ensure that the original data is preserved and accessible for re-execution. This approach guarantees the availability of the necessary portions of the file needed for accurate re-execution, even in the presence of destructive changes made during the application's execution.

4.5.3 Logging Data Structure

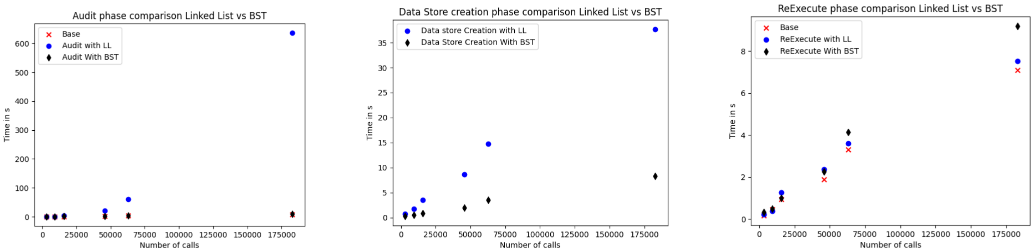
In order to effectively track the offsets that have been read and written, we consider two suitable data structures: Linked Lists and Interval Binary Search Trees (BST) based on AVL trees. These data structures will assist in maintaining a sequential record of the offsets for efficient tracking.

Linked Lists provide a straightforward approach to store the offsets in a sequential manner. Each node in the linked list contains the offset information along with a reference to the next node. This structure allows for easy insertion and deletion of nodes. However, searching for a specific offset may require traversing the entire linked list, resulting in linear time complexity.

On the other hand, Interval BSTs based on AVL trees are more advanced data structures that efficiently handle intervals. By treating the read and write offsets as intervals, an Interval BST enables efficient searching, insertion, and deletion of intervals. It offers logarithmic time complexity for operations, making it an optimal choice for interval-related queries.

Additionally, in our solution, we maintain linked lists for backups and to track the I/O lineage. These linked lists facilitate efficient management and retrieval of backup data and the recorded I/O lineage.

To compare the effectiveness of the two data structures, we execute the audit phase of the code using our I/O-sensitive benchmark problems. By running the audit phase with these benchmark problems, we can analyze and evaluate the performance and efficiency of both data structures. This comparison will help determine which data structure provides better tracking capabilities for the offsets and ultimately assist in making an informed decision about the most suitable structure for our specific use case.



(a) $AuditPhase$ (b) $DatastoreCreation$ (c) $Re - Execution$

Figure 4.3: Linked List vs BST comparison

The results obtained align with our expectations and validate the effectiveness of using Binary Search Trees (BST) over Linked Lists in our implementation. Here are the additional points to address the specific observations:

1. The runtime scaling of the Audit phase with the number of I/O calls is depicted in Figure 4.3a, where the BST approach exhibits a logarithmic scaling while the Linked List approach shows exponential scaling. This is due to the efficient lookup capability of BSTs and the linear search nature of Linked Lists

2. In the Data store creation phase, the choice of data structure significantly impacts the scaling behavior. Figure 4.3b illustrates the runtime scaling of the Data store creation phase with the number of I/O calls. The BST approach exhibits linear scaling, where the time taken for Data store creation increases proportionally with the number of I/O calls. On the other hand, the Linked List approach shows exponential scaling, resulting in a substantial increase in runtime as the number of I/O calls grows. This confirms that BSTs provide a more efficient and predictable performance in the Data store creation phase, making them the preferred choice over Linked Lists.

3. During the Replay phase, the performance advantage of Linked Lists is observed, as they only need to pop elements from the front, resulting in constant time complexity. However, when considering the overall data debloating process, the significant slowdown in the Audit and Data store creation phases outweighs the marginal benefits in the Replay phase. This is illustrated in Figure 4.3c, showcasing the comparative runtime performance of the Linked List and BST approaches.

The slower performance of Linked Lists in the Audit and Data store creation phases is the key factor behind our decision to utilize BSTs in our implementation. The figures demonstrate that BSTs offer more efficient scaling with logarithmic or linear behavior, while Linked Lists exhibit exponential or constant time complexity, depending on the phase. This validates the superiority of BSTs in terms of runtime efficiency and scalability, supporting their selection over Linked Lists for data debloating.

4.5.4 Data Structures Maintained

To track offsets, we utilize two interval trees: R for read offsets and W for write offsets. Interval tree R maintains the ranges of offsets that have been read, while interval tree W tracks the ranges of offsets that have been written. These structures enable efficient insertion, deletion, and lookup operations for the respective offsets, providing a comprehensive record of read and write operations on the file.

In addition to the interval trees for tracking read and write offsets (R and W), we also maintain two linked lists. One linked list is used for backups to store original data portions, while the other linked list tracks the I/O lineage.

Data Structure Operations

For the ith read operation, denoted as r_i, we perform the following operations:

$R = R \cup (\{r_i\} - W)$

In this equation, we insert into R only the offsets from the read call that have not been written over. By subtracting the write offsets (W) from the read offsets ($\{r_i\}$), we ensure that only the offsets that have not been modified or overwritten are included in R.

This approach allows us to dynamically obtain the offsets that have not been written over during the audit phase. By considering only the offsets that have not been modified, we can reduce redundancy in the offsets we capture limiting ourselves to the original state of the file.

