H5Intent: Autotuning HDF5 with User Intent

Hariharan Devarajan, *Member, IEEE,* Gerd Heber, *Member, IEEE,* Kathryn Mohror, *Member, IEEE*

Abstract—The complexity of data management in HPC systems stems from the diversity in I/O behavior exhibited by new workloads, multistage workflows, and multitiered storage systems. The HDF5 library is a popular interface to interact with storage systems in HPC workloads. The library manages the complexity of diverse I/O behaviors by providing user-level configurations to optimize the I/O for HPC workloads. The HDF5 library exposes hundreds of configuration properties that can be set to alter how HDF5 manages I/O requests for better performance. However, determining which properties to set is quite challenging for users who lack expertise in HDF5 library internals. We propose a paradigm change through our H5Intent software, where users specify the intent of I/O operations and the software can set various HDF5 properties automatically to optimize the I/O behavior. This work demonstrates several use cases where mapping user-defined intents to HDF5 properties can be exploited to optimize I/O. In this study, we make three observations. Firstly, I/O intents can accurately define HDF5 properties while managing conflicts between various properties and improving the I/O performance of microbenchmarks by up to 22×. Secondly, I/O intents can be efficiently passed to HDF5 with a small footprint of 6.74 MB per node for thousands of intents per process. Thirdly, an H5Intent VOL connector can dynamically map I/O intents to HDF5 properties for various I/O behaviors exhibited by our microbenchmark and improve I/O performance by up to 8.8×. Overall, we found that the H5Intent software improves the I/O performance of complex large-scale workloads we studied by up to 11×.

Index Terms—I/O intents, HDF5, autotuning, HDF5 property lists, configurations

1 Introduction

Modern HPC workloads that produce and consume large amounts of data [1] are known as data-intensive workloads. These workloads include applications that run in isolation [2], such as simulation, analysis, and artificial intelligence (AI) applications, or include multiple applications that cooperatively execute in a workflow [3]. The diversification in workloads has changed typical I/O patterns away from being dominated by write-heavy, collective, sequential patterns in bulk-synchronous simulations to now include random write-once-read-many (WORM) patterns in big data, read-heavy patterns in AI, and consumer/producer patterns in multi-stage workflows [4]. To manage the diversity of these workloads, storage system designers and I/O library developers have provided configuration parameters [5]–[7]. The HDF5 I/O library is widely used in the scientific HPC community [8] in many new and existing workloads, such as VPIC simulations [9], BDCATS data analysis [10], and Cosmoflow physics AI [11]. Like other storage system software, the HDF5 library provides an extensive collection of configurations (through so-called property lists) that can be used to fine-tune the library's behavior and to optimize applications' I/O performance. Some examples of optimizations provided by configurations are data filtering, compression, and caching. Setting workload-specific configurations for optimizing I/O in the HDF5 library can yield significant performance benefits for the workloads [5].

However, most application developers do not take full advantage of the performance optimizations available in the HDF5 library for three main reasons. Firstly, many application developers are domain scientists with only the knowledge of "how" they interact with storage systems. In the case of HDF5, the "how" includes high level information such as the dimensions of the data, type of data stored, and granularity of writes to the dataset [12]. However, the high-level information provided by the application is transformed into multiple lower-level I/O operations by the HDF5 library, obscuring which HDF5 configurations may be useful to the application. Secondly, even if application developers are educated about HDF5 configurations, there are hundreds of configurations to learn. Also, understanding which configuration works well for a given use case is rather complex and often involves benchmarking to perform expensive parameter tuning of HDF5 properties [13]. Finally, HDF5 configurations have a performance impact on each other, e.g., where setting one configuration variable can negate the performance benefits of another configuration variable. Incorrect configurations could lead to up to 6× slowdowns for the workloads (Section 2.4). Therefore, we need an intelligent and extensible way to translate users' knowledge of I/O behaviors to HDF5 configurations.

In prior work, we proposed a new paradigm where users pass their I/O intents to the underlying storage software, which then automatically tunes itself [14]. We implemented the new paradigm in the Mimir infrastructure, where I/O intents can be passed by the users and applied at different levels of the software stack with a focus on the POSIX layer. However, as every layer of the software stack is different, this mapping between the user's I/O intent and configurations must be explored for each layer. In general, user intent is defined as "why", "what", and "how" users perform certain I/O operations. Every I/O operation on a storage interface is accompanied by a "Why the user wants to do the I/O", "On what entity the user wants to do the I/O", and "How the users perform the I/O".

In this work, we extend the ideas of Mimir to HDF5, describe the I/O intents for the HDF5 library, and the mapping of I/O intents to HDF5 configurations. I/O intents

- *Hariharan Devarajan and Kathryn Mohror are with Lawrence Livermore National Laboratory. E-mail: {hariharandev1, mohror1}@llnl.gov.*
- *Gerd Heber is with The HDF Group. E-mail: gheber@hdfgroup.org.*

Manuscript received Month Day, Year; revised Month Day, Year.

on high-level I/O libraries can describe high-level use-cases such as chunking of datasets, complex subsetting of data, and compression. The goal of this work is to explore intents in this high-level I/O interface. An example of an intent for the HDF5 interface is "a simulation that performs a checkpointing operation over HDF5 files using a transfer size of 4KB". Here, "checkpointing" is the "why", "HDF5 files" is the "what", and "a transfer size of 4KB" is the "how". HDF5 provides over 200 configurations to users, so this work greatly reduces the complexity of tuning the performance of the HDF5 library.

We developed the H5Intent software suite that facilitates the automatic conversion of HDF5 I/O intents into configurations to dynamically optimize I/O for HPC workloads. Specifically, the suite consists of a) *an extractor* to automatically extract I/O intents from HDF5 applications, b) *a virtual object layer (VOL)* connector to dynamically attach to HDF5 applications at runtime, and c) *an intent API adaptor* for new applications to pass I/O intents to the HDF5 library. To achieve the goals of H5Intent, we investigated a variety of I/O intents that apply to HDF5 applications. Then, we explored the mapping of these HDF5 I/O intents to various HDF5 configurations as well as the interplay that occurs when setting multiple concurrent HDF5 configurations. Finally, we developed the H5Intent suite to make this process automatic and extensible. The main contributions of this work are:

1) *A definition of I/O intents* for the HDF5 library;
2) *A mapping I/O intents* to HDF5 configurations using HDF5 best practices and heuristics; and
3) *Design and Implementation of the H5Intent suite* for extracting I/O intents and autotuning HDF5 configurations;

2 Background and Motivation

2.1 User Intents for I/O

HPC storage-system software utilizes POSIX as its default I/O interface for workloads [15]. For instance, parallel file systems such as GPFS [16] and Lustre [17], node-local burst-buffer libraries such as UnifyFS [18] and Hermes [6], and shared burst-buffer technologies such as Datawarp [19] and IME [20] expose the POSIX interface as their primary mode of interaction with the storage system. The POSIX interface is defined within the Linux kernel for manipulating files and their properties with data APIs (e.g., `read` and `write`), metadata APIs (e.g., `fstat`, `chown`, and `open/close`), and intent APIs using `fadvice`. The `fadvice` API supports a few common OS file-system-specific user intents [21] such as sequential or data access. These intents are used within the kernel to manage OS cache efficiently [22].

However, `fadvice` was designed for processes running on the same system accessing a local file system and thus is insufficient in the context of distributed workloads running on large-scale storage systems. Therefore, we proposed a new set of intent APIs, called Mimir API [14], to systematically extend the existing I/O interfaces and build an intent-sharing infrastructure for HPC storage systems. The work demonstrated the benefit of using I/O intents at the POSIX layer and identified the need to investigate I/O Intents for other software stack layers. Thus, we extend the Mimir work to the HDF5 layer and identify an exhaustive set of I/O intents that are applied at the HDF5 I/O layer with tools and libraries to boost scientists' productivity.

2.2 Diversity of HDF5 workloads

The HDF5 data model is designed as an object-based model with core primitives such as files, groups, datasets, datatypes, and attributes. These primitives can be used to build complex scientific data representations using a set of high-level I/O interfaces aimed at managing data within a file. Additionally, the HDF5 library consists of configurations called property lists that tune the library's I/O behavior. The HDF5 library is used as an I/O interface in many workloads in the HPC ecosystem. This paper showcases exemplary workloads from large-scale simulation, high-performance data analytics, and AI.

