Access-Based Carving of Data for Efficient Reproducibility of Containers

Rohan Tikmany[a], Aniket Modi[a], Raffay Atiq[b], Moaz Reyad[a], Ashish Gehani[c], Tanu Malik[a]
[a]School of Computing, DePaul University, Chicago, IL, USA
[b] Lahore University of Management Sciences, Lahore, Pakistan
[c] SRI International, CA, USA
Email: *{rtikmany, amodi10, mabdelh4@depaul.edu, raffayatiq2341@gmail.com, gehani@csl.sri.com, tanu.malik@depaul.edu}*

***Abstract*—Scientific applications often depend on data produced from computational models. Model-generated data can be prohibitively large. Current mechanisms for sharing and distributing applications, such as containers, assume all model data is saved and included with a program or is downloaded during build time to support its successful re-execution. However, including model data increases the sizes of containers. This increases the cost and time required for deployment and further reuse. We present `ABCD` (Access-Based Carving of Data), a framework for specializing I/O libraries which, given an application, automates the process of identifying and including only a subset of the data accessed by the program. To do this we show how such specialization can be achieved at two levels of granularity: at a library level and at a system call level. The different levels help to include data for a single parameter run or over several parameter runs. We show several orders of magnitude reduction in data size via the specialization of HDF5 I/O libraries associated with model-based data-intensive applications, such as those operating on precipitation and geophysical data.**

***Index Terms*—Reproducibility of results, Application virtualization, Big Data applications**

I. Introduction

Reproducible applications reduce the time for validation. This increases the trust in their results. However, reproducing data-intensive applications continues to be a challenge in the domain sciences. We consider data-intensive applications that can be entirely reproduced when provided along with their code, data, and environment specifications. We focus particularly on applications that access a large amount of structured data, using custom I/O-optimized libraries. Such applications are common in the scientific domain, where data is stored in formats such as NetCDF4 (1), HDF5 (2), or UniProt (3). Most of these formats are self-descriptive, that is, the data and metadata are stored in the same file. Being self-descriptive, they require specific I/O libraries for reading and writing these formats to be included with the application.

Consider such a scientific application, consisting of a simulation and analysis phase. The simulation phase uses a computational model to simulate a physical phenomenon, and usually generates large volumes of multi-dimensional data consisting of attributes such as latitude, longitude, temperature, pressure, and humidity. The analysis phase projects future changes in the environment, such as improving weather or flood forecasting. Our experience with such analysis applications shows that applications often do not use all the datasets generated from the simulation phase, but only a few of them. However, in a containerized form, all data generated from the simulation phase are entirely packaged into a single file leading to *container bloat*.

Container bloating, in general, can arise due to application code, libraries, data files, or all three. For example, machine learning software like TensorFlow (4), Scikit-learn (5) bundle a large number of libraries to be used by a variety of applications. Similarly, HDF5 (2) and Avro (6) data formats allow multiple data files/datasets to be bundled together. There is also a tendency by well-known agencies to include and disseminate extra data in standard data files of which only a small part may ever be accessed by a given application. Several recent works (7; 8; 9; 10) have determined that containerized versions of even simple applications come close to or above a gigabyte, leading to high storage and network transfer costs, and increased security risk (10).

Common methods to reduce bloat are compression and deduplication. Deduplication, however, only removes repeating data from a dataset. Compression reduces the size of the containers but compressed data must be decompressed for use within a container. An alternative way is to use lineage-based methods (11; 12; 13) which analyze data access patterns, but these methods also work at the coarse-grained level of identifying whether a data file is accessed or not, not what portions of it are accessed. Consequently, container bloat overall remains a pressing challenge in research and in practice.

By interposing on I/O calls and adding wrapper functions that re-map I/O access from the original data file to the carved file, we can introduce arbitrary *specialized* behavior. Since analysis applications access simulation data, which is stored in self-describing files, we consider two types of specialization of I/O calls – either by interposing at the format-specific I/O library level or by interposing, generically, at the system call level. The advantage of the latter is that it is a precise interposition leading to necessary and sufficient data accessed by the application and introduces no changes to the binary of the source code; the former interposition, however, is often preferred especially if the application is meant to be distributed such that it can be re-executed repeatedly with different input parameters. In both cases, however, the carved data is robust only to the subset of dataset objects accessed and used. Any data not accessed but needed for peruse purposes must be dynamically loaded.

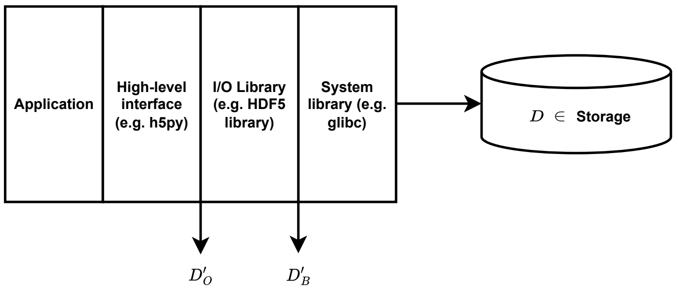
We now present Access-Based Carving of Data (ABCD), which monitors accesses to relevant data and extracts and stores a single copy of only necessary data based on given program parameters. These are data chunks accessed by the application while executing with the specified user inputs. ABCD provides two levels of interception: system-level and object-level. System-level interception determines the necessary and sufficient data for a given reference execution of input parameters. Object-level interception determines objects accessed at the level of self-describing data files used by the application.

The approach of interposition adopted in ABCD is distinct and orthogonal to both deduplication and compression, commonly used methods for data reduction. In these methods, duplicate data chunks not needed are eliminated via hashing of current data chunks, and if necessary data chunks can be compressed within a container. While ABCD does not suffer from the issue of fragmentation, which is a major limitation of deduplication algorithms (14; 15; 16), it can still adopt deduplication.