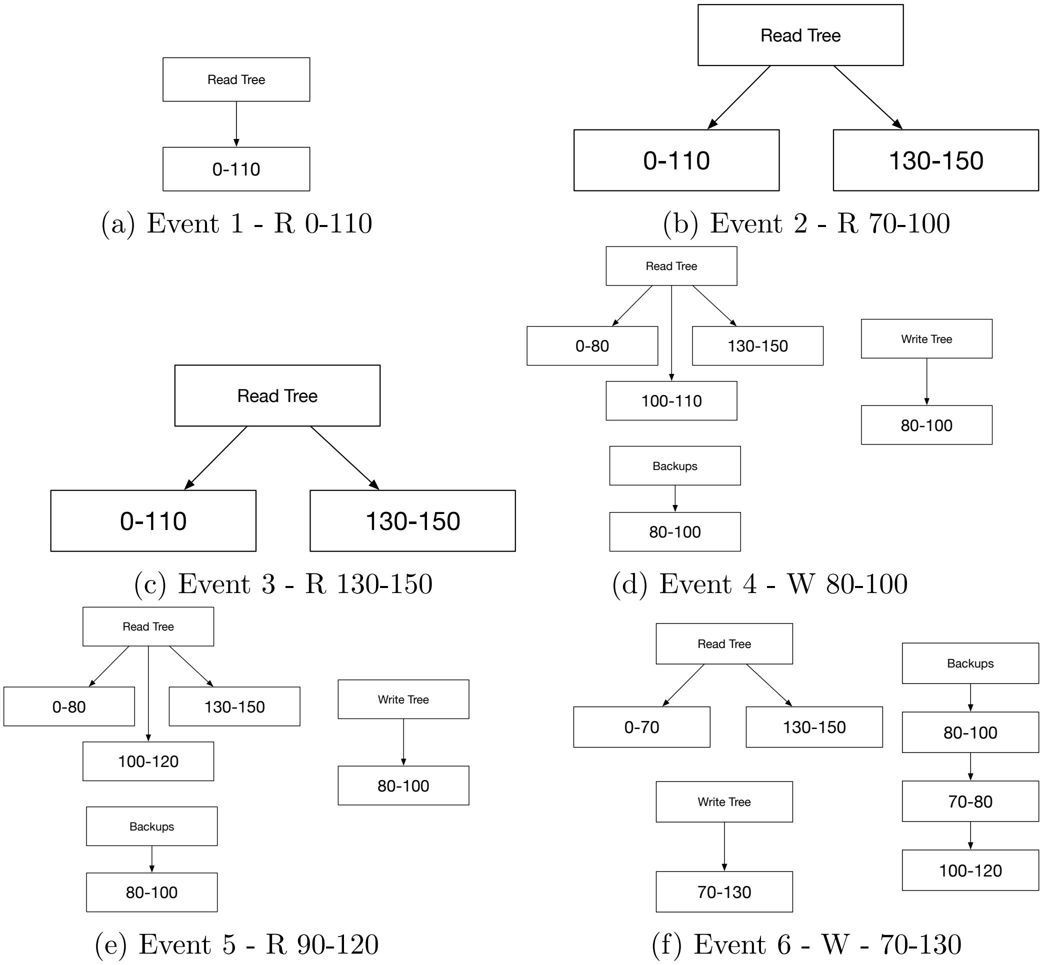
For the ith write operation, denoted as w_i, the following operations are performed:

1. CreateBackup($\{w_i\} \cap R$): Create a backup for any write operation that is overwriting a previous read which has not yet been overwritten. We identify the offsets that are both in $\{w_i\}$ and in R, indicating the portion of the file that is being overwritten. A backup is created for this section to preserve the original data.

2. $W = W \cup \{w_i\}$: Add the write call w_i to the write interval tree W. This ensures that the write offsets are properly tracked and maintained.

3. $R = R - \{w_i\}$: Remove the section written to by w_i from the read interval tree R. This update reflects that the corresponding portion of the file has been modified and is no longer considered part of the original file that has been read.

By performing these operations, we create backups for writes over previous reads, update the write interval tree, and update the read interval tree by removing the sections that have been written to. These steps help maintain the integrity of the data and accurately reflect the modifications made during the write operations. To better illustrate the operations we can see the snapshot of all the data structures in Figure 4.4 at each step of the execution for the sample execution in Table 4.1



(a) Event 1 - R 0-110

(b) Event 2 - R 70-100

(c) Event 3 - R 130-150

(d) Event 4 - W 80-100

(e) Event 5 - R 90-120

(f) Event 6 - W - 70-130

Figure 4.4: Audit Phase Data Structures Snapshot

4.5.5 Optimizations

Definition 4.5.2. Minimum Data Requirement The minimum data requirement (MDR) of a program P, with user parameters T, on dataset D, is the minimum subset of data from the dataset D which is required to execute the program P with user parameters T such that both produce consistent results R.

Theorem 4.5.1. The audit created by our system produces a complete trace w.r.t to the whitelist specified and the tracking mechanism used.

Proof. As we are capturing all I/O calls using LD_PRELOAD, and we perform only merges on the I/O offsets, we can prove that the audit is complete w.r.t to the whitelist specified and tracking mechanism used □

Theorem 4.5.2. The MDR for a program **P** with user parameters **T** only includes data from the original dataset **D** w.r.t to the specified whitelist for the system

Proof. Let the original dataset be D. After the i^{th} write event w_i let the modified dataset be D_i. Let the union of all the reads between the write event w_i and w_{i+1} be R_i. Using this we can represent our execution trace in the following manner.

$$D \xrightarrow[R_0]{w_1} D_1 \xrightarrow[R_1]{w_2} D_2 ... \xrightarrow[R_{n-1}]{w_n} D_n$$

As all writes can be dynamically generated during runtime we can express the minimum data requirement as:

$$\begin{aligned} MDR = & R_0 \cup (\\ & \quad R_1 \cup (\\ & \quad\quad R_2 \cup (\\ & \quad\quad\quad ... \quad R_{n-1} \\ & \quad\quad\quad) \\ & \quad\quad) - w_2 \\ & \quad) - w_1 \end{aligned}$$

For any two consecutive version of the dataset D_i and D_{i+1} created due to write events w_i and w_{i+1}, if we subtract the region of impact of w_{i+1} which is $\{w_{i+1}(l), w_{i+1}(l) + w_{i+1}(sz)\}$ from the dataset version D_{i+1} then the leftover

parts of the dataset are a subset of D_i
As we subtract all w_i we can recursively apply the formula above to show that MDR contains only the data elements from D, i.e $MDR \cup D$ □

As the Minimum Data Requirement (MDR) represents a subset of the original dataset, we leverage this optimization strategy in our data debloating process. By tracking offsets only for the MDR, we reduce overhead and improve efficiency. This approach ensures that the subsetted dataset contains the necessary data for accurate re-execution while minimizing unnecessary tracking and storage.