VPIC [9] is a general-purpose particle-in-cell simulation code for modeling kinetic plasmas in one, two, or three spatial dimensions. VPIC has a native I/O format, built over the HDF5 library, that interfaces with the high-performance visualization software. This simulation application performs large collective I/O using the H5Part library using the MPI-IO driver within HDF5. Bd-Cats [10] is an HDF5 application in the cosmology and plasma physics domain that can perform end-to-end analysis at a trillion particle scale. Regarding data loading, the application reads data produced by a simulation application over eight variables with a data size in the terabyte scale. The CosmoFlow AI workload [11] estimates the values of critical cosmological parameters from a 3D cosmological simulation using deep learning. The CosmoFlow dataset contains 10K simulated universes with four redshifts (channels) and 5123 voxels, stored as 16-bit integers, with the four cosmological parameters used by the workload to generate the universe. The 1.5TB dataset contains 50K samples of 32MB each, stored as an HDF5 file.

These workloads demonstrate the diversity of I/O characteristics among HPC workloads using HDF5. This makes the I/O characteristics and data exchange between these workloads even harder to manage by monolithic storage software such as HDF5.

2.3 Configurability of HDF5 library

Property lists are the main HDF5 feature available for configuring the behavior of HDF5 API functions [23]. The configurations provided through property lists are the backbone of satisfying diverse I/O requirements from modern HPC workloads. Property Lists are attached to different API classes with HDF5, such as files, datasets, links, and groups. Additionally, each class of HDF5 object can be used to modify the behavior of opening or creating HDF5 objects or when reading or writing data. Property lists use a session scope with explicit open/create or close calls from the property lists APIs. Some examples of these property list configurations are: users can reduce the size of files on the storage system by enabling compression filters and metadata compactions; users can perform data caching by enabling chunk cache or utilize in-memory datasets for small datasets; users can align I/O with the parallel file system by allowing dataset chunking (that splits the data into multiple pieces), specifying storage layouts, and applying data-sieving optimizations (which reduces random I/O by reading more data); finally, users can increase the reliability

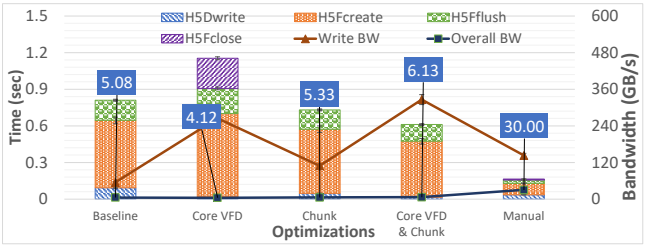


Fig. 1. Applying CoreVFD and chunking optimization for small accesses improves the application's performance, but using intent-driven configuration achieves 4.9× over naively applying multiple optimizations.

of writing data by enabling error detection and checksum operations. In this work, we utilize various property lists which have the most impact on the I/O performance of the applications. These include general properties [23] that enable aligned I/O, file/dataset chunking, caching or in-memory operations, and control file locking. Additionally, there are a few HDF5 properties that are only applicable for file-per-process patterns [24]. For example, the *core virtual file driver (VFD)* opens an HDF5 file in memory and flushes it during close, and the *page buffer* enables buffered I/O through a generic page cache optimization. Finally, there are some HDF5 properties only applicable for collective I/O [25] through the MPI-IO library [26]. For example, within HDF5 users can configure the number of processes participating in collective I/O on a common dataset through multi/one linked datasets, and they can perform collective I/O on a chunked dataset (a dataset split into multiple pieces) using the collective chunk I/O property list.

2.4 Motivation

To understand the impact of diverse and dynamic I/O behaviors on I/O performance for workloads using the HDF5 library, we perform a benchmark test with different I/O behaviors and HDF5 configurations and measure its performance on the Lassen machine [27] from Lawrence Livermore National Laboratory. To motivate our work, we present a use case where we run a write-only workload with a transfer size of 4KB writes with a file-per-process pattern (Figure 1). We run this workload on four nodes with HDF5 configurations: Core-Virtual File Driver [23], 1MB dataset chunking [23], and both. In these tests, we observe that different HDF5 configurations combinations result in different I/O performance. Specifically, for a small transfer size of 4KB, changing the default VFD to core VFD improves write performance 4.8× whereas applying chunking optimization of 1MB with default chunk cache parameters also improves write performance by 2× (Figure 1 Write BW). However, the overall bandwidth of the workload does not improve proportionally; for CoreVFD, the cost of flushing the data from memory to the parallel file system (PFS) slows down the overall process, which negates the benefit of in-memory writes; for Chunking, the creation of the chunked layout along with flushing of chunk cache dominates the overall I/O time (Figure 1 Overall BW). Furthermore, when we apply both these optimizations together, the performance is not compounded as these optimizations have a side effect on each other's performance. For instance, applying a chunked layout with CoreVFD will change the flushing granularity from the whole file to chunks. Therefore, the improvement is only 23% over applying them individually. However, manual tuning of these parameters with CoreVFD having one large allocation can improve the overall performance of the workload by up to 4.9× when compared to the combined case of naively applying those optimizations (Figure 1 Manual). This example shows the potential of performing intent-driven optimizations in the HDF5 library.

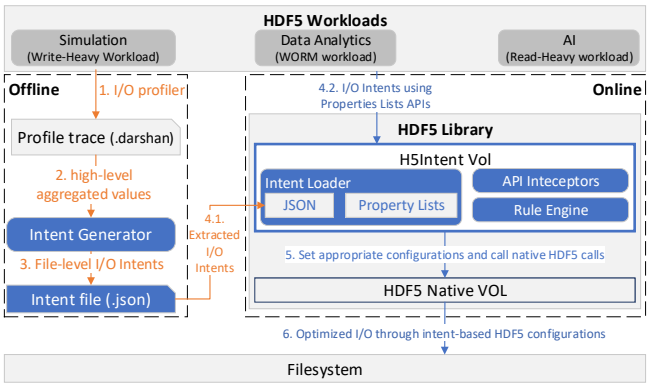


Fig. 2. At the core of the H5Intent suite is the Virtual Object Layer (VOL) that converts I/O intents into HDF5 properties. I/O intents can be passed to the VOL automatically (Orange flow) and/or Intent APIs (4.2).

3 H5Intent

We provide an end-to-end tool suite for HDF5 workloads to enable intent-driven optimizations. We utilize the custom property list infrastructure to capture I/O intents within the HDF5 library [23]. We also build an intent generator to automate the capture of I/O intent for user convenience. Finally, we design our autotuning framework as a virtual object layer (VOL) pass-thru connector for cohesive integration of user intents within the HDF5 library. The VOL is an HDF5-specific concept that allows system designers to utilize HDF5 API's and provide their own optimization within the HDF5 library. In our case, we envision a transformation of property lists to move away from library configurations and towards user-provided I/O intents. The mapping of the I/O intents into HDF5 properties would occur in the VOL layer of the HDF5 library. This transformation would increase the usability of property lists as users can describe "how" they use the HDF5 library to manage, store, and retrieve data. In turn, the HDF5 library will be able to optimize data access transparently. We aim to design a solution that is: a) *Transparent:* requires no change for legacy HDF5 apps, b) *Extensible:* can be easily extended to new data access patterns and HDF5 properties, c) *Automated:* requires minimal user intervention for its usage, and d) *Lightweight:* uses minimal memory for this automated mapping.

At a high level, the H5Intent suite consists of system-level tools that include an intent generator, a new set of intent-based properties, and a VOL layer to perform autotuning of HDF5 properties (Figure 2). At the core of the H5Intent suite is the H5Intent virtual object layer (VOL) connector. The VOL connector consists of three main components: a) Intent Loader: which parses extracted intents (offline 4.1.) and passed intents (online 4.2) into in-memory structures for quick lookup during runtime; b) API Interceptors: part of the VOL that intercept create

and open calls on every type of HDF5 object such as files, datasets, groups, and attributes; and c) Rule Engine: that converts I/O intents into HDF5 properties.

The H5Intent VOL connector can be utilized by users in two ways. The first mechanism, which is more suited for new HDF5 applications, is to invoke the custom intent API implemented through HDF5 property lists (Figure 2 4.2). This gives users fine-grained control to pass intents and benefits from our runtime autotuning of HDF5 properties. The second mechanism, more suited for legacy HDF5 applications, automatically extracts I/O intents using the Intent Generator tool (Figure 2 orange flow). For this mechanism, users must profile their applications (offline and at the same scale) using I/O profilers such as Darshan [28] or Recorder [29] (Figure 2 1.). Then, the trace file can be passed to the Intent Generator tool, which will convert high-level access pattern information into I/O intents (the "how" part) of the workload (Figure 2 2. and 3.). These extracted I/O intents are stored as JSON files which can be loaded using the VOL connector's extra arguments.

Once the H5Intent VOL loads all intents offline or online, it will wait for I/O calls on HDF5 objects. On open/create calls on these objects, the rule engine is executed for the specific I/O intent, and the rule engine recommends a collection of properties. Then, the VOL layer applies these optimizations and calls the native HDF5 VOL connector to perform the actual I/O operation (Figure 2 5.). The autotuning suggested by the VOL connector's rule engine enables transparent data access optimizations within the HDF5 library.