The paper makes the following contributions:

- Developing a system call interception method that identifies bytes of data read and written. Only non-overlapping extracted bytes of data are stored in a debloated/carved file using an interval binary search tree index. The carved data file is stored within the container, and repeated executions are mapped to offsets in the carved file. We describe how write operations maintain the consistency of the file. We also describe an implementation which works at the granularity of data *chunks*, the lowest granularity of access in HDF5-based data files. [Section III].
- Developing an object interception method that identifies and extracts data objects read during application execution. The object interception method operates at the I/O library level. This method is complementary to system call interception method in that it provides coarser granularity of carved data file (i.e. more data is carved), which can be useful when re-executing the container with varying input parameters. [Section IV].
- Building a complete containerized system that includes the application along with the carved file and mechanisms to interpose and map to carved file. [Section III-C and IV-B].
- We evaluate our framework on realistic and synthetic benchmark applications across multiple configurations. These applications operate on NASA datasets (in the order of gigabytes) in two different data formats with varying data access patterns to demonstrate the wide range of applicability of ABCD. [Section V].

Our results show that ABCD can achieve up to 97% reduction on accessed data while still identifying and including all required data for the provided user inputs to ensure reproducibility of application executions.



Application
High-level interface (e.g. h5py)
I/O Library (e.g. HDF5 library)
System library (e.g. glibc)
$D \in$ Storage
D'_O
D'_B

Fig. 1: Application stack

II. Container Storage Debloating Problem

Consider an application A that accesses a data file D using input parameters T. An execution of an application can be described as the result of the analysis performed by the application, such that $\forall T_i,\ A(T_i)$ accesses subset $D' \subseteq D$. The goal of debloating is to extract the subset D' to achieve reduction in container storage and distribution footprint, with guaranteed re-execution of the application for the particular set of parameters.

We explore two approaches that operate at different levels in the application stack. One approach operates at the granularity of byte offsets at the file level, which can be obtained from system calls. Let B_i be the set of all bytes read by an application, where bytes may refer to data bytes or metadata bytes. With this approach, the debloated subset of the data file would be as follows:

$$D' = \bigcup_{i=1}^{n} B_i$$

The other approach operates at the granularity of data objects at the library level. Let O_i be the set of data objects read by an application. In this case, the debloated subset of the data file would be as follows:

$$D' = \bigcup_{i=1}^{n} O_i$$

Considering the example of the HDF5 library, in which one of the key data objects are *datasets* (17) so D' corresponds to the set of HDF5 datasets accessed by the application.

We leverage library interposition to operate at different levels in the application stack, as shown in Figure 1. Specifically, to achieve debloating at the granularity of byte offsets, D'_B, we interpose at the system call level, and to achieve debloating at the granularity of data objects, D'_O, we interpose at the I/O library level. Debloating data at different granularities leads to different container size reductions. High-level data objects are at a coarser granularity and consist of bytes which are at a sharper granularity i.e. $D'_B \subseteq D'_O$.

Figure 2 shows an example of an analysis application that accesses a data file, but ‘uses‘ some part of it only. For example, for the source code shown in Figure 2, lines 6-7 indicate access of whole data objects whereas lines 8-9 indicate access of the subset of those data objects via index-based subsetting and range-based subsetting. Now this

```
1  import h5py
2
3  def analysis(HDF5_filename):
4    HDF5_file = h5py.File(HDF5_filename, 'r')
5
6    data_object_1 = HDF5_file['data_object_1']
7    data_object_2 = HDF5_file['data_object_2']
8
9    data_object_subset_1 = data_object_1[0:1000]
10   data_object_subset_2 = data_object_2 > 50
11
12   return result(data_object_subset_1, data_object_subset_2)
```

Fig. 2: Application Source Code

application, when re-executed may access data in different ways, which can include varying the data objects that are accessed or the range criteria which defines the subset of data objects that are accessed.

To determine the necessary and sufficient data, we can define the execution context of the application. The execution context motivates the need to interpose at a particular level. If the execution context operates at lines 6-7, it means that the execution context of the application varies the data objects accessed and the application intends to perform analysis on a fixed subset of data corresponding to *data_object_1* and *data_object_2*. Since the subset of data objects accessed are fixed in this case, it makes D'_B suitable for the given execution context as this will debloat at the granularity of the subset of data objects. On the other hand, if the execution context operates at lines 9-10, it means that the execution of the application varies the subset of specific data objects and the application intends to perform analysis on the subset of those particular data objects. Since the data objects are fixed in this case, it makes D'_O suitable for the given execution context as this will debloat at the granularity of data objects.

In addition, each level of the application stack exposes different functionality which dictates the mechanism of debloating. For example, at the system call level, additional metadata needs to be maintained in the form of an audit trace after application execution in order to extract the subset of data accessed. On the other hand, at the object level, I/O libraries often allow complete access to data objects which allows the interposition to extract the subset during application execution, without the need for additional metadata.

III. System Call Level Interposition

We describe how system call function interposition helps to determine the minimum amount of data accessed (Section III-A). Given a representation of the minimum amount of data accessed, we describe how a carved datafile corresponding to the minimum amount of data accessed is created and mapped during re-execution (Section III-B). Finally, we describe a complete architecture and implementation details of system call function interposition (Section III-C).

