SCALPEL: Structured Content Access Logging and Pruning for Efficient Layouts

Raffay Atiq*
SRI
Menlo Park, CA, USA
raffay.atiq@sri.com

Ashish Gehani
SRI
Menlo Park, CA, USA
ashish.gehani@sri.com

Tanu Malik
University of Missouri
Columbia, MO, USA
tanu@missouri.edu

Fareed Zaffar
Lahore University of Management Sciences
Lahore, Pakistan
fareed.zaffar@lums.edu.pk

***Abstract*—Containerizing scientific workflows helps ensure their reproducibility. Including all data required for deterministic re-execution aids the process. In data-intensive climate science and other high-performance computing domains, workflows routinely process large-scale data archives through parallel frameworks such as Dask. Packaging these archives inflates container images, with vast swaths of data that are never accessed by the application driving up transfer costs and hindering deployment. We present SCALPEL, a framework for *semantic carving* - selectively retaining only the data an application actually consumes during analysis, while excluding data that is accessed but not used.**

We provide two complementary carving modes, each operating at a different level of observability: (1) *value-level carving*, which traces data access at the resolution of individual rows (in tabular datasets) or specific multidimensional elements (in array datasets), enabling deterministic re-execution with precisely the same inputs; and (2) *partition/chunk-level carving*, a configurable alternative that tracks data access at the coarser granularity of table partitions or array chunks, facilitating flexible re-execution that can accommodate additional inputs within these broader segments. By interposing on HDF5 I/O operations and Dask's execution layer, both widely adopted technologies for large-scale data storage and parallel computation, we capture application-specific data access patterns, achieving reductions in container size by several orders of magnitude. We demonstrate these reductions in data-intensive scientific workflows, including precipitation-driven climate modeling and other geophysical workloads.

***Index Terms*—Containers, Carving, Big Data, Dask, HDF5, NetCDF4, Reproducibility**

I. INTRODUCTION

Reproducibility is a foundational requirement for credible scientific inquiry. However, achieving reproducilibity in modern data-intensive workflows becomes challenging due to the substantial size of input datasets, which significantly increases the overhead of preserving, transferring, and redeploying these workflows. Climate researchers and other high-performance-computing (HPC) practitioners increasingly rely on large-scale data archives stored in hierarchical, self-describing formats such as HDF5 [1] and netCDF4 [2]. Parallel analytics frameworks like Dask [3] orchestrate thousands of tasks across compute clusters to sift through these archives, but a typical task graph execution touches only a small subset of data present in each file. Despite this sparse access pattern, conventional container builds still package the entire data bundle to guarantee deterministic re-execution. Consequently, container images expand to impractical sizes, burdening storage, transfer, and deployment of the workflow. Addressing this mismatch between the data an application could access and the data it actually uses is essential for keeping scientific containers lean, portable, and easy to redeploy.

Consider a geophysical data archive stored in a single HDF5 file that contains five datasets: latitude, longitude, temperature, pressure, and relative humidity. An application that examines seasonal warming opens the file once and issues three read operations to obtain temperature together with its latitude and longitude coordinate grids. The remaining two datasets are never referenced. In addition, the downstream analysis may consume only the temperature measurements that coincide with a subset of latitude-longitude grid points, leaving other values in the three accessed datasets unused. When the workflow is containerized for reproducibility, the entire HDF5 archive, including pressure and relative humidity fields that are irrelevant to this analysis, are included as well as unused values from the latitude, longitude, and temperature datasets. These unused objects and values add unnecessary storage overhead to the container without contributing to the scientific result, thereby inflating storage, transfer, and deployment costs.

The same dataset can be ingested by a parallel data-processing framework like Dask that treats a collection of large binary files as a single virtual corpus, automatically dividing the content into *partitions* for tabular data [4] and multidimensional *chunks* for array data [5]. For example, for a table of ground-based weather-station measurements such as thousands of time-stamped observations of temperature, pressure, and relative humidity recorded on a geographic coordinate system, the framework might group one thousand rows into each partition. For a gridded temperature field, it could create 48-step temporal blocks over 1000×1000 spatial tiles. A task graph that computes the mean surface temperature over a bounded region – for example, latitude 0° to 45° N and longitude 0° to 90° W – touches only the table partitions whose rows fall within that coordinate window or only the array chunks whose tiles intersect the same geographic range, consulting solely the latitude, longitude, and temperature variables. Despite this narrowly focused access pattern, conventional container build tools still package complete input files. Consequently, the container inherits the full data volume rather than the subset actually processed, perpetuating unnecessary bloat.

Container bloat stems from three primary sources: extensive application code, large number of dependencies, and bulky data assets. Machine learning toolkits such as TensorFlow [6]

*While visiting SRI.

and Scikit-learn [7] illustrate the code and library side of the problem–they ship with many auxiliary packages so they can support a wide range of tasks out of the box. On the data side, self-describing formats like HDF5 [1] and netCDF4 [2] encourage bundling many datasets or variables in a single file. In practice, most workflows read only a small portion of these files, yet recent studies [8]–[11] show that even lightweight containerized applications routinely exceed a gigabyte, increasing storage and distribution costs, slowing transfers, and widening the attack surface [11].

Conventional mitigation techniques provide only partial relief. Deduplication only removes repeated data from a dataset and not unused data. Lineage-based systems [12]–[14] only focus on identifying what data files were accessed, and not what subsets of the data files were used. As a result, container bloat remains an open challenge for both practitioners and researchers.

Through interposing on library calls, we introduce specialized behavior to reduce container bloat. We consider two types of specialization - either by interposing at the low-level of I/O on disk or interposing at the high level of data processing at the application layer. The advantage of the former is that it delivers maximal precision for workloads executed with a fixed set of input parameters. The latter records accesses at the granularity of user-defined partitions or chunks. Because these units are scheduled across distributed workers, this layer enables carving to function in cluster environments. As an additional benefit, partition and chunk sizes can be tuned to retain a safety margin of data, supporting lightweight exploratory re-runs without requiring a full re-carve. Post-carving, the re-execution of the application is robust only to the subset of data carved, and additional data must be dynamically loaded.