3.1 H5Intent VOL

We implement the autotuning of HDF5 properties using the virtual object layer (VOL) extension interface within the HDF5 library. The VOL provides a standard interface to intercept all object operations within the library with the ability to call the native HDF5 library calls. This feature makes VOL the most suitable mechanism to transparently and efficiently autotune HDF5 properties for the workloads. The H5Intent VOL connector can be provided workload's intent using the intent API directly or using a JSON file, which is automatically generated. In both cases, the VOL layer gets a list of I/O intents that needs to be converted into HDF5 Properties. The rule engine at the core is a set of if/else statements that look for specific intents and map them to properties. The main logic of the rule engine is to look for I/O intents on a given HDF5 object, resolve conflicts of multiple intents setting the same properties, and then apply the properties on the correct level of access. To extend the H5Intent VOL connector with more I/O intents, developers have to pass new Intents to the system (either through the Extractor or APIs), then enhance the Rule Engine to handle the new I/O intent, and finally resolve any conflicts it might have with existing I/O intent rules. In this work, we have implemented one rule per I/O intent, which sets multiple HDF5 properties while internally resolving conflicts. The source code for the H5Intent VOL, the Extractor, and the Rule Engine will be open-sourced on acceptance.

3.2 Mapping HDF5 I/O behaviors as I/O intents

To identify I/O intents of an application, it is sufficient to capture its I/O behavior [14]. A list of I/O behaviors that can be extracted for HDF5 files and dataset objects are summarized in Tables 1 and 2. These I/O behaviors are high-level aggregated metrics that are captured by most profilers [28]. To capture an intent for "a simulation that performs a checkpointing operation over HDF5 files using a transfer size of 4KB", Darshan captures the transfer size per file which gives us the "what" (the file names) and the "how" (the transfer sizes) of I/O intents. This level of information is easy to capture with low overhead. In addition to these behaviors, we also capture job information useful to understand the allocated resources such as nodes and file system features for the job (Table 3). In this work, all these behaviors were important for capturing I/O intents (and eventually autotuning HDF5 properties). However, only a few I/O behaviors contributed towards the most substantial performance gains, including access pattern, number of processes accessing the dataset, element distribution and size, and read-write behavior of the workload. Also, job allocation information was neccessary to accurately tune HDF5 properties. For instance, process per node information helped determine node-local vs. inter-node dataset accesses, buffer information was needed to allocate HDF5 caches, and file system features such as stripe size and counts enable alignment optimizations within HDF5.

These behaviors can be combined to build high-level I/O intents for the HDF5 workloads. A mapping of I/O intents and I/O behaviors is shown in Tables 4, 5, 6, and 7. For instance, to understand if an application performs

TABLE 1
I/O behavior for Dataset

SNo	Behavior	Description
1	close_t	Timestamp for first and last H5Dclose
2	open_t	Timestamp for first and last H5Dopen
3	write_b	Number of bytes written
4	read_b	Number of bytes read
5	write_t	Timestamp for first and last H5Dwrite
6	read_t	Timestamp for first and last H5Dread
7	access pattern (ap)	Access pattern for dataset from HDF5's filespace parameters
8	create/open API (open_api)	Dataset was created or opened
9	segment access distribution	Top 5 most accessed segments
10	element size	Size of element in dataset
11	# processes accessing ds	Process list involved in accessing dataset
12	size	Dataset size in bytes before and after workload run

TABLE 2
I/O behavior for File

SNo	Behavior	Description
13	close_t	Timestamp for first and last H5Fclose
14	open_t	Timestamp for first and last H5Fopen
15	mode	Mode for opening/creating file
16	create/open API (open_api)	File was created or opened
17	# processes accessing file	Process list involved in accessing file
18	size	File size in bytes before and after workload run

TABLE 3
Job Allocation Information

SNo	Behavior	Description
19	process per node	Number of processes allocated per node
20	buffers	List of buffers available: shm, bb, etc
21	fs	Features: stripe size and stripe count

TABLE 4
Primary I/O intents of datasets mapped to I/O behaviors

SNo	I/O Intents	Description	I/O Behavior
1	multi-session I/O	Object is opened and closed multiple times for I/O	1, 2
2	access pattern	Access pattern for dataset (write-only, read-only, read-after-write, other)	3,4,5,6
3	hot segments	Dataset segments which are most accessed	7, 9
4	transfer size	Distribution of transfer size in dataset	7, 9, 10
5	process sharing	Which processes share dataset for I/O	11
6	fs size	Final file system size for dataset	7, 8, 12
7	sharing pattern	How processes access dataset for I/O	7, 9, 10, 11

TABLE 5
Primary I/O intents for files mapped to I/O behaviors

SNo	I/O Intents	Description	I/O Behavior
1	multi-session I/O	File is opened and closed multiple times for I/O	13, 14
2	mode	Access mode (a, wo, ro, and rw)	15
3	process sharing	Which processes share file for I/O	17
4	sharing pattern	How processes access file for I/O	7, 9, 10, 17
5	fs size	Final file system size for file	18

TABLE 6
Secondary I/O intents for dataset mapped to I/O behaviors

SNo	I/O Intents	Description	I/O Behavior
1	mode	Access mode (a+, wo, ro, and rw)	15

TABLE 7
Secondary I/O intents for files mapped to I/O behaviors

SNo	I/O Intents	Description	Dataset Intents
1	ds pattern	Distribution datasets access pattern in file	2
2	ds hot segments	Distribution datasets hot segments in file	3
3	transfer size	Distribution datasets transfer size in file	4
4	ds process sharing	Distribution datasets process sharing in file	5
5	ds size on fs	Distribution of datasets sizes in file	6

multiple I/O sessions, we need information about open and close time stamps. As another example, to identify hot segments within a dataset, we need to capture the distribution of offsets accessed within a file. Two types of I/O intents are defined in these tables: primary I/O intents, which are directly derived from a group of I/O behaviors, and secondary I/O intents, which are derived from a group of primary I/O intents. For instance, hot segments within a dataset is a primary I/O intent (Table 4 row 3). However, hot segments from multiple datasets is a secondary I/O intent for the file (Table 7 row 2). Using 21 I/O behaviors extracted from HDF5 workloads, we identified 18 I/O intents across the file and dataset. Each I/O intent consists of a key identifier represented by a 16-bit unsigned int (i.e., 65,536 intents per entity) and a list of attributes collected through the I/O behaviors they map from. For instance, for hot-segments I/O intent in the dataset, we store a `uint16` key and a `map` of segment index with access frequency, first access, and last access. This additional information from I/O behavior plays an important role in tuning HDF5 properties. These I/O intent objects are encapsulated per I/O entity (defined as an job, software, or data component associated with a workload [14]) within our tools. Therefore, the footprint of an I/O entity with its I/O intents is variable and dependent on the captured I/O behavior. For our evaluations, the average memory allocated for loading I/O entities was around 6MB per node.

Although all the listed I/O intents play an important role in configuring various aspects of the HDF5 library, only a few I/O intents were key in improving workloads' I/O performance. These include access pattern, hot segments, transfer size, fs size, and process sharing. These account for almost 70% of the performance benefits we get by employing intent-based autotuning of HDF5. As seen in the mapping Tables for I/O intents, multiple I/O behaviors assist in building an I/O intent. Conversely, each I/O intent encapsulates a different aspect of the I/O behavior of the workload. Exploring different I/O intents that may combine I/O behaviors differently was important for configuring HDF5.

3.3 Mapping HDF5 I/O intents with properties

Once the user's I/O intents are extracted or passed to the software stack, we can utilize the intents to optimize the application's I/O. As detailed on our paper on Mimir [14], every layer of the software stack has its own set of intents and properties. Therefore, to correctly match these intents into HDF5 properties, we must understand the plethora of properties in the HDF5 library. As of version 1.14.0, the HDF5 library contains about 150 properties, each tuning a part of the HDF5 library's behavior. However, in practice, most users only tune a handful of parameters, but the impact of many of these parameters is not well documented. To fill this gap, we provide a collection of HDF5 properties that directly impact the I/O behavior of the application (Table 8). The list has been categorized based on HDF5 concepts such as chunking, MPI collective, caching, and layout. For each of these categories, we have multiple properties that can change the I/O behavior of the HDF5 library. Finally, these properties are applied to different objects (e.g., files and datasets) and varying types of access (e.g., creation, access, or transfer). The table comprises 37 properties categorized into 12 categories across datasets and file properties. In this study, the properties that had the most impact on improving I/O performance are chunking, caching, buffering, and MPI optimizations.