A. Determining Minimum Data Accessed

To determine the minimum amount of data accessed, we define each file I/O related system call that is interposed as an *event*, defined as:

Definition 1. ***Event****. A file I/O 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 the event,*
- *c signifies the type of system call,*
- *l marks the start offset location in the file which the event affects,*
- *sz denotes the size of the file affected by the event starting from l, and*
- *h represents the SHA-256 hash of the buffer contents during read and write events.*

Events are predominantly `read` or `write` or `lseek` events and they determine the byte offsets and size of data accessed. Each application execution results in an ordered set of events that fully captures all events during the execution of a given application A. Formally, $A = \{e_1, \dots, e_n\}$. Events in a trace can be merged if they overlap as per a *merge* operation defined as:

Definition 2. ***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*
- *$e_2(l) \leq e_1(l) < e_2(l) + e_2(sz)$.*

If *all* file I/O events are audited, then the application audit trace is deemed complete. Further, if traced I/O events are merged, then the resulting trace corresponds to a carved file or the minimal data requirement of a deterministic application, defined as:

Definition 3. ***Minimum Data Requirement.*** *The minimum data requirement (MDR) of application A, with user parameters T, on dataset D, is the minimum subset of data from dataset D, which is required to execute application A with user parameters T such that both produce consistent results R.*

There are two issues however in computing the minimum data requirement. First, during the execution of the application, there may be modifications to the data file. These changes can pose integrity and accuracy challenges when creating an appropriate subset for re-execution. Second, with a large number of events, it is important to minimize event lookup overhead during merge operation and during application re-execution to check validity of an event. It is important to choose a data structure that balances lookup efficiency and memory usage. We describe how we address these concerns.

1) Management of Writes: Writes alter the original data file D, which creates issues during the creation of the MDR. Thus it is important to preserve the state of D pre-execution. In our data debloating process, we consider two approaches for MDR creation based on the availability of the dataset D post-execution, i.e., if D is preserved in the same state as it was pre-execution. If the dataset D is available post-execution (for e.g., it can still be downloaded), we only need the audit trace to identify the accessed offsets and generate the

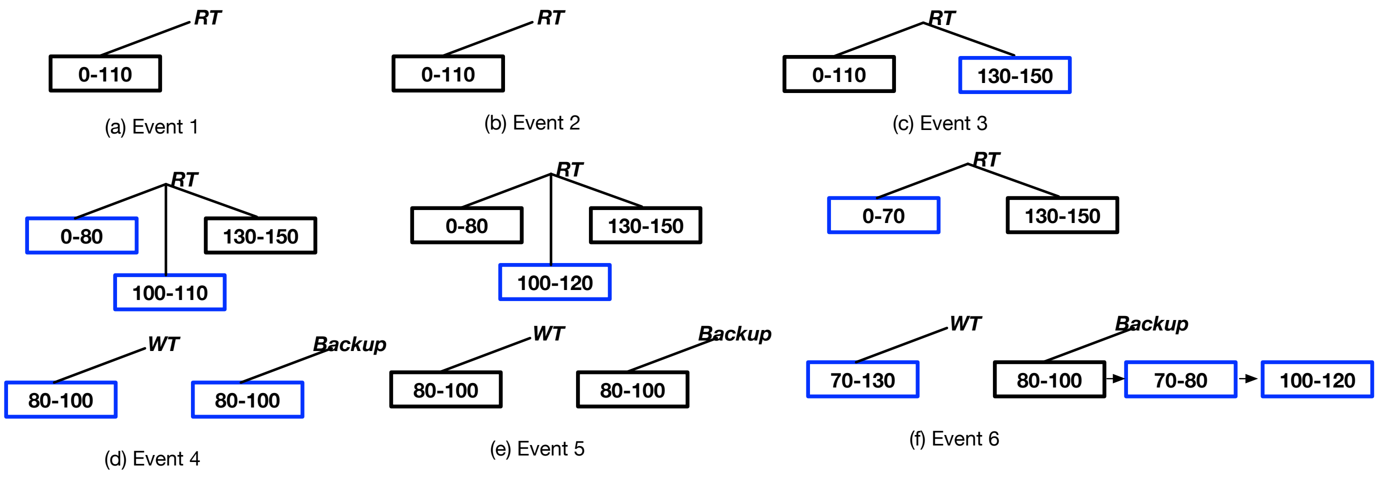


Fig. 3: Read Tree and Write Tree as the events in the audit trace of Table I are processed. Blue boxes present changes in the tree from the previous event.

MDR. However, if D has been overwritten during execution and the original state of D cannot be obtained, then we need to maintain backups of the portions of data that were overwritten so we can generate the MDR correctly. Currently, ABCD leaves it for the user to decide whether or not to create backups.

To create backups ABCD identifies overlapping writes defined as:

Definition 4. ***Overlapping writes*** *There is a read-write overlap when:*

- $e_1(c)$ = *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 an overlapping write.

ABCD creates backups for those portions of the file that are about to be overwritten by an overlapping write as defined in Definition 4. These backups serve as snapshots of the original content *before* any overlapping write is made. By preserving the original data, we ensure that the necessary portions required for accurate re-execution are available even if overlapping writes occur during application execution. 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.

2) Lookup Data Structures: To effectively track the offsets that have been read and written(these can be in the order of millions), we consider interval binary search trees (BST). By treating the read and write offsets as intervals, an interval BST has logarithmic time complexity for searching, inserting, and deleting intervals. To track offsets, we utilize two interval trees: a read tree RT for read offsets that have been *purely* read and a write tree WT for write offsets that have been overwritten. In addition to the interval trees for tracking read and write offsets (RT and WT), we also maintain a linked list for backups to store original data portions.

The trees are updated as the events arrive. For the i^{th} read event, we ensure that only the offsets that have not been modified or overwritten are included in RT. This avoids looking up into the write tree during a read event and obtaining the offsets that have not been written over during the audit phase. In this form, the read tree represents the offsets in the file's original state.

For the i^{th} write event, ABCD creates a backup for any write operation that is overlapping *any* previous read which has not yet been overwritten. Thus, ABCD identifies the offsets that are both in write e_i and in RT, indicating the portion of the file that is being overwritten. A backup is created for this section to preserve the original data. For the same write event, ABCD adds the write offset interval into the WT. It then removes the section written to by the write event from the read interval tree RT. 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. Note backups store the actual data and are like versions preserving the original data before a write. Writes themselves need not be preserved as they will happen again.

We use an example to describe the state of the trees and the backups created. Table I shows a sequence of read/write events from an application to a data file D, with their start offsets and the data size affected. Figure 3 shows the state of the read and write tree and backup list as the events are processed. The second read event is completely overlapping and leads to no change in the read tree. The two write events lead to creation of backup data. The state of the write tree at the end of events is the total written offsets which is 80-100 and 70-130, and the state of read tree is unwritten offsets. The backup is incremental and can lead to any intermediate state of the file D.