We present SCALPEL[1], a framework that mitigates container bloat by transparently interposing in the software stack at two complementary layers. Low-level I/O interposition hooks the HDF5 read path, extracting not only the datasets read but also the precise values used in the analysis of the application. High-level data-processing interposition instruments frameworks such as Dask, capturing every dataframe partition or array chunk computed by workers across a cluster. In Dask, partition and chunk sizes dictate memory pressure per worker, network traffic, and attainable parallelism. By tuning them, users can match carving granularity to the resource profile of the distributed environment, maintaining efficient execution while still generating a compact carve.

The paper makes the following contributions:

- Designing a low level HDF5 library call interception method that identifies the precise values used by the application. We leverage two distinct selection methods used by the HDF5 library: *hyperslab* and *point* selection [15]. Each hyperslab or point selection is subsequently written to a carved file as an application performs a read call. Our scheme preserves the hierarchy of the original files as well as the metadata, key components that make HDF5 files self-describing. [Section III]
- Instrumenting Dask's dataframe and array calls lets us capture every partition and chunk that the task graph materialises. Because partition and chunk sizes are user-tunable, this layer offers controllable carving granularity: smaller partitions or chunks yield smaller carves, while larger partitions or chunks reduce tracing overhead and broaden reuse. The two interposition layers are complementary: value-precise HDF5 carving minimises the footprint for fixed inputs, whereas Dask-level carving scales across distributed workers and retains enough context for iterative, exploratory analyses. [Section IV]
- We evaluate both approaches on realistic workloads that process large-scale data archives from NASA, with diverse access patterns, ranging from fine-grained point queries to regional aggregates, demonstrating broad applicability across configurations. While the system supports hyperslab selection carving, our evaluation is based on point selection carving for a clear comparision with Dask-based partitions and chunks carving. [Section V and VI]
- We make our code available under a permissive open source license [16]–[18].

Our experiments show that SCALPEL reduces containerized datasets by up to 99% while retaining all data needed to guarantee deterministic re-execution for the supplied user inputs.

II. CONTAINER STORAGE DEBLOATING PROBLEM

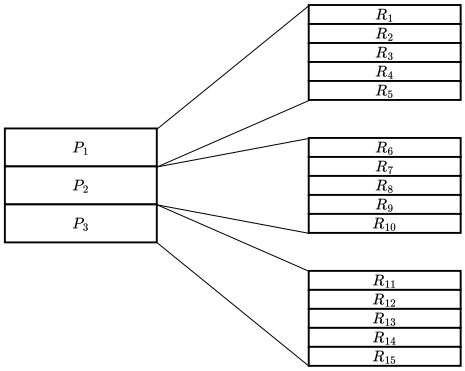
Consider an application that accesses a data file D using input parameters T. An execution of an application can be described as the result of the analysis performed by the application, such that for a particular set of input parameters, the application accesses subset $D' \subseteq D$. The goal of debloating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

We investigate two complementary approaches operating at different levels of granularity within the application stack. For each approach, we assess their implications for debloating semantics by considering data stored in both tabular and multidimensional array formats. The first approach targets fine-grained data access by precisely identifying the individual values required by the application.

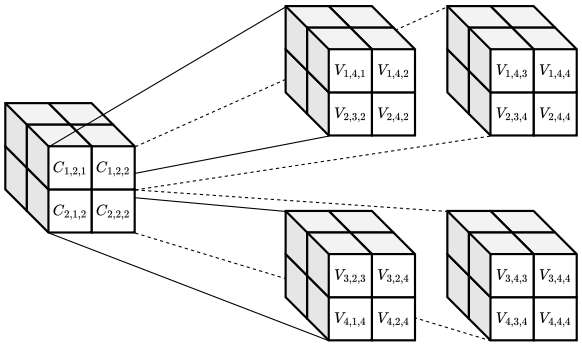
For tabular data, let R_i denote the i^{th} row accessed in the course of the downstream analysis performed by the application. Under this approach, after m row accesses have been performed, the resulting debloated subset of the data file is defined as $D'_R = \bigcup_{i=1}^{m} R_i$. Similarly, for array-based data with n dimensions let $V_{I^{(j)}}$ represent the j^{th} single scalar value accessed in the analysis of the application, located at indices $I^{(j)} = i_1^{(j)}, i_2^{(j)}, \ldots, i_n^{(j)}$ in an n-dimensional array. After m values have been accessed, the debloated subset of the data file in this case is given by $D'_V = \bigcup_{j=1}^{m} V_{I^{(j)}}$.

The second approach provides configurable observability by controlling the granularity at which data is processed and selectively extracted. For tabular data, we use partitions to represent groups of rows accessed by an application. Let P_i denote the set of partitions utilized by the application's analysis. After m' partition accesses, the debloated subset of the data file is then defined as $D'_P = \bigcup_{i=1}^{m'} P_i$. For array-based data, we utilize n-dimensional chunks to control data extraction along each dimension. The chunk dimensionality matches that of the array

[1] Structured Content Access Logging and Pruning for Efficient Layouts



(a) Partition-to-row mapping for a table with $r=15$ rows split into $p=3$ equal partitions ($\sigma=\lceil\frac{r}{p}\rceil=5$).



(b) Chunk-to-value mapping for a $4\times4\times4$ array divided into $2\times2\times2$ chunks ($\sigma_1=\sigma_2=\sigma_3=2$).

Fig. 1: Mapping of partitions to rows in tabular workflows and chunks to values in array-based workflows.

itself. Let $C_{I(j)}$ denote the chunks accessed by the application. After m' chunks have been accessed, the debloated subset of the data file in this case is defined as $D'_C=\bigcup_{j=1}^{m'}C_{I(j)}$.