4.6 Step 2: Dataset Creation

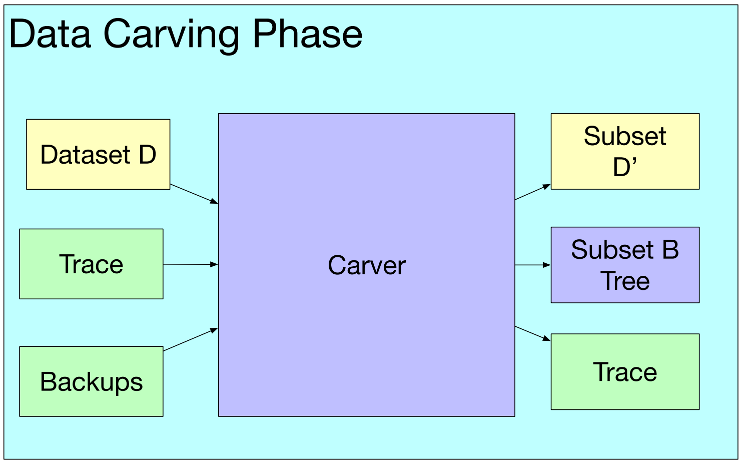


Figure 4.5: Dataset Creation Architecture

The goal of this phase is to utilize the backups and offset metadata obtained in Phase 1 to create a subset datastore and mappings from the original dataset to the subset datastore. These resources will be utilized during the re-execution phase of the program.

To achieve this, we combine the accessed offsets from the original dataset files during the audit phase with the corresponding backup offsets. This combination enables us to create a condensed subset datastore that contains only the essential data for re-execution. The mapping and condensation process of the datastore can be visualized in Figure 4.6, where the blue nodes represent items from the Read Tree in Phase 1, and the green nodes represent the backups created during Phase 1.

During this process, three distinct items are created to facilitate the re-execution phase:

1. Original I/O Trace: This component provides a blueprint for comparison during the re-execution phase. It allows us to check for consistency between the original audit and the re-execution. It contains a detailed record of I/O operations captured during the audit phase, enabling accurate replication of the original program's data interactions.

2. Subset Data: This item represents a condensed version of the data required for re-execution. It includes only the essential data components extracted from the original dataset based on the accessed offsets and corresponding backups. The subset data provides a more lightweight and efficient representation of the original dataset, tailored specifically for re-execution.

3. Subset Metadata: The subset metadata functions as a mapping mechanism that connects the subset data to specific locations within the original file, including file offsets. This metadata includes crucial information, such as file names or identifiers, which facilitates the accurate alignment of the subset data with the corresponding sections of the original dataset. By incorporating file offsets and other necessary details, the subset metadata ensures precise re-execution by correctly referencing the relevant portions of the original dataset during the data debloating process.

By creating these three components—Original I/O Trace, Subset Data, and Subset Metadata—we establish the foundation for the re-execution phase. The Original I/O Trace gives us a blueprint to check against to ensure consistency with the original audit, while the Subset Data and Subset Metadata provides a condensed dataset and the necessary mappings, respectively, for efficient and accurate re-execution of the program.

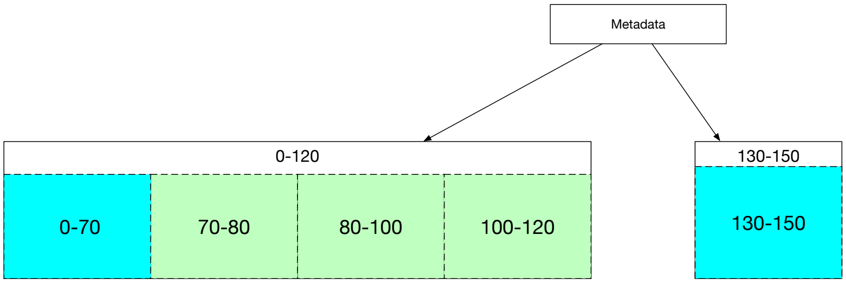


Figure 4.6: Subset Metadata in Phase 2

4.7 Step 3: Re-execution

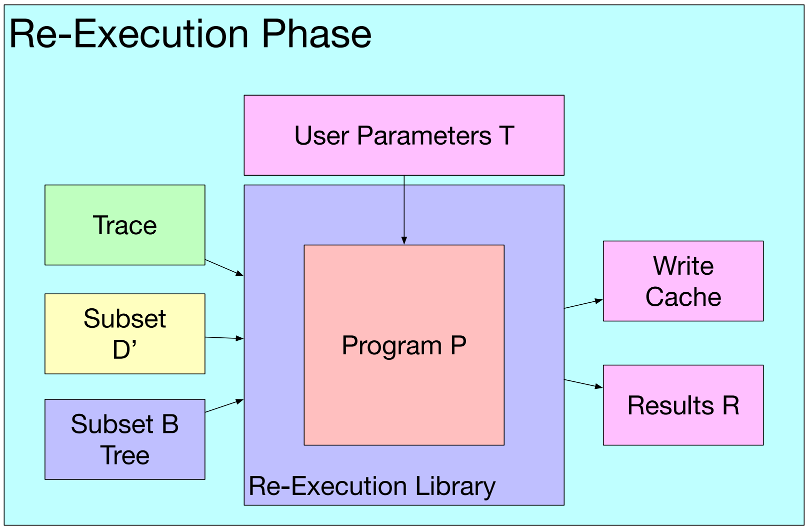


Figure 4.7: Re-Execution Architecture

The goal is to replay the original application in its original form, with the same program definition and parameters.

To achieve this, we intercept all I/O calls using the LD_PRELOAD mechanism. We utilize the mappings and subsets created in Step 2 to redirect all I/O system calls accordingly. By redirecting the I/O calls, we ensure that the application operates with the subset data and mappings, replicating the original execution.