TABLE I: Sample Application Read/Write Events

Event #	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

Let the original dataset be D. After the i^{th} write event e_i^W let the modified dataset be D_i. Let the union of all the reads between the write event e_i^W and e_{i+1}^W be R_i. Using this we can represent our audit trace as:

$$D \xrightarrow[R_0]{e_1^W} D_1 \xrightarrow[R_1]{e_2^W} D_2 ... \xrightarrow[R_{n-1}]{e_n^W} D_n$$

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

$$MDR = R_0 \cup (R_1 \cup (\ldots R_{n-1})) - w_2) - w_1 \qquad (1)$$

For any two consecutive version of the dataset D_i and D_{i+1} created due to write events e_i^W and e_{i+1}^W, if we subtract the region of impact of e_{i+1}^W which is $\{e_{i+1}^W(l), e_{i+1}^W(l)+e_{i+1}^W(sz)\}$ from the dataset version D_{i+1} then the leftover parts of the dataset are a subset of D_i. As we subtract all e_i^W we can recursively apply the formula above to show that MDR contains only the data elements from D.

B. Re-execution with Minimum Data Required

To re-execute with the minimum data required, it is sufficient to combine the metadata corresponding to accessed offsets of the read tree and the backups since per equation 1 only those offsets are needed. Also, intuitively, these offsets represent the original data file. Figure 4 shows the combined tree of offset ranges at the top. At the bottom, the corresponding data offset chunks from the RT and the backups is shown, where in the solid boxes represent ranges from the RT, and the grey boxes represent the backups, which are combined to range 0-120. Note, if there were no writes issued by the original application A there will be no grey boxes.

We term it the *combined metadata tree*, as ABCD adds further information to this tree, such as file names or identifiers, which facilitates the accurate alignment of the subset data with the corresponding sections of the original dataset. During re-execution, as each system call is intercepted again, ABCD looks into the ranges of the combined metadata tree M to find the appropriate data chunk. Thus for example for each read event, ABCD simply intersects the requested offsets in the read event $\{e_i^R\}$ with the metadata tree M, to obtain the relevant metadata nodes associated with the read operation. Since offsets may be scattered, ABCD concatenates the retrieved data and returns it to the application. This ensures that the application receives the necessary data for the read operation.

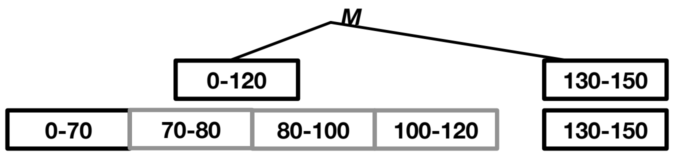
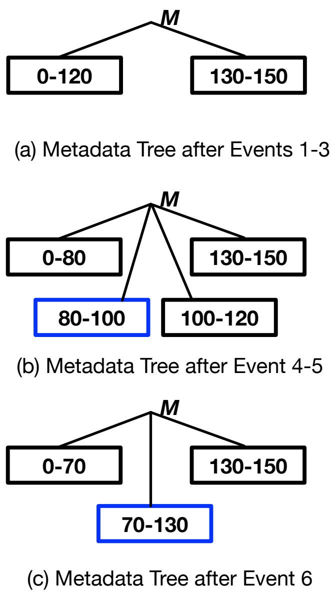


Fig. 4: Combined Metadata Tree

For write operations, since this is the re-execution phase, all such write operations are needed if the user wishes to explore the update file. However, the user should also have the option to perform repeated re-execution. Thus, on the i^{th} write event, denoted as e_i^W, ABCD flushes the corresponding write operation to a write cache and updates the metadata tree to reflect that the corresponding chunks of data are unavailable for the subsequent read. We note that there is no change to the carved data file at this stage and such bookkeeping is only at the level of metadata.

Figure 5 shows re-execution and state of M for sample execution in Table I.



M
0-120
130-150
(a) Metadata Tree after Events 1-3
M
0-80
130-150
80-100
100-120
(b) Metadata Tree after Event 4-5
M
0-70
130-150
70-130
(c) Metadata Tree after Event 6

Fig. 5: State of Metadata Tree during Re-execution. Blue boxes represent the state of the Write Cache represented in the Metadata Tree.

We finally want to state that the recording and lookup data structures and metadata subset tree create two structurally preserving executions, one on the original data file and the carved data file. In other words the order of events generated from a deterministic program on the original data file with a given set of input parameters, is homomorphic to the order of events generated from the same deterministic program on the carved data file with the same set of input parameters.

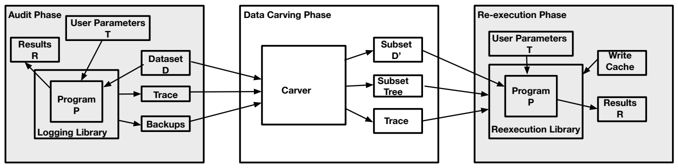


Fig. 6: Full Architecture of System Call-level interposition

C. *Architecture of System Call Level Interposition.*

Figure 6 shows the architecture of the system call interposition method. The initial phase, known as the *audit phase*, involves monitoring the application while executing and capturing the I/O lineage at a byte-level granularity, forming the *Trace*. By employing function interposition (via `LD_PRELOAD`), our tool dynamically inserts a library into the application. Following the Audit Phase, the `ABCD` uses the recorded events to create a carved data file. In this data carving phase, the tool creates a combined metadata/subset tree which has the necessary mapping to map the calls to the original dataset (D) to the newly created subset/carved file (D'). The data carving phase occurs immediately after application execution is over and the trace is generated. In the final phase, referred to as the *re-execution phase*, the application re-executes with (D'). This phase also uses function interposition (via `LD_PRELOAD`) to dynamically insert a library into the application to intercept all I/O calls and redirect the reads to the carved file (D') and writes to the write cache. In this architecture, it is possible to combine the audit and data carving phase, though we keep them separate and take a lazy procedure to carve data files to not impact the application performance.