The mapping of I/O intents to their corresponding properties is a complex task. We found a few studies that detail experiences of configuring HDF5 properties. However, deriving the isolated impact on I/O performance from these existing studies was challenging. Therefore, we perform benchmarking over these properties to generate the heuristics and present our mapping of the HDF5 properties to I/O intents. Understanding the HDF5 properties in the context of I/O intents is more intuitive. For example, setting the correct chunk size parameter requires the knowledge of the application accesses without which the developers have to try an expensive brute force method for setting this parameter. Another example is understanding how the process sharing intent can guide the selection of appropriate file drivers and data synchronization primitives for the workload. Intuitively, we want to use the "how" of I/O intents to set these properties. For instance, an intent for "a simulation that performs a checkpointing operation over HDF5 files using a transfer size of 4KB for a 1GB dataset", we want to use the core VFD optimization with buffer size as the file that is written by the workload. This will buffer all the checkpoints in memory and then flush the 1GB file to parallel file system.

TABLE 8
Selected HDF5 properties categorized based on intended behavior

S.No.	Conf Type	Properties	Property Class
1	Chunking	chunk_cache	Dataset Access
		chunk	Dataset Creation
		chunk_opts	Dataset Creation
2	MPI	dxpl_mpio	Dataset Transfer
		dxpl_mpio_collective_opt	Dataset Transfer
		fapl_mpio	File Access
		mpi_params	File Access
3	MPI + Chunking	dxpl_mpio_chunk_opt	Dataset Transfer
		dxpl_mpio_chunk_opt_num	Dataset Transfer
		dxpl_mpio_chunk_opt_ratio	Dataset Transfer
4	Reduction	szip	Dataset Creation
		deflate	Dataset Creation
5	Layout	layout	Dataset Creation
		small_data_block_size	File Access
6	Integrity	edc_check	Dataset Transfer
		fapl_direct	File Access
		core_write_tracking	File Access
7	Data	hyper_vector_size	Dataset Transfer
		dataset_io_hyperslab_selection	Dataset Transfer
8	Caching	fapl_core	File Access
		cache	File Access
		file_image	File Access
		file_image_callbacks	File Access
		sieve_buf_size	File Access
9	Caching + Chunking	chunk_cache	Dataset Access
		file_space_page_size	File Creation
		file_space_page_size	File Creation
		file_space	File Creation
10	Buffering	fapl_stdio	File Access
		page_buffer_size	File Access
11	Metadata	evict_on_close	File Access
		fclose_degree	File Access
		file_locking	File Access
		gc_references	File Access
		meta_block_size	File Access
12	MPI + Metadata	coll_metadata_write	File Access
		all_coll_metadata_ops	File Creation

Table 9 presents our mapping of I/O intents to corresponding HDF5 properties (from Table 8). This table maps each I/O intent (in column 3) to a list of properties in HDF5. In most cases, I/O intents have exclusive HDF5 properties. However, a few HDF5 properties are generic enough to overlap among multiple intents. In these cases, we have special handling of setting properties, dependent on the list of intents present for the workload. For instance, chunking parameters are used in most I/O intents as their value would impact I/O behavior for many I/O intents, such as access pattern, hot segments, transfer size, and process sharing. In these cases, we derive a value for chunking based on available intents on an HDF5 object. Also, `H5Pset_dataset_io_hyperslab_selection` (in Table 8 Data Operation category) is used to optimize data selection based on transfer size I/O intent as well as file access mode intent (i.e., Table 9 row 4 and 9).

Simultaneously configuring multiple HDF5 properties (out of about 150) may have unexpected side effects. For instance, chunking datasets into large pieces will change how processes write to a chunk (shared or independent). This will, in turn, change the I/O performance of the HDF5 library. This use case was shown in the motivation section, where incorrectly combining Core-VFD and chunking could not achieve the desired I/O boost. However, as we capture I/O intents, we can set these properties to eliminate side effects and work together cohesively to achieve the best possible I/O performance. For instance, let us assume we have an access pattern where the workload collectively writes a fixed number of elements on the dataset. We

TABLE 9
Mapping Intents to HDF5 Properties

S.No	Category	I/O Intents	Properties Type Index
1	Dataset Primary	mutli-session I/O	9
		access pattern	1, 7
		hot segments	1, 4, 8, 9, 10
		transfer size	1, 3, 7
		process sharing	1, 2, 3, 12
		fs size	1, 8, 9
		sharing pattern	1, 2, 3, 12
2	File Primary	multi-session I/O	9
		mode	7, 8, 9, 11
		process sharing	1, 2, 3, 12
		sharing pattern	1, 2, 3, 12
		fs size	1, 8, 9
		data integrity	
3	Dataset Secondary	mode	7, 8, 9, 11
4	File Secondary	ds pattern	1, 5, 7
		ds hot segments	1, 4, 8, 9, 10
		transfer size	1, 3, 7
		ds process sharing	1, 2, 3, 12
		ds process sharing	1, 8, 9

can identify this workload behavior by examining access patterns, transfer sizes, and process sharing I/O intents. In this case, we select HDF5 properties for chunking, MPI, and MPI+Chunking to perform I/O with a chunk size equal to the transfer size of each process and set MPI parameters to do collective I/O on the chunk with system-specific aggregators, and set MPI-chunking options to use independent I/O for more performance. As we show in the next section, setting these complex parameters with I/O intents becomes simple and efficient.

3.4 Illustrative Example

We present an example to illustrate the mappings of I/O behavior to I/O intents to HDF5 properties. Let us consider a simulation that performs a checkpointing operation over HDF5 files using a transfer size of 4KB (using a hyperslab selection) over a 1GB HDF5 dataset (Figure 3). The I/O behaviors for the workload are shown in Figure 3 I/O behaviors. These I/O behaviors can be categorized into I/O intent as described in the above sections with specific intent attributes as shown in Figure 3 I/O Intents. These I/O intents, with their corresponding attributes, guide setting various HDF5 properties as shown in the Figure 3 HDF5 Properties. For instance, a transfer size of 4KB will result in a small I/O on the file system. As the application makes multiple such requests, we can aggregate them into a chunk of size 1MB to optimize data access. Similarly, as the file size is 1GB, we can use the Core-VFD first to cache all I/O and then flush the whole file to the PFS. Here, since we know the size of file, we can do one large allocation during file creation which

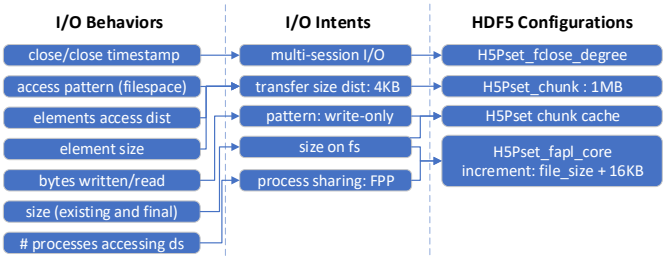


Fig. 3. I/O behaviors extracted from a simulation that checkpoints using a file-per-process pattern. The I/O behaviors are mapped to I/O intents and then to HDF5 properties using the information from Tables 1-8.

```
1.  int main(int argc, char* argv[]) {
2.    /* init program structures*/
      ...
3.    // file, dataset creation and access property list: H5Pcreate
      ...
4.     // app write 4096 data twice in file-per-process (fpp) pattern
5.     H5Pset_file_mode(fapl_id, H5INTENT_WO); // write-only
6.     H5Pset_process_sharing(fapl_id, &rank); // fpp
7.     file_id = H5Fcreate("my_file.h5", flags, H5P_DEFAULT, fapl_id);
8.     id = H5Dcreate(file_id, "my_dataset", H5T_INT32, dataspace);
9.     // xfer size dist
10.    H5Pset_xfer_intent(dxpl_id, [0,4096], [4096,4096]);
11.    //hyperslabs for memory and dataspace for writing 4096 data
12.    H5Dwrite(d_id, datatype, mem_4096, ds_0_4KB, dxpl_id, &data);
13.    H5Dwrite(d_id, datatype, mem_4096, ds_4KB_8KB,  dxpl_id, &data);
14.    // close all ids
15.    H5Dclose(d_id); H5Fclose(file_id); // intents get removed
16.    // app reads 8096 data
17.    H5Pset_file_mode(fapl_id, H5INTENT_RO); // read-only
18.    H5Pset_process_sharing(fapl_id, &rank); // fpp
19.    file_id = H5Fopen("my_file.h5", H5F_ACC_RDONLY,  H5P_DEFAULT,
                                                        fapl_id);
20.    H5Pset_xfer_intent(dxpl_id, [0], [8096]); // xfer size dist
21.    d_id = H5Dopen(file_id, "my_dataset");
22.    H5Dread(d_id, datatype, H5S_ALL, H5S_ALL, dxpl_id, &read_data);
23.    // close all ids
24.    H5Dclose(d_id); H5Fclose(file_id);
25.  }
```

Fig. 4. Example of using H5Intent properties to set I/O intents. Each I/O intent can be applied to the correct scope so that the H5Intent VOL can optimize the data access.

is more efficient than multiple small increments in memory [6]. As we know the exact I/O behavior of the workload through I/O intents, we can accurately set each parameter automatically for the workload. The performance implication of these properties is similar to our motivating example in Section 2.4. Specifically, applying these properties can improve I/O performance by up to 4.9× compared to the combined case of naively applying those optimizations.