One concern is the potential for destructively changing the subset data, which could corrupt the container for future executions.

4.7.1 Dealing with Destructive changes to the subset

To address this concern, we implement a caching mechanism for all writes in a separate file. This approach allows us to preserve the original subset data while redirecting the interval tree nodes to point to the appropriate location where the data exists, either in the subset file or the write cache. By effectively managing the write operations, we maintain data integrity and avoid corruption during re-execution.

4.7.2 Data Structures Maintained

During the Re-Execution phase, we simplify the data structure by maintaining only one metadata interval tree, denoted as M. Each node in this tree represents a specific region of the original file, including its associated metadata and its location within the subset file or the write cache. This streamlined metadata structure facilitates the efficient retrieval and utilization of the necessary data during the re-execution process.

By implementing these mechanisms, we ensure that the original application can be replayed without any data corruption or loss. The caching of writes and the utilization of the metadata interval tree M contribute to preserving the integrity of the subset data, allowing for unlimited re-execution cycles while maintaining consistency with the original trace.

Data Strcuture Operations

On the i^{th} read operation, denoted as r_i, the following operations are performed:

- Required Metadata: $RequiredMetadata = \{r_i\} \cap M$ This operation gives us the set of metadata nodes that correspond to the required read r_i. By intersecting the set $\{r_i\}$ with the metadata tree M, we obtain the relevant metadata nodes associated with the read operation.

Read Data Retrieval: Based on the required metadata set, we read the corresponding data from the specified locations indicated by the metadata. We concatenate the retrieved data and return it to the application. This ensures that the application receives the necessary data for the read operation.

On the i^{th} write operation, denoted as w_i, the following operations are performed:

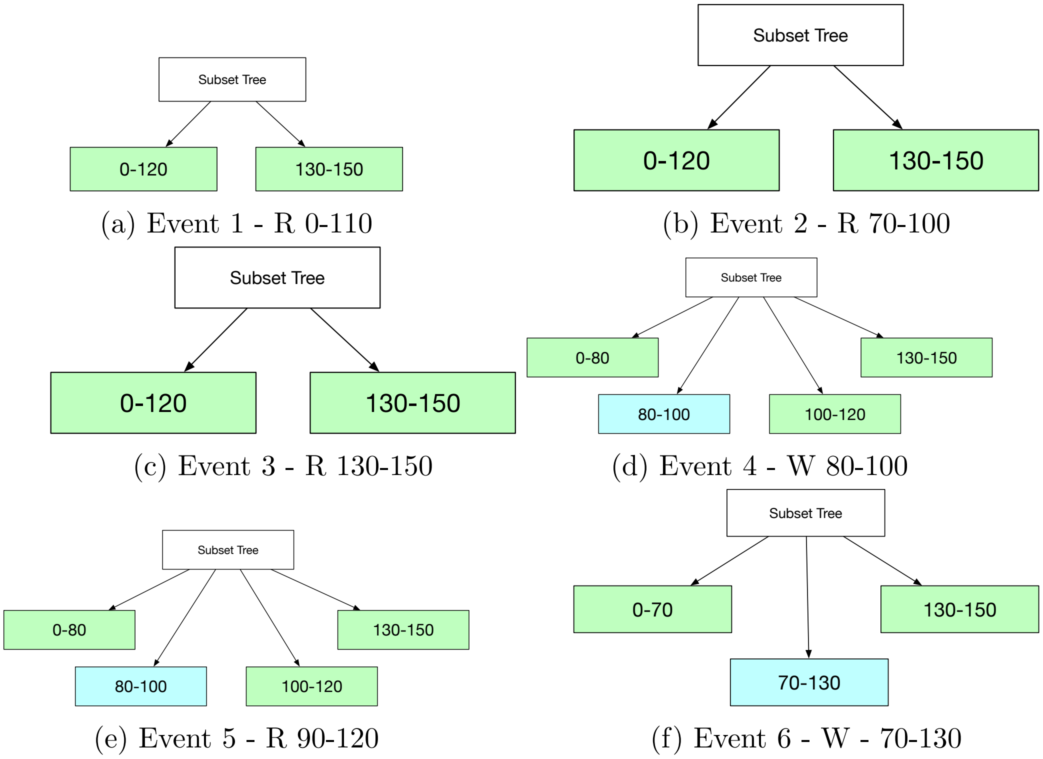
- $M = M - \{w_i\}$ We remove the portions of the metadata tree M that overlap with the write operation w_i. These overlapping portions are being overwritten, and thus their metadata is no longer needed.

- Write Cache: The write operation w_i is flushed to the write cache. This involves storing the write data in the designated write cache location.

- $M = M \cup \{w_i\}$ We add the metadata of the write operation w_i to the metadata tree M. This metadata entry points to the corresponding location in the write cache, ensuring that the metadata tree correctly reflects the updated state after the write operation.

By performing these operations, we ensure that the necessary metadata is retrieved for reads and that writes are properly handled. The removal of overlapping metadata, flushing of writes to the write cache, and updating the metadata tree guarantee the accurate representation of the file state during the re-execution phase.

To better illustrate the operations we can see the snapshot of all the data structures in Figure 4.8 at each step of the execution for the sample execution in Table 4.1



(a) Event 1 - R 0-110

(b) Event 2 - R 70-100

(c) Event 3 - R 130-150

(d) Event 4 - W 80-100

(e) Event 5 - R 90-120

(f) Event 6 - W - 70-130

Figure 4.8: ReExecution Phase Data Structures Snapshot

Chapter 5

Guaranteeing Consistent Re-Execution

Definition 5.0.1. Consistent Re-Execution When two executions of the same program P and the same user parameters T have the same data flow, the two executions are said to be consistent

Theorem 5.0.1. Subset Data mapping (SDM) mechanism is homeomorphic for all events e in the trace of the program P with user parameters T and dataset D (P(D,T)) w.r.t to P and T on the spaces D and D' i.e
$SDM(e, P, T, D) = SDM(e, P, T, D')$

Proof. The SDM takes all the original offsets of the dataset D accessed by the program and condenses it to form D'. It also creates a mapping function which maps each byte of D' to it's original location in D, thus having $SDM(e, P, T, D) = SDM(e, P, T, D')$ □

Theorem 5.0.2. Combination of MDR and SDM results in consistent re-execution

Proof. We will prove this by induction:
Base Case:

Let the program state at the beginning audit phase be P_o^A and at the beginning of the re-execution phase be P_o^R, we know that $P_o^A = P_o^R$ as the program P and user parameters T are the same.