Debloating data at different granularities leads to different container size reductions. Carving precise rows or values leads to reduction at the finest possible granularity, whereas carving partitions or chunks leads to reduction at a coarser granularity – i.e. $D'_R\subseteq D'_P$ for tabular data and $D'_V\subseteq D'_C$ for array-based data. We can model the relationship between these two carving approaches by defining a function f that maps a set of input parameters T to a subset of data stored within a file. Depending on the data representation, the resulting subset could be either tabular rows R or array values V:

$$f:T\rightarrow R \quad \text{(tabular data)}, \quad f:T\rightarrow V \quad \text{(array data)}$$

Let r be the number of rows, p be the number of partitions, and σ be the size of a partition, where $\sigma=\lceil\frac{r}{p}\rceil$. For the i^{th} input parameter T_i, $f'(T_i)$ maps partitions that may be accessed that encompass specific rows:

$$f'(T_i)=P_i=\bigcup_{j=\sigma(i-1)+1}^{\min(\sigma i,r)}R_j$$

Let σ_k be the size of the chunk for the k^{th} dimension d_k. For the i^{th} input parameter T_i, $f'(T_i)$ maps chunks that may be accessed that encompass specific values:

$$f'(T_i)=C_{i_1,i_2,\ldots,i_n}=\bigcup_{j_k=\sigma_k(i_k-1)+1}^{\min(\sigma_k i_k,d_k)}V_{j_1,j_2,\ldots,j_n}$$

Figures 1a and 1b provide an example illustrating these relationships for tabular and array data, respectively: the former maps partitions to their constituent rows, while the latter maps multi-dimensional chunks to the array elements they correspond to.

We leverage library interposition to operate at different levels in the application stack, as shown in Figure 2. Specifically, to achieve debloating at the granularity of the exact set of rows or values accessed, we interpose at the HDF5 library level, and to achieve debloating at the granularity of partitions and chunks, we interpose at the Dask library level.

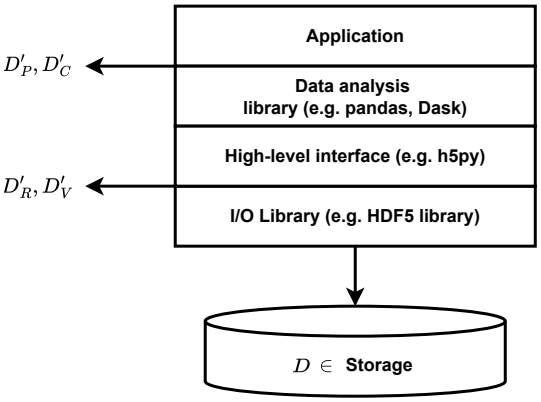


Fig. 2: Application stack.

Figure 3 illustrates a typical HDF5-based workflow, in which an application opens an HDF5 file, reads a target dataset, and then filters or queries portions of that data based on specified parameters. HDF5 supports two primary selection operations for extracting data subsets: *hyperslab selections*, which allow multidimensional subsetting (also known as slicing) of a dataset, and *point selections*, which pinpoint individual elements. Such workflows are often used for datasets that can fit within a single machine's memory and computations that can be executed locally. In these workflows, debloating at the granularity of precise rows or values is sufficient if the application is to be re-executed on the exact set of input parameters as the original execution and provides the highest possible container size reduction.

Figure 4 illustrates a typical Dask-based workflow designed to handle large-scale data or distributed computations. The process starts by opening a data file. Next, a dataset is read into memory. A Dask *DataFrame* or Dask *Array* is then constructed from the dataset and divided into a

```
def analysis(filename):
  file = file_open(filename)
  dataset = read_dataset(file)
  subset = query_data(dataset)
  return result(subset)
```

Fig. 3: HDF5 application source code.

specified number of partitions or chunks. In the case of Dask DataFrames, each partition is represented by an underlying Pandas DataFrame [4]. For Dask Arrays, each chunk is represented by an underlying NumPy *Array* [5]. With Dask's *lazy evaluation* model, all operations (such as queries) are staged into a task graph. When the query is finally triggered to be executed, Dask schedules these tasks across the available workers, processes partitions and chunks in parallel when feasible, and merges the partial results into a final output. This approach is used for workloads where the data size exceeds the memory capacity of a single machine or for computations that demand distributed execution over multiple nodes.

```
def analysis(filename):
  file = file_open(filename)
  dataset = read_dataset(file)
  dask_dataframe_or_array = create_dask_dataframe_or_array(
       dataset)
  subset = query_data(dask_dataframe_or_array)
  return result(subset)
```

Fig. 4: Dask application source code.

Although Parallel HDF5 supports parallel I/O across multiple processes [19], it requires explicit compilation instructions for parallel capabilities. By contrast, libraries like Dask are able to orchestrate high level computations in addition to disk I/O without requiring specialized parallel builds and are thus widely used. This motivates our approach to perform debloating at the granularity of Dask partitions or chunks, in addition to being able to control the granularity at which data is processed and thereby carved.

III. HDF5 Selections Carving

HDF5 provides selection mechanisms that allow applications to operate on precise subsets of dataset data, rather than entire arrays. By modifying these selection capabilities, we can monitor and "carve" out exactly the data that an application accesses, enabling fine-grained data extraction. An HDF5 selection specifies which elements of a dataset are involved in an I/O operation, forming the basis for tracing and isolating accessed data at a granular level. This section discusses the types of selections supported in HDF5 and how they enable fine-grained tracing of data access, which can be used to carve out accessed portions of data for aiding reproducibility.

A. Selection Mechanisms in HDF5

HDF5 supports flexible definitions of dataset subsets through its dataspace selection interface. HDF5 *dataspaces* define the layout of data elements of a dataset on disk and in memory [20]. Two primary selection types are provided [21]:

a) ***Hyperslab selection****:* A hyperslab is a selection of a logically contiguous block of points or a regular pattern of points or blocks within an n-dimensional dataspace [15]. A hyperslab is defined by an offset (starting coordinate), stride (step size), block size, and count (number of blocks) for each dimension of the dataspace. This allows selecting simple contiguous subarrays as well as periodic sampling (for example, every k-th element or block). NetCDF4, which uses HDF5 as an underlying storage layer [22], also utilizes hyperslab selection to extract dataset subsets. Hyperslabs can also be combined: the HDF5 API allows multiple hyperslab regions to be united or intersected, enabling representation of complex non-contiguous regions as a single selection [15].

b) ***Point selection****:* A point selection enumerates an arbitrary set of individual points within the dataspace [15]. Using point selection, an application can specify a list of discrete coordinates to include in the selection. This offers maximal flexibility, as any collection of points (even irregular or scattered positions) can be selected. Point selections are useful when the access pattern cannot be described as a simple stride or block. They directly identify specific data elements of interest.