3.5 I/O intents adapter for HDF5

We developed adapters to be transparent mechanisms to integrate intent within the existing software stack, enabling existing I/O software like HDF5 to expose intent-based APIs that allow users to interact with only one interface (i.e., their traditional I/O interface) and use I/O intents implicitly behind the scenes [14]. For HDF5, the property lists infrastructure already provides a flexible and extensible way to add APIs for each I/O intent in the system and map it to different objects at runtime. We use a user-defined property lists [23] to define each intent within the HDF5 library for a specific object (e.g., file or dataset) and access type (create, access, or transfer). We use the `H5Pregister2` function to create and register an intent within the HDF5 library. The H5P interface would provide finer control of setting intents of individual datasets, files, or phases of I/O. This allows new applications to take advantage of HDF5's intent APIs to optimize their workload. We register a callback while creating these property lists. This callback converts the user inputs from the HDF5 application into a data structure for utilization during I/O calls. This conversion enables efficient $O(1)$ access to each property list by storing these properties in `unordered_map`. An example of using the property list APIs (Figure 4).

3.6 Automatic Extraction of Intents for legacy workload

User intents (especially the "how") can be automatically extracted by analyzing the I/O profile of the workload. In this work, we built an Extractor to convert I/O behaviors of file and dataset objects into I/O intents of the HDF5 workload. Additionally, we extract job-level information using JobUtility. These tools extract information from the workloads and build a JSON file of user intents. During the workload execution, we attach a Darshan profiler [28] that provides high-level counters for HDF5. After workload execution, we run our `JobUtility` tool to extract job information (e.g., the total number of nodes and number of CPU and GPU cores allocated per node) and data intent (e.g., number of files in the dataset, the format of files in the dataset, and size of each file). Darshan and utility logs are collected and stored in a global file system such as GPFS. After job execution, we use the `Extractor`, written in Python, using py-darshan to extract high-level counters. We write mappings to convert the extracted I/O behaviors into I/O intents and store them in JSON format. The H5Intent VOL connector loads this JSON file during runtime to generate HDF5 properties by using the Rule Engine.

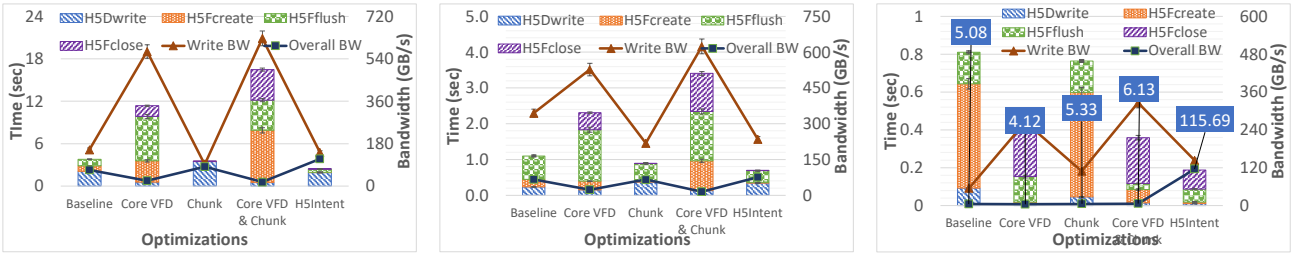
3.7 Comparision of Intents from different layers

The nature of I/O intents depends on the layer on which they are applied.. In our previous work, Mimir, we explored the POSIX layer, expressing I/O intents for bytes of data on files. In contrast, HDF5 has a rich application interface with several abstractions that help application scientists. This motivates this work, exploring different I/O intents applicable to this high-level I/O library. In terms of applying I/O intents, as the HDF5 interface fits more naturally with the way application scientists think about their data, it fits more naturally in the concept of I/O intents. Thus, we were able to represent higher-level concepts familiar to application scientists in the I/O intents for HDF5. However, HDF5 abstractions come with many optimizations compared to fewer optimizations available for the POSIX layer. This made mapping I/O intents to these optimizations and configurations much more complex to implement within H5Intent. Finally, intents for both layers differ based on the information available at each layer. As HDF5 supports richer abstractions, the amount of information per intent is more than the POSIX layer.

4 Evaluation

To evaluate the benefit of autotuning HDF5 properties over parallel file systems (PFS), we perform experiments across various workloads, including different transfer sizes, file-sharing patterns, and heterogeneous workloads. Additionally, we evaluate H5Intent with large-scale HDF5 workloads to demonstrate the use of the complete system. In this evaluation, we used GPFS as the baseline PFS and ran these tests five times with the variance in the data noted in the figures. In all these tests, we observe an average memory footprint of 6.74 MB per node on the Lassen supercomputer with I/O intents stored for file and dataset objects. Note that the idea behind these tests is to demonstrate the adaptability of intent-based autotuning using properties in the HDF5 library.

Hardware: We run the workloads on the Lassen supercomputer at Lawrence Livermore National Laboratory (LLNL) [27]. A 23-petaflop IBM Power9 system consists of 795 nodes connected with a Mellanox 100 Gb/s enhanced data rate (EDR) InfiniBand network and a 24



(a) Large Stripe: 1MB block size. (b) Medium Stripe: 64KB block size. (c) Small Stripe: 4KB block size.

Fig. 5. Manually setting HDF5 properties for different workloads can lead to different impacts on I/O. Additionally, combining these properties needs special care; otherwise can lead to sub-optimal results. I/O intents provide the necessary information to make this choice heuristically.

PiB IBM Spectrum Scale file system (also known as GPFS). Individual Lassen nodes consist of two IBM POWER9 CPUs (IBM AC922 servers) with a total of 40 cores per node and 256GB of system memory.

Workloads: We use the HDF5 benchmark, called H5bench [30], to showcase various workloads on the designed components. H5bench is a suite of parallel I/O benchmarks or kernels representing I/O patterns that are commonly used in HDF5 applications on HPC systems. Specifically, we use the *h5bench_write* and *h5bench_read* kernels with striping properties to emulate different data access patterns of HDF5 workloads. We use three striping properties, namely a) large-stripe: 32 blocks of size 1MB each, b) medium-stripe: 128 blocks of 64KB each, and c) small-stripe: 128 blocks of 4KB each. Both kernels of H5bench perform I/O operations on eight dataset objects. All timings and bandwidths reported for the benchmarking is the output from the H5bench benchmark. We also employ two scientific HPC workloads, VPIC-IO [9] and BD-CATS-IO [10], which are I/O kernels of a large-scale space weather plasma simulation code and a corresponding data analysis code. In these tests, H5Intent uses Darshan profiling to extract the I/O behaviors (described in Table 1, 2, and 3), which incurs an overhead of up to 1% for the workloads tested. Additionally, the online Rule Engine (O(1) algorithm) cost is 15us per I/O operation, which is 0.02% of the application's execution.

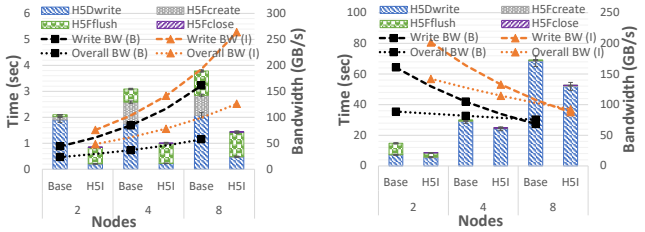
4.1 The impact of HDF5 properties

To demonstrate this complexity and highlight intent-aware autotuning, we perform an experiment by applying specific HDF5 properties individually and together, and measure their impact on the workload's I/O performance. Specifically, we run the *h5bench_write* workload on four nodes (160 processes) with a file-per-process (FPP) pattern with three stripes (as described in the workloads section). The x-axis shows the different optimizations performed, the primary y-axis shows the time in seconds, and the secondary y-axis shows the aggregate I/O bandwidth achieved across all ranks in GB/s.