Thus given the example program shown in Figure 2, the user whitelists the paths of the data files that are to be utilized by the application. It then audits the Python script as:

```
LD_PRELOAD=./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 and thus all system call events are interposed and recorded. A subsequent script automates the carving process and creation of D'. The application is then re-executed using the optimized dataset created, but this time loads a repeat library shared object, `repeatLib.so`, into the application's execution environment using the `LD_PRELOAD` directive, as illustrated below:

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

The application is then launched with the optimized dataset, ensuring accurate execution and improved re-execution time.

IV. I/O Object Level Interposition

Interposition at the library object level forces the finding of the minimal data requirement to the implementation of the library itself. This restricts the data debloating to high-level data objects defined by the library, as well as the library functions.

We consider such data objects at the HDF5 library call level. We wrap additional functionality around the original functionality of library functions to carve the exact subset of data accessed. Through this approach, we achieve data debloating at the granularity of HDF5 data objects without modifying the application. The application can be re-executed with the carved HDF5 file with guaranteed reproducibility. The system additionally implements fallback machinery to maintain correctness should the application access data outside of the subset accessed in the original execution.

A. *Identifying and Carving HDF5 Library Objects*

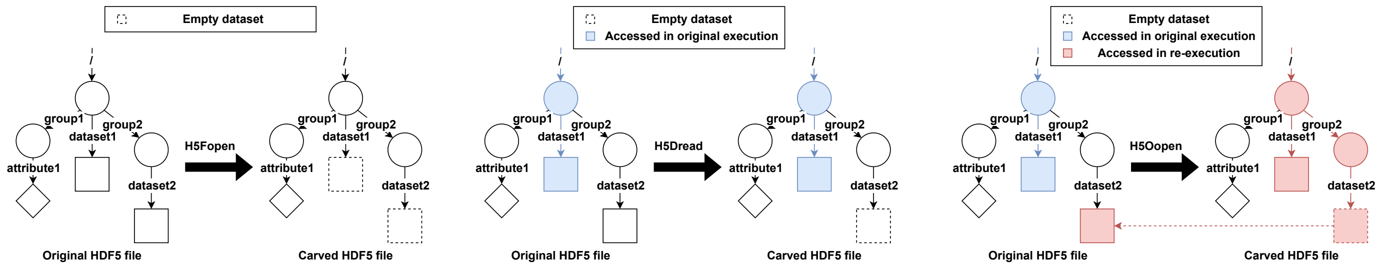
The HDF5 library defines a data model that includes two primary data objects: *groups* and *datasets*. Groups are containers that hold datasets and other groups, providing structure and organization in an HDF5 file. Datasets are multidimensional arrays that store data. HDF5 also provides small metadata objects called *attributes* that can be attached to groups and datasets. Attributes make HDF5 a self-describing format, allowing complicated data to be stored with all the information needed to describe the data.

Data objects in an HDF5 file are organized as nodes in a directed graph structure rooted at the "/" group, and each object is connected via named *links*. This structure allows the objects in the HDF5 file to be accessed in a syntax similar to how directories are accessed in Unix file systems.

The HDF5 library parallels its data model by following an *object-oriented* approach in its function naming convention. Each function is prefixed with "H5" denoting the HDF5 library, followed by the object it operates on, and ending with the purpose of the function. For example, the function *H5Gopen* is used to open groups. Since each application that uses HDF5 interfaces with the library, it presents an opportunity to strategically interpose at particular function calls to implement additional functionality. This is leveraged for the purpose of carving data that is accessed.

We interpose on three functions: *H5Fopen*, *H5Dread*, and *H5Oopen*. For each of these functions, additional functionality is wrapped around the original functionality via library interposition, as follows:

- **H5Fopen**. This function is used to open HDF5 files. The additional functionality includes making an identical "skeleton" copy of the HDF5 file via a depth-first search into the directed graph structure, as shown in Figure 7a. The copy is a carved version of the original HDF5 file and includes all groups, datasets, and attributes of the original file but excludes the contents of the datasets.
- **H5Dread**. This function is used to read datasets from an opened HDF5 file. The additional functionality includes monitoring which datasets have been accessed and populating the empty datasets in the carved file with the contents of the datasets accessed in the original file, as shown in Figure 7b.
- **H5Oopen**. This function is used to open an object within an HDF5 file, serving as a preliminary call to fetch the identifier of the object to be accessed. The additional functionality



(a) H5Fopen interposition (b) H5Dread interposition (c) H5Oopen interposition

Fig. 7: HDF5 library call interposition

includes determining whether the datasets accessed during re-execution were the same as in the original run. If the datasets were accessed previously, the carved version of the HDF5 file is used. Otherwise, fallback machinery is invoked.

An HDF5 file may be available in a persistent repository. Carving and including content from such a file provides multiple benefits, including reducing the storage footprint of a container, improving access latency (and hence re-execution runtime), and eliminating the dependency on external content, making the resulting package self-contained.

In the event that program parameters during re-execution differ significantly from the original run, datasets may be needed that are not present in the carved version. Internally, the HDF5 library uses a collection of different drivers, depending on how the content is being accessed. For example, local files are accessed with the *SEC2* driver (18) by default. Files hosted on Amazon S3 storage are accessed via the *ROS3* driver (19). This allows alternative behavior to be defined in the case of missing datasets.

Currently, missing datasets in S3-hosted HDF5 files are automatically retrieved. In the future, support can be added to allow the user to map the location of local HDF5 files to a fallback URL. This approach allows the correctness of re-executing applications to be ensured in as wide a set of circumstances as feasible, as shown in Figure 7c.

B. Architecture of Library Object Interposition

The carving system operates in two modes: *execution* and *re-execution*.