Recursive Case::

Let the current event of both the executions be e_n and the state be the same i.e. $P_n^A = P_n^R$

For event e_{n+1} as the states are the same before it, and the SDM guarantees that we can recreate all events. We can assume that e_{n+1} will be same □

Chapter 6

Experiments

6.1 Experimental Setup

In our experimentation, we conducted tests on two different sets of applications: real-life applications and synthetic benchmarks. Our experiments were conducted on an AWS t3.xlarge instance running Ubuntu 22.04.

6.2 Experiment Details

Real-life Applications:

1. Krigging: We utilized a generalized interpolation method based on Gaussian regression, specifically on the Ozone profile data obtained from the Ozone Monitoring Instrument on the Aura Satellite [4]. This application involved processing real-life atmospheric data.

2. Visual Data Comparison: We employed a visual comparison program to compare data from the IMERG dataset with the data produced by the GEOS-5 model [1]. This application focused on visualizing and analyzing real-life climate and weather data.

3. Global Precipitation Visualization: We utilized an ipynb notebook obtained from GitHub [9] , which involved using precipitation data from an IMERG dataset file to plot precipitation for different regions of the world. This application allowed us to visualize global precipitation patterns.

Synthetic Benchmark: We developed a synthetic benchmark using a C program that utilized the HDF5 library [14] . The program was designed to access large datasets using five possible different access patterns. This benchmark aimed to simulate and evaluate various scenarios and patterns of dataset access, enabling us to assess the efficiency and performance of our data debloating techniques.

Figure **??**- **??** depicts the different access patterns used in the synthetic benchmark. While presented in two dimensions, these patterns can be scaled to three dimensions based on user-defined parameters such as stride and read size.

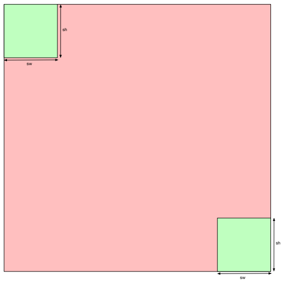


Figure 6.1: Pattern 1 Visualization

Parameter	Usage
sw	Horizontal width of selection
sh	Vertical height of selection
StrideZ	Stride/Step in Z direction

Figure 6.2: Pattern 1 input parameters

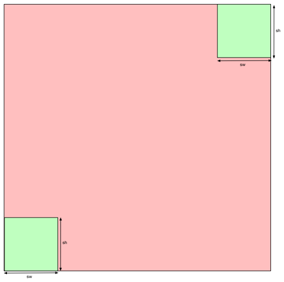


Figure 6.3: Pattern 2 visualization

Parameter	Usage
sw	Horizontal width of selection
sh	Vertical height of selection
StrideZ	Stride/Step in Z direction

Figure 6.4: Pattern 2 input parameters

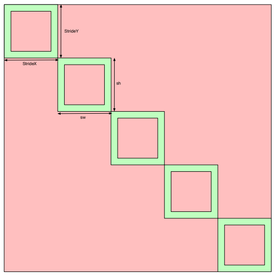


Figure 6.5: Pattern 3 visualization

Parameter	Usage
sw	Horizontal width of selection
sh	Vertical height of selection
StrideZ	Stride/Step in Z direction
StrideX	Stride/Step in X direction
StrideY	Stride/Step in Y direction

Figure 6.6: Pattern 3 input parameters

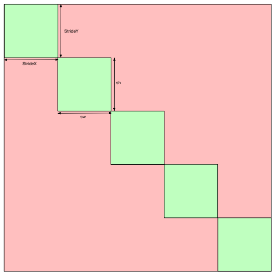


Figure 6.7: Pattern 4 visualization

Parameter	Usage
sw	Horizontal width of selection
sh	Vertical height of selection
StrideZ	Stride/Step in Z direction
StrideX	Stride/Step in X direction
StrideY	Stride/Step in Y direction

Figure 6.8: Pattern 4 input parameters

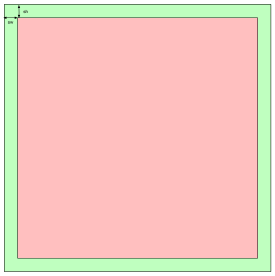


Figure 6.9: Pattern 5 visualization

Parameter	Usage
sw	Horizontal width of selection
sh	Vertical height of selection
StrideZ	Stride/Step in Z direction

Figure 6.10: Pattern 5 input parameters

By conducting experiments on both real-life applications and synthetic benchmarks, we were able to evaluate the effectiveness and applicability of our data debloating approach across different use cases and scenarios. With these experiments, we aimed to answer the following questions:

1. What is the time overhead for all three phases of our approach?
2. How do these overheads scale and on what factors?
3. What are the reductions in size achieved by creating subsetted datasets?

To address these questions, we conducted a total of 36 different configurations of our benchmarks. These configurations involved varying parameters such as the number of images in the dataset, the access pattern, the layout of the dataset on disk, and the amount of data subsetted from the original dataset. This variation allowed us to explore a wide range of scenarios and assess the impact of different factors on the performance of our approach.

The number of I/O calls in the resulting traces varied from 347 to 122,870, providing a diverse set of data points for analysis. To ensure reliable results, we performed each experiment 30 times, enabling us to capture variations and calculate statistical measures.

For the analysis and visualization of the results, we utilized boxplots, which offer a concise representation of the data distribution and facilitate comparisons across different configurations.