The default property of HDF5 properties provides relatively good performance for HDF5 workloads (Figure 5). In these tests, as the data is sequentially written with a fixed block size and the I/O per-node fits in memory, popular recommendations from the HDF5 library are to use dataset chunking and Core-VFD to optimize writes to the HDF5 file. Specifically, we set the chunk size [23] (for chunking) to align with the GPFS stripe size of 1MB, and the increments in Core-VFD [23] equal to the transfer size. For the Core-VFD optimization, the write bandwidth (Figure 5 brown line with a triangle) improves significantly as all I/O on the dataset happens in memory. However, the file creation (`H5Fcreate`), file flush (`H5Fflush`), and file close (`H5Fclose`) times increase for the Core-VFD. This is because of the memory allocation cost and flushing cost of the Core-VFD driver. Specifically, we see an average increase in write bandwidth by 3.3×, but the overall bandwidth (i.e., including metadata) decreases by almost 3× in some cases.

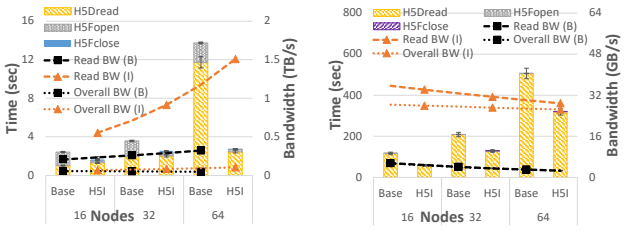
For the chunking optimization, we see that the write bandwidth improves for the small-stripe transfer size, but as the transfer size is relatively big for medium and large-stripe cases, the chunking does not improve the write performance of the workloads. However, the metadata and flushing cost reduces as compared to the baseline. Therefore, the application's overall performance improves by 10% for large and medium stripes and about 2× for small stripes. Finally, naively applying both optimizations together with the same parameters reduces the performance of the workloads by 4.2× for large and medium stripes and improves the small stripes by just 20%. These tests show the complexity of setting properties to optimize I/O workloads. This is because applying the chunking optimization reduces the flush granularity of the HDF5 library and therefore reduces the overall performance.

For H5Intent, we get workload information from the I/O intents described in Table 9. Therefore, H5Intent can optimize correctly for each workload while combining the properties based on the Rule Engine. For fairness, we restrict H5Intent to only two properties used by our baseline and present the result in Figure 5 as H5Intent. Specifically, based on the intents for each case, H5Intent sets the chunk size to be transfer size only if chunking is applied individually. If the Core-VFD optimization is involved, we do not use the chunking optimization to allow the HDF5 library to flush the whole file in one operation. Additionally, we set the CoreVFD increments to the file size to allocate one large memory buffer at the start of the application instead of many small memory reallocations. This workload-aware optimization from H5Intent balances the write cost with metadata cost to improve overall performance. Specifically, we see that we improve the application's overall performance by 1.68×, 1.13×, and 23× compared to the baseline. If we compare the best optimization case among the three optimization baselines again, we have an average improvement of 7× across all stripe cases. These tests demonstrate the complexity of manually optimizing HDF5 properties and the role I/O intents can play in



(a) Large stripe and FPP. (b) Small stripe and shared file.

Fig. 6. Under different workloads, fine-tuning HDF5 properties without workload-knowledge is very difficult. Introducing intents from workloads can assist in the accurate setting of parameters. We can achieve up to 2.1× by performing intent-based HDF5 autotuning.



(a) File-per-process (FPP). (b) Shared file.

Fig. 7. For workloads with different data access patterns within phases, intent-based HDF5 properties reduce the complexity of autotuning HDF5 and optimize workloads by up to 8.8×.

autotuning the HDF5 library for HPC workloads.

4.2 The benefit of H5Intent with file-sharing pattern

The HDF5 library allows users to utilize two file-sharing patterns: a) file-per-process and b) shared file access. In both cases, the library sets default properties that might optimize users' workloads. However, each application is unique; therefore, the library can not achieve optimal I/O performance over the storage system. To demonstrate the role of I/O intents through H5Intent in these workloads, we run the *h5bench_write* workload and show FPP with large-stripe and shared file pattern with small-stripe. We weakly scale the workload from 2 to 8 nodes (i.e., 320 processes in the largest scale). The x-axis shows the scaling of nodes, the primary y-axis shows the time in seconds, and the secondary y-axis shows the aggregate I/O bandwidth achieved across all ranks in GB/s.

For file-per-process with large stripes, the default property of the HDF5 library improves the workload's write and overall I/O bandwidth (Figure 6(a) black lines with square) but it is sub-linear to the weak scaling. This is because we increase flushing costs (during `H5Fflush` and `H5Fclose`) as we increase node counts. Specifically, the write bandwidth increases almost linearly (i.e., 1.92×), whereas the overall bandwidth only increases by 1.5×. In the case of H5Intent, we capture the I/O intents with the following values; a) transfer size: 1MB, b) access pattern: write-only, c) the size on fs: 1GB per file, d) hot segments: sequential access on recency, and e) process sharing: file-per-process. Using these intents for the 8-node case, the Rule Engine configured HDF5 with the following final file-level properties; a) Core-VFD: increments-1GB, b) cache: size-1GB and slots-320, c) file close degree: STRONG, and d) page buffer: size-1GB. Also, the Rule Engine configured HDF5 with the following final dataset-level properties; a) chunk cache: size-128MB and slots-1, b) chunk: size-32MB, and hyperslab selection: offset-0, size-32MB. With these properties, we see that the write bandwidth of H5Intent improves by 1.58× over the baseline (Figure 6(a) dashed lines with a triangle). This optimizes the overall bandwidth by 2.1× on the largest scale (Figure 6(a) dotted lines with a triangle). This is because we fine-tuned the HDF5 dataset chunk caches so that the HDF5 library can perform large flushes between the memory and the file system by accurately setting the cache parameters for the dataset.

For shared-file with small stripes, the default property of the HDF5 library does not scale with the number of nodes as both write, and the overall I/O bandwidth of the workload decrease by 53% (Figure 6(a) black lines with square). In this case, the MPI-IO driver has a lot of contention as it does not know how the workload is accessing the dataset. In the case of H5Intent, we capture I/O intents with the following values; a) transfer size: 4KB, b) access pattern: write-only, c) the size on fs: 160MB, d) hot segments: sequential access on recency, and e) process sharing: shared-file collective I/O. Using these intents for the eight-node case, the Rule Engine configured HDF5 with the following final file-level properties; a) *mpiio*: comm-size, b) *cache*: size-4MB and blocks-320, and c) *close degree*: STRONG. Also, the Rule Engine configured HDF5 with the following final dataset-level properties; a) *chunk cache*: size-2MB and slots-320, b) *chunk*: size-160KB, c) *mpiio chunk opt*: multi, d) mpiio chunk opt num: 40, e) *mpiio chunk opt per*: 12.5%, and f) *hyperslab selection*: offset-0, size-16MB. These properties improve write bandwidth by 28% and overall bandwidth by 41% (Figure 6(b) orange lines with a triangle). This is because all processes contribute 4KB requests together on a chunk of 2MB which improves the overall granularity of I/O through MPI-IO. Additionally, the library is aware of this behavior and can apply MPI-IO level optimizations for the workload.

These tests demonstrate the effectiveness of passing accurate I/O intents (with behaviors) to the HDF5 library. We can see fine-tuning even in complex cases for PFS, ssuch as small data shared-file accesses to the PFS, can be optimized within the HDF5 libraries using H5Intent.

4.3 Benefit of H5Intent with heterogeneous workloads

In all the above tests, the I/O on the datasets was homogenous. However, when different datasets within a workload exhibit different I/O behaviors, tuning the library for each dataset and the whole file is a more complex task. To demonstrate the role of I/O intents through H5Intent in heterogeneous workloads, we run the *h5bench_read* workload and show FPP and shared file pattern with all stripes (i.e., large, medium, and small stripe). We weakly scale the workload with 16, 32, and 64 node (i.e., 2560 processes in the largest scale). The x-axis shows the scaling of nodes, the primary y-axis shows the time in seconds, and the secondary y-axis shows the aggregate I/O bandwidth achieved across all ranks in GB/s.

For file-per-process, the default property of the HDF5 library improves the read bandwidth by only 21% and the overall bandwidth by about 9% (Figure 7(a) black lines with square). This is because, by default, read access on the HDF5 file does not avoid file locking as the benchmark

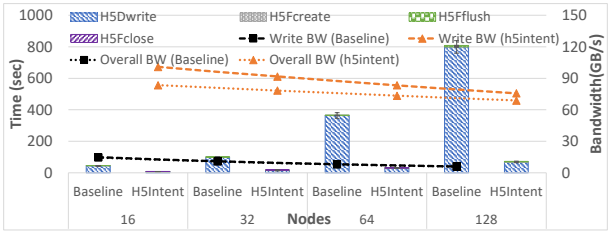


Fig. 8. H5Intent optimizes the large writes from VPIC by up to 12× by dynamically tuning the chunking and MPI-IO parameters.