High-level libraries build upon these HDF5 selection primitives to make them easier to use. For example, Python's *h5py* library exposes hyperslab selections through NumPy-like slicing syntax and point selections through fancy indexing [23]. By design, these mechanisms allow partial I/O at the granularity of structured blocks or individual data points, which facilitates accessing large datasets efficiently.

B. Fine-Grained Access Tracing and Data Carving

Because HDF5 selections precisely define which data points are accessed, they provide a foundation for fine-grained tracing of application I/O. Each time an application issues a read or write with a given selection, the HDF5 library knows exactly which elements of the dataset are involved. By monitoring these calls, one can record the access pattern in detail – down to specific indices or blocks within the dataset. For example, if an application reads a large 4D dataset but only needs a few scattered slices or points (say, a particular region of interest or sporadic timestamps), those accessed indices can be logged as a series of hyperslab and point selections. Over the entire run of the application, the union of all such selections constitutes the exact set of data that was actually used.

Leveraging this fine-grained trace, one can perform data carving – extracting just the accessed portions of datasets out of the original file to create a smaller, self-contained carved file. Upon file open, we first create an empty "skeleton" copy of the original HDF5 file's structure which includes all groups and datasets but with datasets initially empty, by augmenting the *H5Fopen* function [24]. After the desired selections are defined, the HDF5 library issues an *H5Dread* call to retrieve the chosen elements. Our interposition method intercepts each *H5Dread* call: it first forwards the call to the native library so the application receives the requested values. We then copy that same subset into the corresponding dataset in the carved file. Figure 5 illustrates how a hyperslab selection is carved from a two-dimensional dataset, whereas Figure 6 demonstrates the carving of a point selection from the same dataset. To preserve dataset

layout, we leave unaccessed regions in place and populate them with default fill values. Because those regions consist of long, identical byte sequences, applying per-dataset *gzip* compression collapses the repetition to a few back-references and code words, virtually eliminating the storage cost of the fill data while retaining full re-execution fidelity.

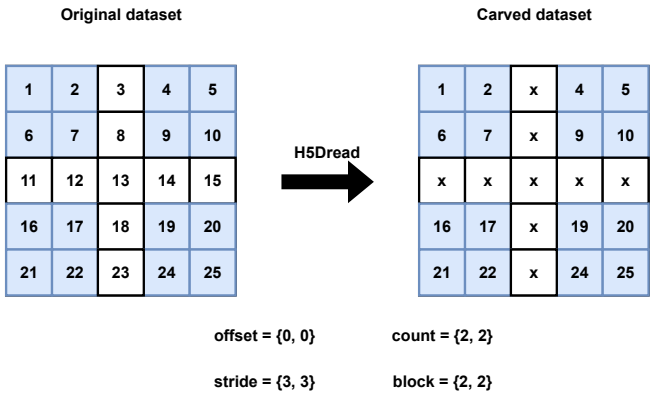


Fig. 5: Example of hyperslab selection carving.

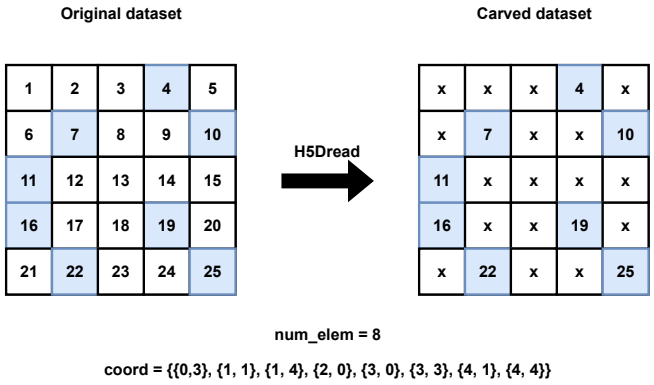


Fig. 6: Example of point selection carving.

When the application terminates, we intercept the *H5_term_library* function to copy any remaining metadata, preserving the file's self-describing nature. The outcome is a carved HDF5 file that contains only the subset of data the application actually read, achieving reduction at the sub-dataset granularity.

IV. Dask Partitions and Chunks Carving

Modern data-analytics pipelines often rely on Dask to scale Python-based workflows from a laptop to clusters with minimal change to application code. Dask achieves this by dividing tabular data into partitions and multidimensional arrays into chunks, then orchestrating fine-grained tasks over those pieces through a scheduler. Users can tune partition and chunk sizes – large blocks reduce scheduling overhead, while smaller blocks expose finer structure – thereby giving direct control over the granularity at which both computation and any subsequent carving can operate. By monitoring the exact partitions and chunks materialized after a task graph execution, we construct a minimal, self-contained subset that reproduces the original results while discarding unused portions of the dataset.

A. Partition-Based Carving for Tabular Workflows

Partition-based carving leverages Dask's capability to decompose a DataFrame into n user-defined partitions. Each partition is a Pandas DataFrame in the underlying representation. By tracing which of those sub-dataframes are actually materialised in a given task graph execution - and which columns within them are accessed - we can reconstruct a minimal carved replica that is functionally equivalent for that workload, as shown in Figure 7.