By conducting these experiments and analyzing the results, we gained insights into the time overheads of each phase, their scalability with respect to various factors, and the reductions in dataset size achieved through subset creation. These findings contribute to a comprehensive understanding of the performance and efficiency of our data debloating approach.

6.3 Results

6.3.1 Synthetic Benchmarks

Audit Overhead

As seen in figure 6.11 the applications with a lower number of I/O calls exhibit a higher interquartile range (IQR), indicating a greater variation in their execution time. This is primarily due to the significant impact of system context switching on these quick applications. As the number of calls increases, the IQR decreases, indicating a more consistent execution time, and the overhead converges to approximately 30%. On average, the overhead across all applications is close to 31%. These observations highlight the influence of system context switching on the performance of applications with different I/O call frequencies.

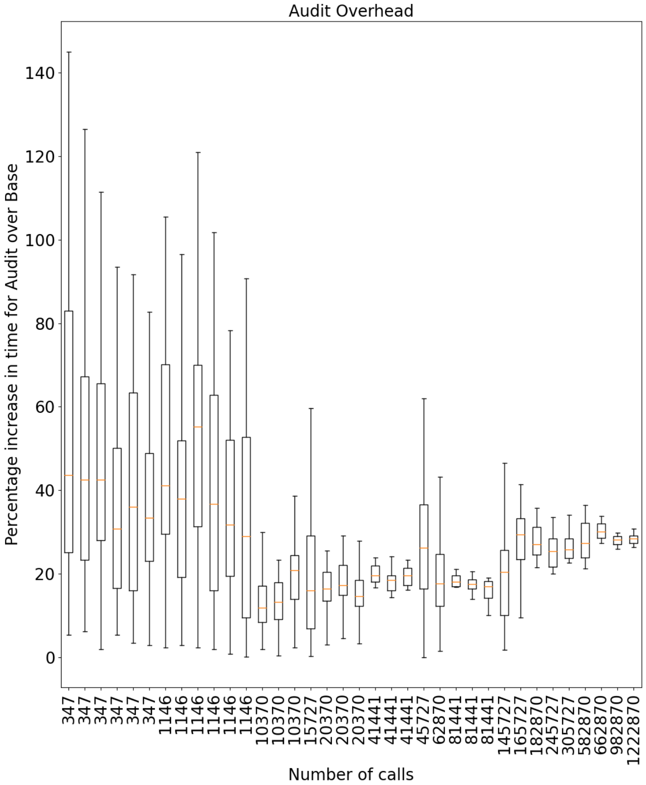


Figure 6.11: Audit Overhead for Synthetic Benchmarks

Datastore Creation Phase

Our earlier comparison results, as seen in Figure 4.3b revealed that the time taken by the datastore creation phase exhibits a linear relationship with the number of I/O calls in the trace. These findings, as seen in Figure 6.12, confirm those results and suggests that as the number of I/O calls increases, the time required for creating the subset datastore also increases proportionally.

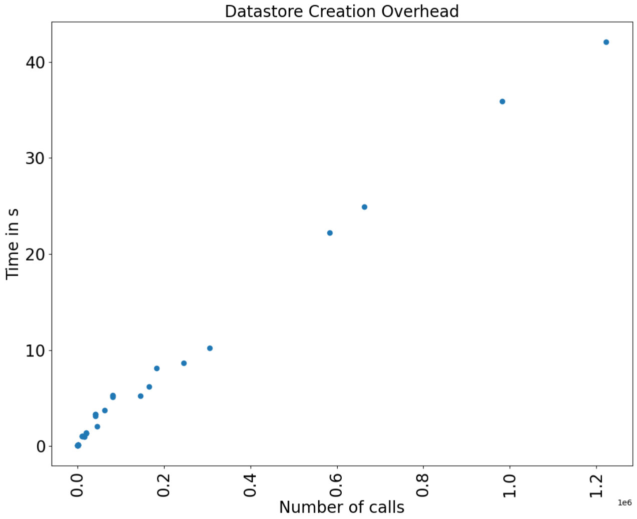


Figure 6.12: Data Store Creation for Synthetic Benchmarks

Re-Execution phase

Based on Figure 6.13, we can observe that applications with a lower number of I/O calls have a significantly higher interquartile range (IQR). This higher IQR is due to the substantial impact of system context switching on the execution time of these quick applications. As the number of calls increases, the IQR decreases, indicating a more consistent execution time. Additionally, the overhead converges to approximately 30

The mean overhead across all applications is approximately 27

These results support our observations and further emphasize the relationship between the number of I/O calls, the IQR, and the convergence of overheads.

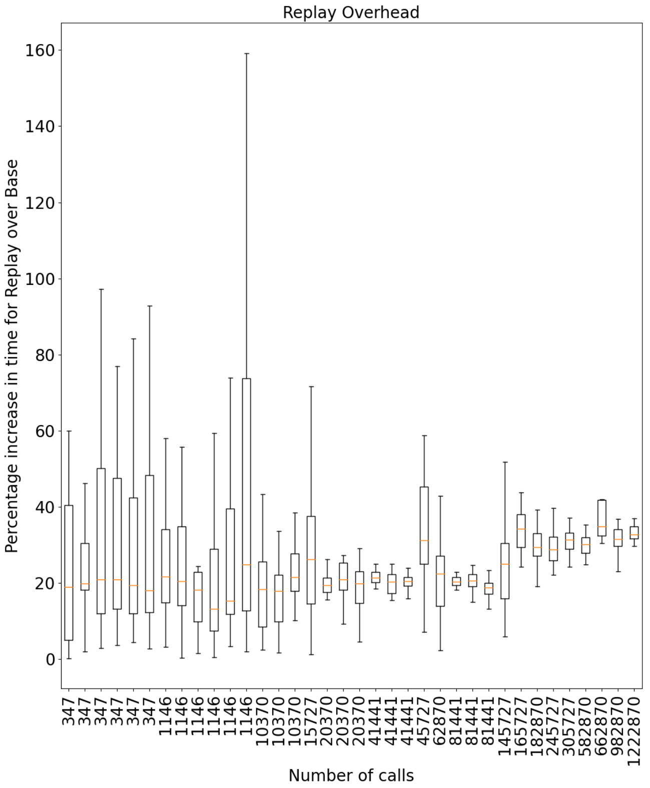


Figure 6.13: Replay Overhead for Synthetic Benchmarks

Data Reduction

In Figure 6.14, we can observe that there is no clear correlation between the number of I/O calls and the reduction in dataset size. This lack of relation is expected since the system can make multiple calls of small size or a single call reading the entire dataset.