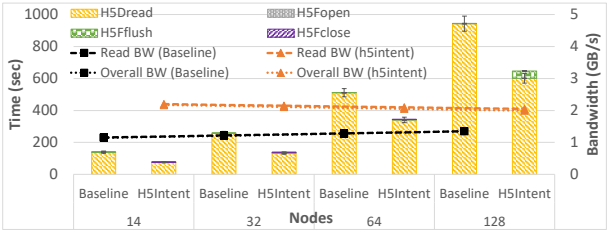


Fig. 9. BD-CATs is optimized by H5Intent VOL connector's autotuning of HDF5 properties by up to 68%.

opens the HDF5 file in the read-write mode. Additionally, this bandwidth is dominated by the small transfer size operations, and therefore, the improvements from large accesses are negated by the slowdown from small accesses. In the case of H5Intent, we capture I/O intents with the following values; a) transfer size: 4KB, 64KB, and 1MB each 33% accesses, b) access pattern: read-only, c) the size on fs: 1.26GB per file, d) hot segments: sequential access on recency, and e) process sharing: file-per-process. Using these intents for the 64-node case, the Rule Engine configured HDF5 with the following final file-level properties; a) Core-VFD: increments-160MB, b) cache: size-160MB and slots-320, c) file close degree: STRONG, d) page buffer: size-160MB, and e) file-locking: disabled. Also, the Rule Engine configured HDF5 with the following final dataset-level properties; a) chunk cache: size-160MB and slots-1, b) chunk: size-32MB for large, size-2MB for medium, and 512KB for the small dataset, and hyperslab selection: offset-0, size-32MB for large, size-2MB for medium, and 512KB for the small dataset. As the dataset properties are based on per-dataset I/O intents, the specialized property for this workload results in 3.6× speedup in read bandwidth and 2.2× speed up on the overall bandwidth in the largest scale (Figure 7(a) orange lines with a triangle). Finally, as the scale of the workload increases, the benefit from the file's specialization also increases.

For the shared file case, the default property of the HDF5 library does not scale well as we increase the node count. Specifically, we see a slowdown of 35% in read performance. Again, similar to the FPP case, the MPI-IO driver is not optimized for heterogeneous dataset transfer sizes. In the case of H5Intent, we capture I/O intents with the following values; a) transfer size: 4KB, 64KB, and 1MB each 33% accesses, b) access pattern: read-only, c) the size on fs: 3.1TB, d) hot segments: sequential access on recency, and e) process sharing: shared-file collective I/O. Using these intents for the 64 node case, the Rule Engine configured HDF5 with the following final file-level properties; a) *mpiio*: comm-size, b) *cache*: size-50GB and blocks-320, c) *close degree*: strong, and e) *file-locking*: disabled. Also, the Rule Engine configured HDF5 with the following final dataset-level properties; a) *chunk cache*: size-52GB and slots-320, b) *chunk*: size-1.2GB for large, size-80MB for medium, and 20MB for the small dataset, c) *mpiio chunk opt*: multi, d) *mpiio chunk opt num*: 40, e) *mpiio chunk opt per*: 1.56%, and f) *hyperslab selection*: offset-0, size-32MB for large, size-2MB for medium, and 512KB for the small dataset. With these properties for the workload, we see an aggregate overall bandwidth increase of 7× and an aggregate read bandwidth increase of 7.8×. The benefit of fine-tuning HDF5 properties increases on larger scales by up to 8.8× for 64 nodes.

4.4 Large-scale workload performance

4.4.1 VPIC

Scientific simulations, such as VPIC [9], typically progress in time steps. After one or more time steps of computations, all processes collectively checkpoint data to the storage system. We use the VPIC-IO kernel to evaluate how an application will benefit from H5Intent. In VPIC-IO, each process writes eight variables with a total size of 256MB in each time step. We run VPIC for ten-time steps. Essentially, VPIC creates a single large file that stores all the variables and time steps. We weakly scale the workload with 16-128 node (i.e., 5120 processes in the largest scale). The x-axis shows the scaling of nodes, the primary y-axis shows the time in seconds, and the secondary y-axis shows the aggregate I/O bandwidth achieved across all ranks in GB/s. The default properties of the HDF5 library do not scale well as we increase the number of nodes (Figure 8 Baseline). Specifically, the write bandwidth reduces by 34% as we double the number of nodes. Similarly to the shared file benchmark, the MPI-IO driver has a lot of contention as it does not know how the workload is accessing the dataset. In the case of H5Intent, we capture I/O intents with the following values for 128-node case; a) transfer size: 1MB and 256KB, b) access pattern: write-only, c) the size on fs: 1.2TB, d) hot segments: sequential access on recency, and e) process sharing: shared-file collective I/O. Using these intents for the 128-node case, the Rule Engine configured HDF5 with the following final file-level properties; a) *mpiio*: comm-size, b) *cache*: size-1MB and blocks-5120, and c) *close degree*: strong. Also, the Rule Engine configured HDF5 with the following final dataset-level properties; a) *chunk cache*: size-1MB and slots-5120, b) *chunk*: size-10GB, c) *mpiio chunk opt*: multi, d) *mpiio chunk opt num*: 40, e) *mpiio chunk opt per*: 0.78%, and f) *hyperslab selection*: offset-0, size-256MB. With these properties, we see the write bandwidth, as well as the overall bandwidth of VPIC, increases at an average of 8.33×. Additionally, as the scale increases, the benefit of fine-tuning increases up to 12.45× write bandwidth for 128 nodes.

4.4.2 BD-CATS

Scientific applications often involve workflows where data producers and consumers share data. BD-CATS is an analytics application that reads data produced by VPIC and finds centroids of clusters [10]. We use the BD-CATS IO kernel to evaluate how an application will benefit from H5Intent. In BD-CATS-IO, each process reads back eight variables with a total size of 256MB in each time step from a single shared file (as written by VPIC). We run BD-CATS for ten time steps

on 16-128 node (i.e., 5120 processes in the largest scale). The x-axis shows the scaling of nodes, the primary y-axis shows the time in seconds, and the secondary y-axis shows the aggregate I/O bandwidth achieved across all ranks in GB/s. The default properties of the HDF5 library do not scale well as we increase the number of nodes (Figure 9 Baseline). Specifically, the read bandwidth increases by only 5% as we double the number of nodes. Similarly to the shared file benchmark, the MPI-IO driver has a lot of contention as it does not know how the workload is accessing the dataset. In the case of H5Intent, the I/O intents and Rule Engine output were similar to VPIC. Therefore, we highlight the differences here. We capture I/O intents; a) access pattern: read-only. Using these intents for the 128-node case, the Rule Engine configured HDF5 with the following final file-level properties; a) file-locking: disabled. Also, the Rule Engine configured HDF5 with the same final dataset-level properties as VPIC. With these properties, the BD-CATS application gets 68% higher bandwidth than the Baseline. This increase is not as substantial as in our other micro-benchmarks. The reason is that the number of I/O operations is only eight here as opposed to thousands in the other micro-benchmarks.

5 Related Work

Autotuning for HPC applications applies to many use cases ranging from compile-time to run-time optimizations [31]. Specifically for parallel I/O, researchers have tuned I/O parameters using heuristics, genetic algorithms, and even machine learning techniques. SHAMan is an autotuning framework for HPC I/O parameters that utilizes heuristics to find the optimal parametrization of an Input/Output (I/O) accelerator for an HPC application [32]. Researchers in another reference optimize I/O parameters for LSM-tree-based key-value stores by quantitatively studying and comparing five optimization methods [33]. Finally, the authors of CAPES propose a q-learning approach using AI techniques to optimize the performance of Lustre file systems and compare their results with genetic algorithms and Bayesian optimization. Specifically for the HDF5 library, the authors of H5Evolve [13] propose using a genetic algorithm to explore the parameter space for understanding which parameters improve I/O performance based on benchmarking results. Also, authors of [34] present an I/O autotuning framework for the MPI-IO library with a naive strategy based autotuning that tries all possible combinations of configuration parameters to find the best. All these references highlight the complexity of fine-tuning I/O based on workloads. In this work, we propose to use I/O intents so that the storage systems do not need to run expensive autotuning algorithms. This work extends our previous work by defining HDF5-based I/O intents, designing an adaptor based on HDF5 property lists, and demonstrating use cases of using HDF5 properties correctly for the HPC community.