In execution mode, the application is run with the original HDF5 file. `ABCD` outputs a carved version of the original HDF5 file that contains only the subset of data accessed by the application. Let $S_{\text{reduction}}$ be the reduction in container storage size, D_{original} be the size of the datasets in the original HDF5 file, and D_{accessed} be the size of the datasets accessed in the original execution. The total reduction in the storage size of the container is equivalent to the complement of the subset of data accessed, as follows:

$$S_{\text{reduction}} = D_{\text{original}} - D_{\text{accessed}}$$

In re-execution mode, the application is run with the carved HDF5 file instead of the original HDF5 file. The original HDF5 file is only opened if the application attempts to use data outside the subset accessed in the original execution.

Name	Data Format	Language	Description
`AIST`	HDF5	Python	(20)
`Krigging`	HDF5	Python	(21)
`GlobalPre`	HDF5	Python	(22)

TABLE II: Real Life Applications

V. Experiments

In this section, we detail the comprehensive experiments conducted to evaluate the efficacy of `ABCD` in discerning data bloat within programs. The experimentation encompassed two distinct categories: real-life applications and synthetic benchmarks. All tests were executed on a Ubuntu 22.04 machine boasting eight cores, 15GB of memory, and an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, ensuring a standardized computing environment.

Implementation All programs involved in the experiments utilized HDF5 data files. `ABCD` system is developed in Python 3.8 and C, with the C language used for system call auditing and redirection, and Python used for the intermediate data store creation. For system call interception, the system uses LD_PRELOAD mechanism to determine accessed environment and data files. During re-execution the system maps a system call's arguments to the appropriate offset of the debloated file. For library interception calls to H5Fopen, H5Dread and H5Oopen are recorded along with input arguments which are subsequently stored in a data store as a debloated file.

A. Real Life Applications

To provide a robust assessment of our data debloating system's capabilities, we subjected it to three real-life applications sourced from precipitation domains. Table IV provides details of these three applications describing the format of data that they access, the GitHub repository from where they were acquired, and the programming language they were developed in.

B. Synthetic Benchmark

We used four micro-benchmarks programs available from the h5bench library (23). H5bench is a suite of parallel I/O benchmarks or kernels representing I/O patterns commonly used in HDF5 applications on high-performance computing systems. I/O patterns *i.e.,* the order in which locations in the data file are accessed and how much data is read in one

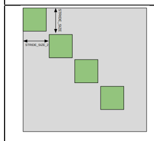
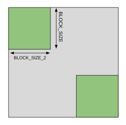
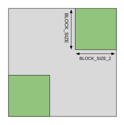
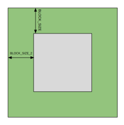
CS	LDC	RDC	PRL
			

TABLE III: Illustrations of the benchmark patterns

Pattern	Parameter	Description
CS/LDC/RDC	BLOCK_SIZE	Data block size along dim_1
CS/LDC/RDC	BLOCK_SIZE_2	Data block size along dim_2
CS	STRIDE_SIZE	Data block stride along dim_1
CS	STRIDE_SIZE_2	Data block stride along dim_2
PRL	BLOCK_SIZE	Frame Size along dim_1
PRL	BLOCK_SIZE_2	Frame size along dim_2

TABLE IV: Synthetic Benchmark Parameters

access are supported via a *stencil* data abstraction (24) and an *apply* function. Intuitively, a stencil represents a geometric neighborhood of an array in an HDF5 data file. Two types of stencils are provided: A solid rectangular shape and a rectangular shape with a hole. An I/O pattern is obtained when a program *applies* the stencil in different patterns. Two patterns are supported: In the fixed pattern, the application subsets a stencil worth of data from a set of specific locations in the file and in the iterative pattern it subsets by incrementally iterating (in a 'for' loop) over a set of different locations defined by a constraint in the program. The suite consists of four different micro-benchmark programs illustrated in Table III. Each micro-benchmark program within the suite mimics a scientific application.

Each pattern is detailed below:

1) **CS:** Cross Stencil pattern, involving a fixed-sided block with a specified stride in each dimension. Data is read from the HDF5 file till the end.
2) **LDC:** Left Diagonal Corner pattern, reading data from two identical blocks positioned in the top-left and bottom-right corners of the 2D HDF5 file.
3) **RDC:** Right Diagonal Corner pattern, similar to LDC but with blocks in the top-right and bottom-left corners of the 2D HDF5 file.
4) **PRL:** Peripheral pattern, reading data from the periphery of the file, forming a fixed-width and height frame around the dataset.

To run these benchmarks the C program accepted additional parameters required in the configuration for each pattern as listed in Table IV.

We run the benchmarks against a *ptrace* based interposition method. This interposition method is used in container pruning tools such as (25). The whole file is either included or excluded from the ptrace based interposition method. In our experiments, we measure the storage reduction because it is the main objective of ABCD. But we also measure the audit and re-execution time to ensure that both solutions work without excessive overhead costs.

Figure 8 shows the storage reduction percentage using byte-level and object-level methods for each program in the benchmark and real applications with respect to the original file. The system call level interposition gives a very high reduction

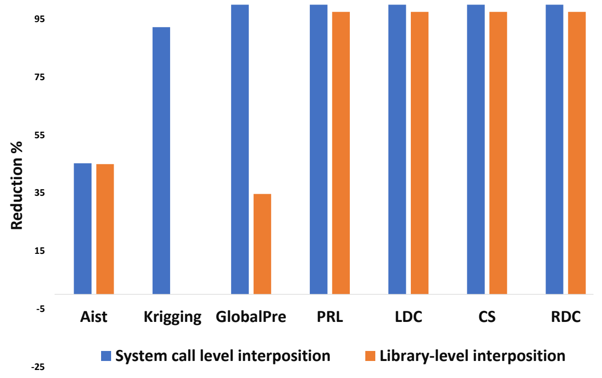


Fig. 8: Reduction Percentage

percentage. This is because the debloating at file I/O level uses low-level tracing of system calls to find the near to exact size of the required dataset. The library-level interposition depends on the attributes chosen by the application. In the case of GP, the difference is stark as the program further subsets on the chosen attributes which is identified by the byte-level interposition but not the object-level interposition. *ptrace*-based interposition is not shown as the reduction % is negative. The interposition increases the storage cost due to extra metadata per call that is included in the container.