It is worth noting that some datasets have actually increased in size. This outcome is also anticipated as systems that read the full dataset will include both the original dataset and the newly created mapping data in the container, resulting in a larger overall size compared to the original dataset.

These findings underscore the variability in dataset sizes and highlight the complexities involved in determining the size reduction solely based on the number of I/O calls.

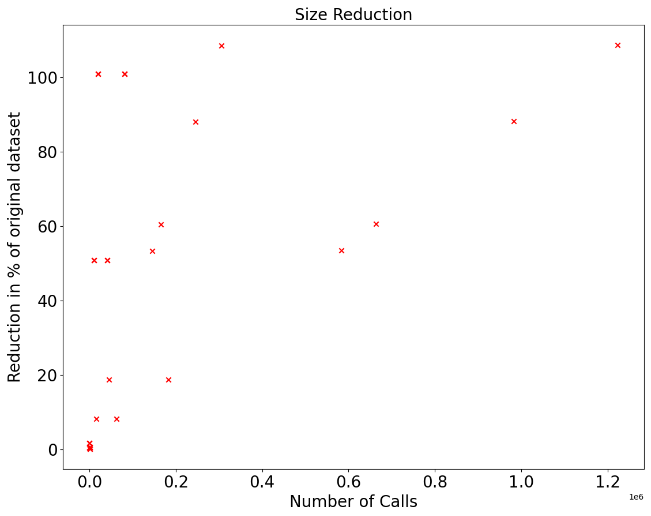


Figure 6.14: Dataset size reduction for Synthetic Benchmarks

6.3.2 Real life Experiments

Time Overhead

In our real-life experiments, where the applications involve not only I/O but also other calculations, we observed that the time overhead during both the audit and re-execution phases was significantly lower compared to the synthetic benchmarks. This disparity in overhead can be attributed to the additional computational tasks performed by the real-life applications, which reduce the relative impact of the I/O operations on the overall execution time.

The reduced time overhead in real-life applications suggests that the data debloating approach can be effectively integrated into practical scenarios without significantly affecting the overall performance. These findings highlight the importance of considering the specific nature of the applications when assessing the impact of data debloating techniques.

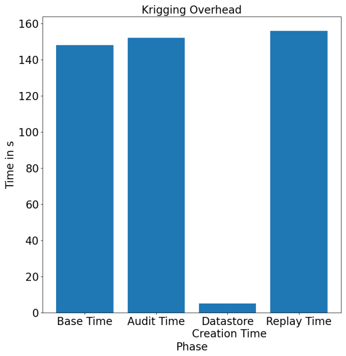


Figure 6.15: Krigging Time Overhead

Scenario	Time(s)
Base	148.32
Audit	152.71
Carving	5.07
Re-Execute	156.54

Figure 6.16: Krigging Time Overhead raw data

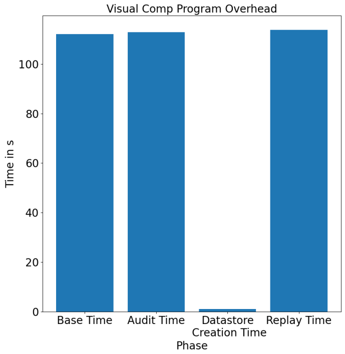


Figure 6.17: Visual Comparison Time Overhead

Scenario	Time(s)
Base	112.02
Audit	112.78
Carving	1.01
Re-Execute	113.82

Figure 6.18: Visual Comparison Time Overhead raw data

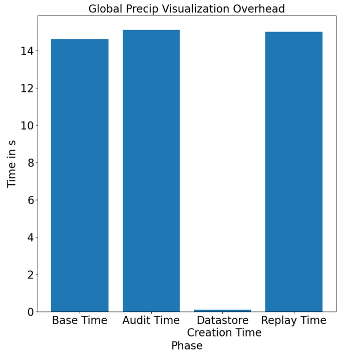


Figure 6.19: Global precipitation Time Overhead

Scenario	Time(s)
Base	14.61
Audit	15.11
Carving	0.10
Re-Execute	15.01

Figure 6.20: Global precipitation Time Overhead raw data

Real Life Experiments Dataset Reduction

In our analysis of real-life datasets, we compared the sizes of different components: the original dataset, the subsetted dataset, the sciunit container (without data debloating), and the sciunit container with data debloating.

The results demonstrated the impact of data debloating on dataset size reduction. Specifically, the subsetted dataset was smaller in size compared to the original dataset, indicating the elimination of unnecessary data. Furthermore, the sciunit container, which included the subsetted dataset, showed a reduction in size compared to the container without data debloating.