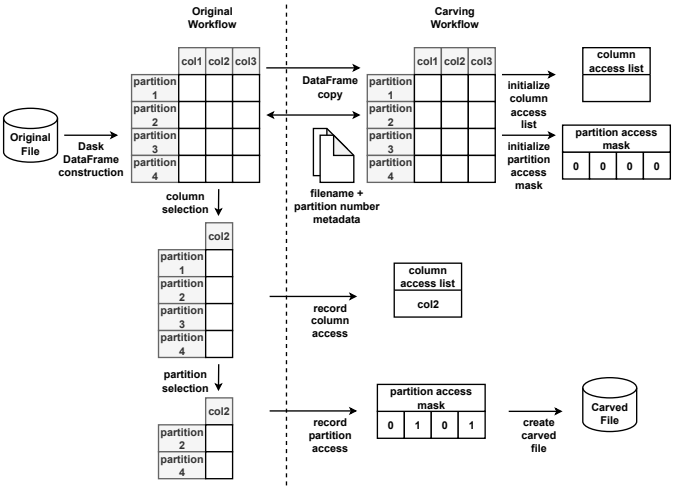


Fig. 7: Workflow illustrating Dask partition carving: partitions and columns accessed during execution are recorded to create a reduced carved file.

At Dask DataFrame construction time, SCALPEL makes a global copy of the DataFrame, and for both the original and the duplicate, each Dask partition is annotated with two lightweight metadata fields – file name and partition number – to identify each partition's provenance. The copy of the DataFrame is stored per file in a global dictionary; this isolation guarantees that normal application logic proceeds unchanged. The global dictionary allows sharing of carving information across workers in a cluster environment. The same dictionary holds (i) a *partition-access mask*, a Boolean array whose length equals the number of partitions, and (ii) a *column-access list* that records the first occurrence of every column reference.

During execution we interpose on two types of library functions:

- *Column-selection operators* are over-ridden to append newly referenced columns to the access list.
- *Dask's task graph execution functions* are wrapped to capture the materialised result, reading the *partition_number* field, and setting the corresponding bits in the partition-access mask.

At program shutdown SCALPEL operates file by file and modifies the global copy of each Dask DataFrame to reflect the original workflow's computed results:

- **Column pruning** removes those not in the access list.
- **Row pruning** replaces those in unaccessed partitions with default fill values.
- **Materialisation** computes the pruned global copy once, with the partitions and columns that remain written into a pre-created HDF5 skeleton. Unread partitions remain as gzip-compressed blocks of fill values, adding almost no overhead to the final file.

The resulting carved file preserves the full schema of the original yet contains only the data actually touched by the workflow, achieving reduction at sub-partition granularity.

B. Chunk-Based Carving for Dask Arrays

Dask arrays are divided into chunks based on a user-specified chunk shape. In the internal representation, each chunk is an independent NumPy array. Chunk-based carving leverages this division to isolate precisely those chunks that materialize from a computation and eliminate the rest from a given dataset.

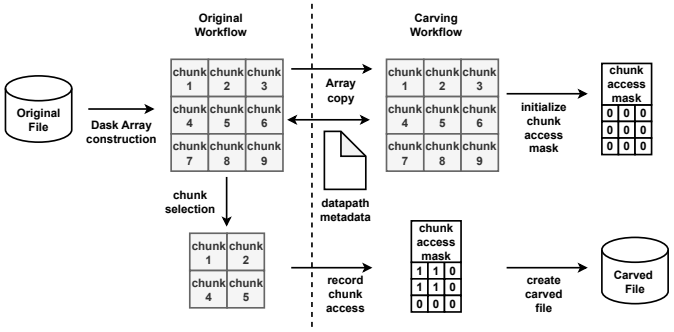


Fig. 8: Workflow illustrating Dask array carving: chunks accessed during execution are recorded to create a reduced carved file.

When a Dask Array is constructed, SCALPEL makes a global copy of the array and annotates both the original and the copy with metadata specifying the data path of the array, which includes the file the array comes from as well as the internal dataset name. SCALPEL then allocates a Boolean chunk-access mask whose shape mirrors the chunk grid that the array was divided into. Each entry – initially false – corresponds to a single chunk. Both the copy and its access mask are placed in a global dictionary whose key is the array's unique data path, so every worker in a cluster can retrieve the correct carving state with a single lookup. As the task graph is executed, the instrumentation sets a mask bit to true whenever the associated chunk is materialized. At program completion, SCALPEL walks through every tracked file: for each array, chunks marked as accessed are kept intact, while those never accessed are replaced with a default fill value. The storage overhead from these fill values is minimized via gzip compression. The carved data is then written into the appropriate datasets of a pre-generated HDF5 skeleton, producing a carved file that preserves the original structure while containing only the data actually used by the workflow, achieving reduction at sub-chunk granularity, as shown in Figure 8.

V. IMPLEMENTATION

The SCALPEL system is developed in C and Python, with the C language used to implement HDF5 selections carving and Python used to implement Dask partition and chunk-based carving. The LD_PRELOAD mechanism is used to intercept HDF5 library calls. For HDF5 selections carving, the mechanism is used to create skeleton files and populate them with data accessed in a workflow. For Dask carving, the LD_PRELOAD mechanism is used to create skeleton files.

Below we describe the experiments that we conducted to assess the ability of SCALPEL to reduce container bloat in scientific computing settings. The applications were executed on Ubuntu 22.04, running on an AMD EPYC 7451 24-core 1.2 GHz processor with 1.5 TB RAM.

Name	File Format	Total File Size
POMD-PF South America Storms [25]	HDF5	934 MB
IMERG View [26]	netCDF4	36 GB
SpatioTemporal Feature Database [27]	HDF5	18 GB

TABLE I: Scientific applications.

VI. EVALUATION

Applications: We evaluate the effectiveness and performance trade-offs associated with data carving in three representative scientific applications: *POMD-PF South America Storms*, *IMERG View*, and *SpatioTemporal Feature Database*. Table I lists these applications and provides references to their respective GitHub repositories. All three applications are implemented in Python by climate scientists. The *POMD-PF South America Storms* application ingests tabular data, performing threshold-based and spatial grid subsetting for its analysis. In contrast, both *IMERG View* and *SpatioTemporal Feature Database* are array-based workflows applying threshold subsetting operations on precipitation data, with dataset shapes of (1, 1800, 3600) and (1, 3600, 1800), respectively.