For measuring the time cost, we show the audit time in Figure 9 and the re-execution time in Figure 10. Both figures show the time in the log scale. We calculated the time for the synthetic benchmark by taking the average of 10 runs with various parameters to the benchmark. The Audit and Re-Execution for the real-life applications is done without varying their parameters, hence there is no need to take an average and we reported the exact time value. Due to lack of space, we have combined the real and synthetic programs. We see positive log values for real applications as they access large data files and are compute-intensive. We see negative log values for synthetic benchmark programs which read small data files.

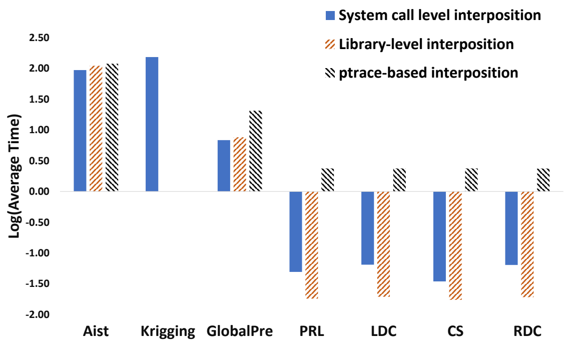


Fig. 9: Average Audit Time

We notice a significant difference between the real applications and the synthetic benchmark. The real applications take more than one second, so their log time is positive, while the synthetic benchmark runs in less than one second, hence their log time is negative. During the audit time, the system call-level interposition will always take more time than the library-level interposition because the number of system calls is significantly larger than the number of library calls. *ptrace*-based interposition auditing time is higher in the synthetic benchmark, but it is lower in the case of real applications.

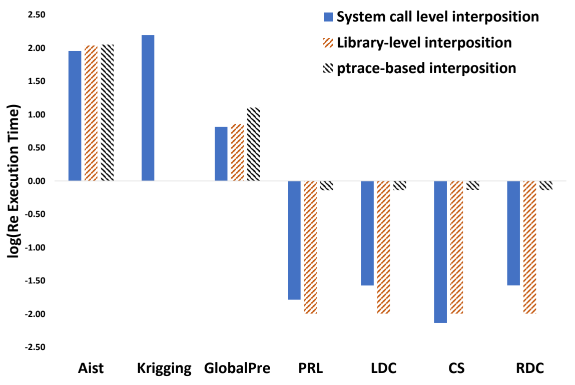


Fig. 10: ReExecution Time

The re-execution of both ABCD approaches takes less time than the *ptrace*-based interposition baseline. This is because *ptrace*-based interposition makes redundant system calls, while the ABCD provides a smaller debloated file specialized for this application. The container storage is reduced on the disk, and the run-time becomes significantly faster.

VI. Related Work

We focus on related work primarily relating to debloating.

Data-based Debloating. Redundant downloading of data within each layer is a known issue in container-based deployments, resulting in large size images (26). This has triggered research in the direction of data based debloating in user space. A related tool, Slacker (8), uses deduplication of file blocks to create more efficient container systems. However, block-level deduplication techniques do not eliminate data redundancies within structured arrays. Whereas, ABCD is designed to remove unused data in structured array-based data formats such as in NetCDF4 (1). Another work, LLIO (27), improves runtime performance of applications through elimination of filesystem accesses. However, it does so by lifting entire files. Such an approach becomes untenable for large data sets. In an attempt to mitigate this limitation, ABCD demonstrates the possibility of lifting only subsets of data (instead of entire files) and packaging it along with data-intensive applications.

Code-based Debloating. Study of code bloat in software with consequent debloating has received a lot of attention in the recent past. Xin *et. al.* (28) have analyzed the tradeoffs between code reduction and function generality in debloated software. Jiang (29; 30) have studied the issue of software bloat in real-world Android applications and have proposed static analysis based techniques to remove dead code. OCCAM (31) and Trimmer (32) have demonstrated that configuration based debloating can be applied to modern applications. These techniques primarily work on code written in C/C++ by automatically building them into LLVM bitcode. Unlike these code-based debloating techniques, ABCD explores data-based debloating for efficient storage space utilization and application reproducibility.

Program Specialization. A prior work on program specialization by Medicherla (33) has been used to verify whether or not a program 'conforms' to the specified format and to specialize the code to the 'restricted' file format. This is program specialization being used for verification. As opposed to this, program specialization is being used for data reduction in ABCD.

Lineage Methods. A precise lineage model informs about data flows of an application. There are several provenance models for capturing system call lineage (34). In this paper, we adopt the system-event trace analysis process which is also used in other whole system provenance tracking methods (11; 13; 35). None of these models, however, log and infer on data subsets in the system-event trace.

I/O level monitoring and access patterns. Darshan (36) is an I/O characterization tool used to profile and analyze I/O behavior in high-performance computing (HPC) applications. It is designed to collect detailed information about I/O patterns, file access patterns, and other relevant metrics to help users understand how their applications interact with the file system. Darshan's DXT module (37) can also instrument I/O calls in the application code to collect data on file operations. Thus, Darshan can be used to profile an application for I/O in the context of debloating. However, Darshan will also need to carve the byte-level information and reuse it. Determining how Darshan can interact with the carving and re-execution phase remains part of our future work.

Scientific computing programs supported by libraries like HDF5 often perform data subsetting. Several shapes of I/O access patterns are explored in (24; 38; 39). However, automatically examining the amount of data subsetting is unexplored. We have used available benchmarks based on these patterns to analyze how much data must be subset and containerized.

VII. Conclusion

In this paper, we presented the container storage debloating problem with examples from real-life use cases. Two solutions for this problem based on specializing I/O libraries are discussed in detail. While both solutions can significantly reduce the container storage footprint, each has advantages and disadvantages. We performed experiments to identify the differences between the two approaches and show when each can be more suitable.