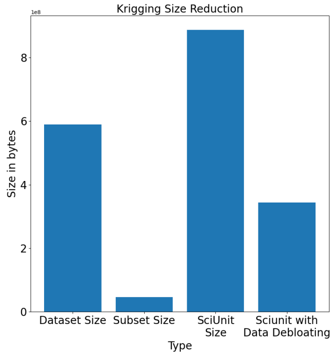


Figure 6.21: Krigging Size Reduction

Scenario	Size(MB)
Base	562.08
Subset	43.67
Sci-Unit	846.82
Sci-Unit with Debloat	328.41

Figure 6.22: Krigging Size Reduction raw data

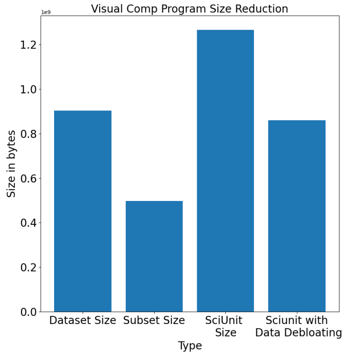


Figure 6.23: Visual Comparison Size Reduction

Scenario	Size(MB)
Base	861.03
Subset	473.89
Sci-Unit	1208.31
Sci-Unit with Debloat	821.16

Figure 6.24: Visual Comparison Size Reduction raw data

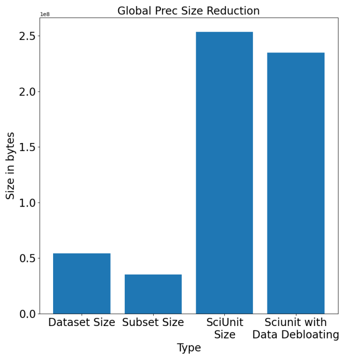


Figure 6.25: Global precipitation Size Reduction

Scenario	Size(MB)
Base	51.69
Subset	33.77
Sci-Unit	241.83
Sci-Unit with Debloat	223.91

Figure 6.26: Global precipitation Size Reduction raw data

Chapter 7

Future Work

7.1 Integration with SciUnit

Integrating our system with SciUnit offers several advantages. Currently, our system relies on the LD_PRELOAD mechanism to intercept system calls, which may result in intercepting more calls than necessary. By leveraging SciUnit's p-trace interception system, we can benefit from its targeted and efficient interception capabilities. SciUnit's interception mechanism is specifically designed for tracing and intercepting system calls, ensuring that only the relevant calls are intercepted. This integration will optimize the interception process, reducing unnecessary overhead and improving the overall performance of our data debloating system.

7.2 Interception of More I/O Sys Calls

Expanding the interception of I/O system calls is crucial for enhancing the versatility and applicability of our data debloating system. While our current system focuses on intercepting specific I/O calls used by the HDF5 interface and standard I/O operations, integrating with a comprehensive interception tool like SciUnit allows us to intercept a broader range of I/O system calls.

This expanded interception capability enables us to handle files of various sizes, shapes, and formats, making our system more adaptable and effective in debloating datasets across different domains and applications.

7.3 Integration of Containerization for Multiple Traces

Our current system supports the conversion of a single trace into a container. However, integrating containerization for multiple traces brings additional flexibility and utility to our approach. By allowing the combination of multiple traces, we can create containers that encapsulate the necessary subset of data from multiple sources. This functionality opens up possibilities for creating more generic and multipurpose containers that can be used across different scenarios and applications. The ability to utilize multiple traces in a single container expands the versatility of our system and enables more efficient sharing and distribution of debloated datasets.

7.4 Creation of Generic Containers

Our current solution is good for specific cases where we want containers to work with specific parameters (T). However, we want to improve it for more versatility. In the future, we aim to create generic containers that can execute a program (P) on different datasets (D) with a wide range of parameters (T). To achieve this, we plan to use advanced techniques like program analysis, invariant detection, and code fuzzing. These methods will help us understand how the program interacts with the dataset and the constraints it has on accessing the data. With this information, we can build containers that adapt to different parameter combinations, making them more flexible for diverse analyses. Our goal is to enhance containerization, allowing users to explore and analyze programs on various datasets and parameter sets with ease.

Chapter 8

Conclusion

In conclusion, this thesis has focused on data debloating for containers, addressing the challenge of efficiently sharing data-intensive applications. Through novel techniques, we have successfully demonstrated the effectiveness of data debloating in reducing container sizes and optimizing resource usage. By identifying and removing unused data components, we have improved container performance and storage utilization, facilitating easier container sharing.

The research presented here highlights the importance of data debloating in the future of containerization. As data generation grows, efficient container sharing becomes crucial for collaborative research and reproducibility. The proposed solutions contribute to the advancement of containerization technologies, and future work could explore integrating program analysis for more versatile containers. Data debloating offers promising possibilities for lightweight and efficient containers that cater to the evolving demands of data-intensive applications.

Bibliography

[1] Geos-5 a high-resolution global atmospheric model. `https://earthobservatory.nasa.gov/images/44246/geos-5-a-high-resolution-global-atmospheric-model`. [Online; accessed 8-Jun-2023].

[2] The National Aeronautics and Space Administration. Nasa global precipitation measurement. `https://gpm.nasa.gov/`. [Online; accessed 30-Sep-2022].

[3] Raza Ahmad, Madeline Deeds, Tanu Malik, Young-Don Choi, Jonathan Goodall, and David Tarboton. Sciunit: A reproducible container for EarthCube community. jul 2020.

[4] Johan De Haan and Oeoijn Veefkind. Profile 1-orbit l2 swath 13x48km v003. 2009.

[5] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 380–394, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-*

Directed and Runtime Optimization, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[7] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[8] Chaitra Niddodi, Ashish Gehani, Tanu Malik, Sibin Mohan, and Michael Lee Rilee. Iospred: I/o specialized packaging of reduced datasets and data-intensive applications for efficient reproducibility. *IEEE Access*, 11:1718–1731, 2023.

[9] Prajjwal Pathak. Global mean precipitation imerg analysis. ://github.com/pyGuru123/WinterLong-2021/tree/main/Global [Online; accessed 20-Mar-2023].

[10] Podman. Podman. `https://podman.io/`. [Online; accessed 2-Jun-2023].

[11] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through Piece-Wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 869–886, Baltimore, MD, August 2018. USENIX Association.

[12] Kyle Quest. Slimtoolkit by slimai. `https://slimtoolkit.org`. [Online; accessed 10-Jun-2023].

[13] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 329–339, New York, NY, USA, 2018. Association for Computing Machinery.

[14] The HDF Group. Hierarchical Data Format, version 5, 1997-NNNN. https://www.hdfgroup.org/HDF5/.