Data Format: Programs involved in the experiments utilized HDF5 and netCDF4 data files. However, netCDF4 libraries do not utilize HDF5 point selection primitives. Therefore, we modified applications that use these libraries to instead use HDF5 libraries directly. Our tool comprehensively carves HDF5 files, including mirroring the filesystem hierarchy and preserving metadata. Since the features of netCDF4 are a subset of the features provided by HDF5 [22], the resulting carved files remain valid netCDF4 files.

Carving Tests: To test carving on tabular workflows (*POMD-PF South America Storms*), we measured running time as a function of the granularity of the partitions. To do this, we divided the data into an exponentially increasing number of partitions. When successive divisions resulted in an order of magnitude increase in the running time, we stopped our analysis. In HDF5, data subsets are selected through the conjunction of application-specific queries, subsequently converting these subsets into dataframes for analysis. For Dask, datasets were loaded into Dask Dataframes, applying analogous query subsets.

To test carving on array-based workflows (*IMERG View* and *SpatioTemporal Feature Database*), we measured running time as a function of chunk shape. To do this, we repeatedly reduced the chunk shape by roughly a factor of four along

every dimension except the first (since it already had the smallest possible size of 1). When successive divisions resulted in an order of magnitude increase in the running time, we stopped our analysis. To minimize I/O overhead in these tests, datasets were re-chunked prior to each execution using the *h5repack* tool, matching the chunk shapes to those defined for the Dask arrays.

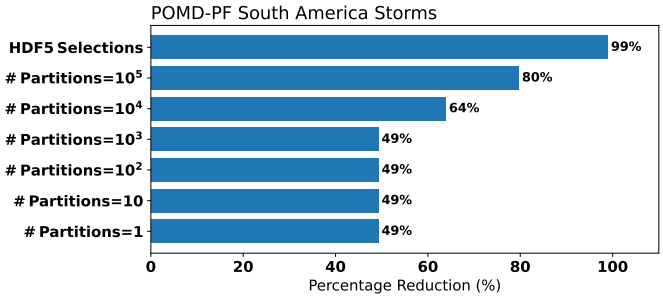
Metrics: The evaluation quantifies both the percentage of data reduction achieved and the associated runtime impacts when applying HDF5 selections carving and Dask carving at varying granularities. These metrics specifically emphasize the interplay between partitioning or chunking granularity, data reduction efficiency, and the scheduling overhead incurred.

Results: Figure 9 shows the percentage reduction in container size achieved using HDF5 selections carving and Dask carving, with respect to the number of partitions into which a tabular dataset was divided and the chunk shape specified for an array-based workflow. Apart from data, the containers include only the application code and minimal runtime dependencies whose footprint is orders of magnitude smaller than the original datasets. Because HDF5 selections carving precisely captures only those elements directly accessed by the workflow, it consistently achieves the highest data reduction. In Dask carving, the percentage reduction depends on the granularity of partitioning or chunking. Coarser layouts (with fewer, larger partitions or chunks) result in lower data reductions, while finer layouts (with more numerous, smaller partitions or chunks) yield higher reductions.

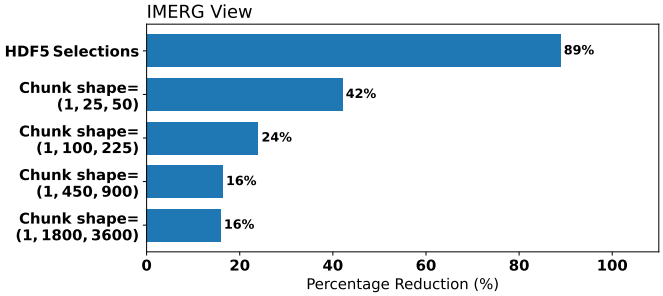
Figure 10 decomposes runtime into three distinct phases: (1) the original execution reading the complete dataset, (2) the carving execution producing the reduced dataset, and (3) a re-execution of the analysis over the carved datasets. In all cases, carving execution runtimes are higher than original execution runtimes. In most cases, the re-execution runtimes are comparable to the original execution runtimes.

Discussion: The observed differences in data reduction and runtime are directly attributable to the granularity-dependent trade-offs between data reduction efficiency and associated scheduling overhead. While coarser Dask carving layouts produce fewer tasks and thus minimize task-scheduling overhead, they include substantial amounts of unaccessed data, leading to less effective reductions. Conversely, finer Dask layouts approximate the selectivity of HDF5 selections carving and thus achieve greater reductions, but the accompanying increase in task count significantly raises scheduling overhead, negatively affecting overall runtime.

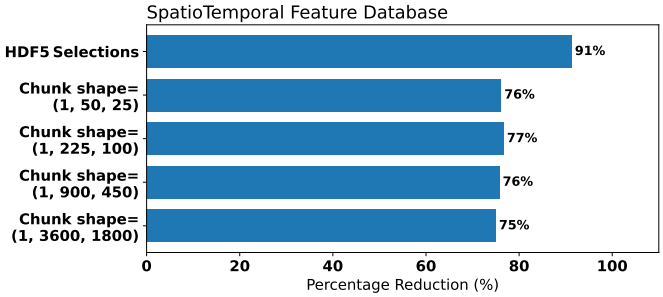
During carving execution, HDF5 selections carving adds only a small constant overhead due to direct element copying within the same library call, whereas Dask carving overhead increases with finer granularity because of elevated scheduler load from a higher number of tasks. The re-execution phase demonstrates the effectiveness of data carving, as the running time remains comparable to that of the original execution, as would be expected given the functional equivalence of the carved datasets.



(a) POMD-PF South America Storms reduction percentage.



(b) IMERG View reduction percentage.



(c) SpatioTemporal Feature Database reduction percentage.

Fig. 9: Percentage reduction with HDF5 selections carving and partition/chunk based Dask carving.