The general solution both systems follow is to audit an application at runtime to discover its data-access requirements and create a specialized storage version for this specific application. By observing the application's calls, the specialized storage is optimized to respond only to the application needs later in the re-execution phase.

Our results will guide scientists and engineers who implement reproducibility for data-intensive applications through containerization. Storage debloating can positively affect other areas in high-performance computing, such as computing cost efficiency, improved security, simplified distributed systems, and reduced energy consumption. We plan to study some of these areas in our future work.

Acknowledgements

This work is supported by NASA under grant NASA-AIST-21-0095 and National Science Foundation under grants CNS-1846418, NSF ICER-1639759, ICER-1661918.

References

[1] R. Rew *et al.*, "Netcdf-4: Software implementing an enhanced data model for the geosciences," in *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanograph, and Hydrology*, vol. 6, 2006.
[2] M. Folk *et al.*, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, 2011, pp. 36–47.
[3] "Uniprot: the universal protein knowledgebase in 2023," *Nucleic Acids Research*, vol. 51, no. D1, pp. D523–D531, 2023.
[4] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: www.tensorflow.org
[5] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[6] Apache avro. [Online]. Available: avro.apache.org
[7] X. Wu *et al.*, "Totalcow: Unleash the power of copy-on-write for thin-provisioned containers," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, 2015, pp. 1–7.
[8] T. Harter *et al.*, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 181–195.
[9] V. Rastogi *et al.*, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 476–486.
[10] H. Zhang *et al.*, "Machine learning containers are bloated and vulnerable," *arXiv preprint arXiv:2212.09437*, 2022.
[11] A. Gehani *et al.*, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Middleware*, 2012.
[12] Q. Pham *et al.*, "LDV: Light-weight database virtualization," in *ICDE'15*, 2015, pp. 1179–1190.
[13] M. Stamatogiannakis *et al.*, "Looking inside the black-box: capturing data provenance using dynamic instrumentation," in *IPAW*. Springer, 2014, pp. 155–167.
[14] L. Lin *et al.*, "Inde: An inline data deduplication approach via adaptive detection of valid container utilization," *ACM Trans. Storage*, nov 2022. [Online]. Available: doi.org/10.1145/3568426
[15] D. Zhang *et al.*, "Improving the performance of deduplication-based backup systems via container utilization based hot fingerprint entry distilling," *ACM Trans. Storage*, vol. 17, no. 4, oct 2021. [Online]. Available: doi.org/10.1145/3459626
[16] N. Zhao *et al.*, "DupHunter: Flexible High-Performance deduplication for docker registries," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 769–783. [Online]. Available: www.usenix.org/conference/atc20/presentation/zhao
[17] docs.h5py.org/en/stable/quick.html, HDF5 APIs.
[18] Hdf5 virtual file layer. [Online]. Available: bit.ly/3V6Iuxn
[19] Cloud storage options for hdf5. [Online]. Available: www.hdfgroup.org/2022/08/cloud-storage-options-for-hdf5/
[20] Pomd-pf aist notebook. [Online]. Available: bit.ly/3TrxwkO
[21] Krrigging application. [Online]. Available: bit.ly/43cEQE1
[22] Global percepitation notebook. [Online]. Available: bit.ly/3v3m6dt
[23] T. Li *et al.*, "h5bench: HDF5 I/O Kernel Suite for Exercising HPC I/O Patterns," in *Proceedings of Cray User Group Meeting, CUG 2021*, 2021.
[24] B. Dong *et al.*, "Terabyte-scale particle data analysis: an arrayudf case study," in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, 2019, pp. 202–205.
[25] R. Ye *et al.*, "Jslim: Reducing the known vulnerabilities of javascript application by debloating," in *Emerging Information Security and Applications*, W. Meng *et al.*, Eds. Cham: Springer International Publishing, 2022, pp. 128–143.
[26] A. Verma *et al.*, "Large-scale cluster management at google with borg," in *Tenth European Conference on Computer Systems*, 2015.
[27] C. Smowton, "I/o optimisation and elimination via partial evaluation," Ph.D. dissertation, University of Cambridge, 2014.
[28] Q. Xin *et al.*, "Studying and understanding the tradeoffs between generality and reduction in software debloating." in *The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2022*, 2022.
[29] Y. Jiang *et al.*, "Jred: Program customization and bloatware mitigation based on static analysis." in *Proceedings of the IEEE 40th Annual Computer Software and Applications Conference*, 2016.
[30] Y. JIANG *et al.*, "Reddroid: Android application redundancy customization based on static analysis." in *IEEE International Symposium on Software Reliability Engineering*, 2018.
[31] G. Malecha *et al.*, "Automated software winnowing," in *30th ACM Symposium on Applied Computing*, 2015.
[32] H. Sharif *et al.*, "Trimmer: Application specialization for code debloating," in *ASE*, 2018.
[33] R. K. Medicherla *et al.*, "Program specialization and verification using file format specifications," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 191–200.
[34] M. Stamatogiannakis *et al.*, "Trade-offs in automatic provenance capture," ser. IPAW 2016. Berlin, Heidelberg: Springer-Verlag, 2016.
[35] N. Balakrishnan *et al.*, "{OPUS}: A lightweight system for observational provenance in user space," in *5th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 13)*, 2013.
[36] P. Carns *et al.*, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, pp. 1–26, 2011.
[37] C. Xu *et al.*, "Dxt: Darshan extended tracing," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.
[38] J. Lofstead *et al.*, "Six degrees of scientific data: Reading patterns for extreme scale science io," in *Proceedings of the 20th international symposium on High performance distributed computing*, 2011, pp. 49–60.
[39] B. Dong *et al.*, "Real-time and post-hoc compression for data from distributed acoustic sensing," *Computers & Geosciences*, vol. 166, p. 105181, 2022.