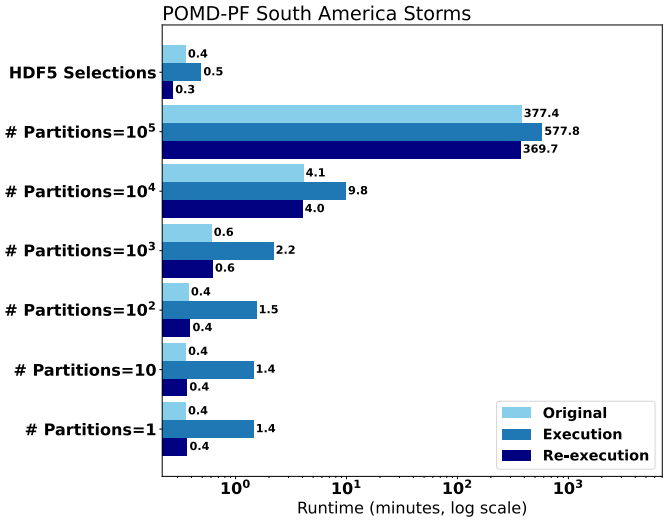
VII. Related Work

Several threads of research relate to data debloating:

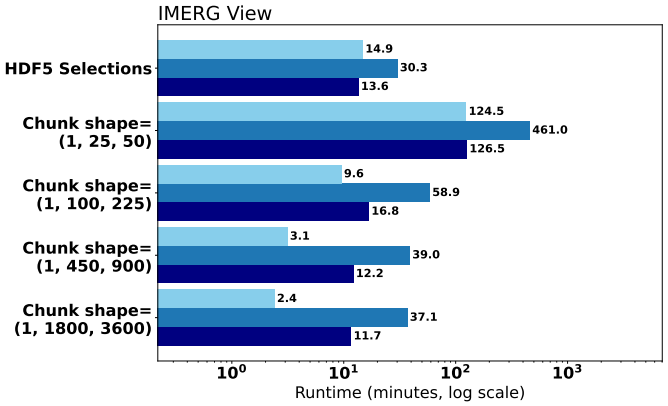
Code Debloating: Study of code bloat in software with consequent debloating has received significant attention in the recent past. For example, OCCAM [28], [29] and Trimmer [30]–[32] demonstrated that configuration based debloating can be applied to modern applications. Unlike these code-based debloating techniques, SCALPEL explores data-based debloating for efficient storage space utilization and application reproducibility.

Program Specialization: Work on program specialization by Medicherla *et al.* [33] checked that a program conformed to a specified format and transform code to use a restricted file format. In contrast, SCALPEL maintains the format but eliminates data that is not needed for specific executions.

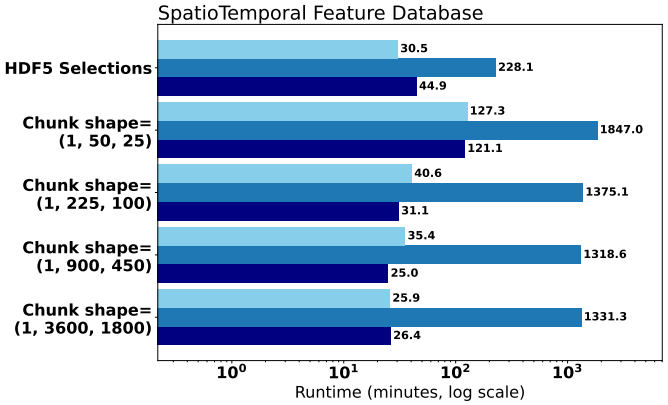
Container Debloating: Redundant downloading of data within each layer is a known issue in container-based deployments, resulting in large size images [34]. Consequently, several efforts have studied container debloating [35]. Notably, Slacker [9] used deduplication of file blocks to create more efficient containers. However,



(a) POMD-PF South America Storms runtimes.



(b) IMERG View runtimes.



(c) SpatioTemporal Feature Database runtimes.

Fig. 10: End-to-end runtime for each carving strategy for original application execution, carving execution, and re-execution on carved files.

block-level deduplication techniques do not eliminate data redundancies within structured arrays. SCALPEL is designed to remove unused data in formats such as HDF5 [1].

Data Lifting: LLIO [36] improves the runtime performance of applications by specializing I/O calls using file contents. MiDas [37] and IOSPReD [38] also lift accessed data into the application binary, allowing the remaining file to be discarded. However, these approaches are untenable for big data. In contrast, SCALPEL retains the data in files.

I/O Specialization: ABCD [24] uses system-call-level interposition to support re-execution with identical inputs, and HDF5-library-level interposition allow applications to vary inputs values (but not the parameters used). Kondo [39] aims to identify the data subset needed for all possible executions using input fuzzing. In contrast, SCALPEL operates at the HDF5 selections level, capturing exact data points accessed for executions with specific parameters, and specializes Dask library calls to enable flexible exploratory re-executions with user-tunable additional parameters.

I/O Monitoring: Darshan [40] is an I/O characterization tool used to profile and analyze I/O behavior in high-performance computing (HPC) applications. Darshan's DXT module [41] can also instrument I/O calls in the application code to collect data on file operations. Determining how Darshan can interact with the carving and re-execution remains as future work.

VIII. CONCLUSION

Efficient reproducibility demands that scientific containers carry only the data an analysis actually needs. We introduced a carving framework that achieves this goal through transparent interposition at both the storage and execution layers.

At the storage layer, by interposing on the HDF5 selection interface, we detect each hyperslab or point selection read in real time and produce a carved sub-dataset that contains only the referenced values. At the execution layer, interposition on Dask's task graph execution reveals exactly which partitions and chunks are materialised during computation, allowing users to tune carving granularity and balance size against overhead.

These findings demonstrate that careful interposition at both the I/O and application execution layers can turn existing provenance signals into actionable data minimisation. Future work will explore support for reducing scheduling overhead for Dask carving, fuzzing-based approaches, and use of machine learning techniques. We believe the principles laid out here pave a practical path toward lean, transparent, and fully reproducible science at scale.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Aeronautics and Space Administration (NASA) under Grant AIST-21-0095. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NASA.

REFERENCES

[1] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "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.

[2] R. Rew, E. Hartnett, J. Caron *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.

[3] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," 01 2015, pp. 126–132.

[4] Dask DataFrame. [Online]. Available: https://docs.dask.org/en/stable/dataframe.html

[5] Dask Array. [Online]. Available: https://docs.dask.org/en/stable/array.html

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: www.tensorflow.org

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] X. Wu, W. Wang, and S. Jiang, "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.

[9] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016.

[10] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 476–486.

[11] H. Zhang, F. A. Ahmed, D. Fatih, A. Kitessa, M. Alhanahnah, P. Leitner, and A. Ali-Eldin, "Machine learning containers are bloated and vulnerable," *arXiv preprint arXiv:2212.09437*, 2022.

[12] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," *13th ACM/IFIP/USENIX International Middleware Conference*, 2012. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/MW-2012.SPADE.pdf

[13] Q. Pham, T. Malik, and et. al., "LDV: Light-weight database virtualization," in *ICDE'15*, 2015, pp. 1179–1190.

[14] M. Stamatogiannakis, P. Groth, and H. Bos, "Looking inside the black-box: capturing data provenance using dynamic instrumentation," in *IPAW*. Springer, 2014, pp. 155–167.

[15] Dataspaces and Partial I/O. [Online]. Available: https://support.hdfgroup.org/documentation/hdf5/latest/_h5_s__u_g.html

[16] HDF5 Selections Carving. [Online]. Available: https://github.com/data-carving/hdf5-selections-carving

[17] Dask Carving. [Online]. Available: https://github.com/data-carving/dask-carving

[18] Semantic Carving Dockerfiles. [Online]. Available: https://github.com/data-carving/semantic-carving-dockerfiles

[19] Parallel HDF5. [Online]. Available: https://docs.h5py.org/en/stable/mpi.html

[20] HDF5 Dataspaces. [Online]. Available: https://support.hdfgroup.org/documentation/hdf5/latest/group___h5_s.html

[21] Reading From or Writing To a Subset of a Dataset. [Online]. Available: https://support.hdfgroup.org/documentation/hdf5/latest/_l_b_dset_sub_r_w.html

[22] NetCDF4 File Format Specifications. [Online]. Available: https://docs.unidata.ucar.edu/netcdf-c/current/file_format_specifications.html

[23] h5py Datasets. [Online]. Available: https://docs.h5py.org/en/stable/high/dataset.html

[24] R. Tikmany, A. Modi, R. Atiq, M. Reyad, A. Gehani, and T. Malik, "Access-Based Carving of Data for Efficient Reproducibility of Containers," *24th ACM/IEEE Symposium on Cluster, Cloud, and Internet Computing*, 2024. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/CCIC-2024.ABCD.pdf

[25] POMD-PF South Africa Storms. [Online]. Available: https://bit.ly/4cLrRxN

[26] IMERG View. [Online]. Available: https://bit.ly/42X97Ip

[27] Feature Database. [Online]. Available: https://bit.ly/41h1BI6

[28] G. Malecha, A. Gehani, and N. Shankar, "Automated Software Winnowing," *30th ACM Symposium on Applied Computing (SAC)*, 2015. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/SAC-2015.Winnow.pdf

[29] J. Navas and A. Gehani, "OCCAMv2: Combining Static and Dynamic Analysis for Effective and Efficient Whole Program Specialization," *Communications of the ACM*, vol. 66(4), 2023. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/CACM-2023.OCCAMv2.pdf

[30] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: Application Specialization for Code Debloating," *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/ASE-2018.Trimmer.pdf

[31] A. Ahmad, R. Noor, H. Sharif, U. Hameed, S. Asif, M. Anwar, A. Gehani, F. Zaffar, and J. Siddiqui, "Trimmer: An Automated System For Configuration-Based Software Debloating," *IEEE Transactions on Software Engineering (TSE)*, vol. 48(9), 2022. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/TSE-2022.Trimmer.pdf

[32] A. Ahmad, M. Anwar, H. Sharif, A. Gehani, and F. Zaffar, "Trimmer: Context-Specific Code Reduction," *37th IEEE/ACM Conference on Automated Software Engineering (ASE)*, 2022. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/ASE-2022.Trimmer.pdf

[33] R. K. Medicherla, R. Komondoor, and S. Narendran, "Program specialization and verification using file format specifications," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 191–200.

[34] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Tenth European Conference on Computer Systems*, 2015.

[35] M. Hassan, T. Tahir, M. Farrukh, A. Naveed, A. Naeem, F. Shaon, F. Zaffar, A. Gehani, and S. Rahaman, "Evaluating Container Debloaters," *8th IEEE Secure Development Conference (SecDev)*, 2023. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/SecDev-2023.Debloat.pdf

[36] C. Smowton, "I/O optimisation and elimination via partial evaluation," Ph.D. dissertation, University of Cambridge, 2014.

[37] C. Niddodi, A. Gehani, T. Malik, J. Navas, and S. Mohan, "MiDas: Containerizing Data-Intensive Applications with I/O Specialization," *3rd ACM Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS)*, 2020. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/PRECS-2020.MiDas.pdf

[38] C. Niddodi, A. Gehani, T. Malik, S. Mohan, and M. Rilee, "IOSPReD: I/O Specialized Packaging of Reduced Datasets and Data-Intensive Applications for Efficient Reproducibility," *Access*, vol. 11, 2023. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/Access-2023.IOSPReD.pdf

[39] A. Modi, R. Tikmany, T. Malik, R. Komondoor, A. Gehani, and D. D'Souza, "Kondo: Efficient Provenance-driven Data Debloating," *40th IEEE Conference on Data Engineering (ICDE)*, 2024. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/ICDE-2024.Kondo.pdf

[40] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, pp. 1–26, 2011.

[41] C. Xu, S. Snyder, V. Venkatesan, P. Carns, O. Kulkarni, S. Byna, R. Sisneros, and K. Chadalavada, "Dxt: Darshan extended tracing," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.