6 Conclusion

This work showcases the importance of I/O intents for autotuning HDF5 configurations for various I/O workloads using micro-benchmarks and real large-scale applications. We detail the design of the H5Intent software suite, which consists of a) the Intent Generator that automatically maps I/O behavior from darshan traces into I/O intents, b) the Rule Engine that dynamically maps I/O behaviors to HDF5 configurations while managing conflicts, and c) the H5Intent VOL that intercepts HDF5 applications at runtime to transparently configure HDF5 applications based on the rules. Additionally, we demonstrate that using I/O intents to define HDF5 configurations using our Rule Engine can improve the I/O performance of synthetic workloads by up to 22×. Also, we showcase that the H5Intent VOL has a small footprint of 6.74MB per node and 7.24MB per application for file-per-process and shared file access patterns, respectively. Finally, we demonstrate the benefit of I/O intents for diverse I/O behaviors within a workflow through our micro-benchmarking and improve I/O performance by up to 8.8×. In conclusion, I/O intents can enable workload-aware optimizations for large-scale workloads such as VPIC and BD-CATS with an I/O performance improvement of 11× and 1.48×, respectively, on the Lassen supercomputer.

As a part of future work, we plan to incorporate H5Intent as a "root" VOL, which dynamically configures HDF5 properties and other data management VOLs for HPC systems. Additionally, we plan to extend the heuristics in mapping intents to HDF5 properties with a machine learning algorithm that can learn this mapping based on data. Also, we plan to extend our VOL to introduce internal capturing of intents to validate intents provided by the user or capturing new intents. Finally, we will extend our work to capture I/O intents online that can more effective for workloads with irregular patterns.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program (LLNL-CONF-849011).

References

[1] M. Mercier, D. Glesser, Y. Georgiou, and O. Richard, "Big data and HPC collocation: Using HPC idle resources for Big Data analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec. 2017, pp. 347–352.

[2] H. Luu, B. Behzad, R. Aydt, and M. Winslett, "A multi-level approach for understanding I/O activity in HPC applications," in *2013 IEEE International Conference on Cluster Computing*, Sep. 2013, pp. 1–5.

[3] F. Chowdhury, Y. Zhu, F. Di Natale, A. Moody, E. Gonsiorowski, K. Mohror, and W. Yu, "Emulating I/O Behavior in Scientific Workflows on High Performance Computing Systems," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop*, Nov. 2020, pp. 34–39.

[4] H. Devarajan and K. Mohror, "Extracting and characterizing I/O behavior of HPC workloads," in *The 2022 IEEE International Conference on Cluster Computing*. Heidelberg, Germany: IEEE, Sep. 2022.

[5] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming parallel I/O complexity with auto-tuning," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2013, pp. 1–12, iSSN: 2167-4337.

[6] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 219–230.

[7] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, K. Mohror, D. Ivanov, T. Wang, and C. P. Steffen, "UnifyFS: A Distributed Burst Buffer File System - 0.1.0," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. UnifyFS, Oct. 2017. [Online]. Available: https://www.osti.gov/biblio/1408515

[8] 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*, ser. AD '11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 36–47.

[9] N. P. Tan, R. B. F. Bird, G. Chen, S. V. Luedtke, B. J. Albright, and M. Taufer, "Analysis of Vector Particle-In-Cell (VPIC) memory usage optimizations on cutting-edge computer architectures," *Journal of Computational Science*, vol. 60, Jan. 2022, institution: Los Alamos National Lab. (LANL), Los Alamos, NM (United States) Number: LA-UR-21-31487 Publisher: Elsevier. [Online]. Available: https://www.osti.gov/pages/biblio/1844166

[10] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "BD-CATS: big data clustering at trillion particle scale," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2015, pp. 1–12.

[11] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. Van Essen, "The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs With Hybrid Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1641–1652, Jul. 2021, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[12] H. Devarajan, A. Kougkas, L. Logan, and X.-H. Sun, "HCompress: Hierarchical Data Compression for Multi-Tiered Storage Environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 557–566, iSSN: 1530-2075.

[13] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing I/O Performance of HPC Applications with Autotuning," *ACM Transactions on Parallel Computing*, vol. 5, no. 4, pp. 1–27, Dec. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3309205

[14] H. Devarajan and K. Mohror, "Mimir: Extending I/O Interfaces to Express User Intent for Complex Workloads in HPC," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2023, pp. 178–188, iSSN: 1530-2075.

[15] J. Lüttgau, M. Kuhn, K. Duwe, Y. Alforov, E. Betke, J. Kunkel, and T. Ludwig, "Survey of storage systems for high-performance computing," *Supercomputing Frontiers and Innovations*, vol. 5, no. 1, 2018, number: 1 Publisher: South Urals University. [Online]. Available: https://centaur.reading.ac.uk/77664/

[16] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," 2002. [Online]. Available: https://shorturl.at/yLV6P

[17] P. Braam, "The Lustre Storage Architecture," *arXiv:1903.01955 [cs]*, Mar. 2019, arXiv: 1903.01955. [Online]. Available: http://arxiv.org/abs/1903.01955

[18] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, and S. Oral, "UnifyFS: A User-level Shared File System for Unified Access to Distributed Local Storage," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2023, pp. 290–300, iSSN: 1530-2075.

[19] B. R. Landsteiner, D. Henseler, D. Petesch, and N. J. Wright, "Architecture and Design of Cray DataWarp," 2016. [Online]. Available: https://shorturl.at/5pQ2Q

[20] W. Schenck, S. El Sayed, M. Foszczynski, W. Homberg, and D. Pleiter, "Evaluation and Performance Modeling of a Burst Buffer Solution," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 12–26, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3041710.3041714

[21] "fadvise(2): Give advice about file access - Linux man page." [Online]. Available: https://linux.die.net/man/2/fadvise

[22] C. H. H. Lunde, "Improving Disk I/O Performance on Linux," 2009, accepted: 2013-03-12T08:03:49Z. [Online]. Available: https://www.duo.uio.no/handle/10852/10099

[23] T. H. Group, "HDF5: Dataset Creation Properties." [Online]. Available: https://shorturl.at/kkb6D

[24] D. Robinson, "RFC: Core VFD Backing Store Paged Writes," *docs.hdfgroup.org*, p. 6, Sep. 2013.

[25] M. S. Breitenfeld, "Parallel I/O with HDF5 and Performance Tuning Techniques."

[26] A. Bagbaba, "A Comparative Study of MPI-IO Libraries for Offloading of Collective I/O Tasks," in *2021 International Conference on Engineering and Emerging Technologies (ICEET)*, Oct. 2021, pp. 1–6, iSSN: 2409-2983.

[27] "Lassen High Performance Computing." [Online]. Available: https://hpc.llnl.gov/hardware/platforms/lassen

[28] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of petascale I/O workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*, Aug. 2009, pp. 1–10, iSSN: 2168-9253.

[29] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient Parallel I/O Tracing and Analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2020, pp. 1–8.

[30] T. Li, S. Byna, Q. Koziol, H. Tang, J. L. Bez, and Q. Kang, "h5bench: a Parallel I/O Benchmark Suite for HDF5," Apr. 2023, original-date: 2020-10-28T18:31:02Z. [Online]. Available: https://github.com/hpc-io/h5bench

[31] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in High-Performance Computing Applications," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2068–2083, Nov. 2018, conference Name: Proceedings of the IEEE.

[32] S. Robert, S. Zertal, and G. Goret, "SHAMan: an intelligent framework for HPC auto-tuning of I/O accelerators," in *Proceedings of the 13th International Conference on Intelligent Systems: Theories and Applications*, ser. SITA'20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3419604.3419775

[33] Y. Jia and F. Chen, "Kill Two Birds with One Stone: Auto-tuning RocksDB for High Bandwidth and Low Latency," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, Nov. 2020, pp. 652–664, iSSN: 2575-8411.

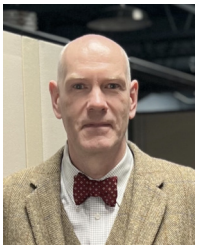
[34] X. Wang, "A light weighted semi-automatically I/O-tuning solution for engineering applications," 2017. [Online]. Available: https://shorturl.at/e9uPZ



Hariharan Devarajan is a Computer Scientist in the Parallel Systems Group in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). He received his Ph.D. in Computer Science at the Illinois Institute of Technology, advised by Dr. Xian-He Sun. His research is focused on Parallel and Distributed Systems, Parallel I/O optimizations, HPC systems, and Key-Value Store solutions.



Kathryn Mohror is a computer scientist in the Parallel Systems Group in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). Kathryn serves as the Deputy Director for the Laboratory Directed Research & Development (LDRD) program at LLNL, Lead for the NNSA Software Technologies Portfolio for the U.S. Exascale Computing Project (ECP), and as the ASCR Point of Contact for Computer Science at LLNL. Kathryn's research on high-end computing systems is currently focused on I/O for extreme scale systems. Her other research interests include scalable performance analysis and tuning, fault tolerance, and parallel programming paradigms.



Gerd Heber is the Executive Director of The HDF Group, a non-profit whose mission is to ensure efficient and equitable access to science and engineering data across platforms and environments, now and forever. He received his Ph.D. and M.S. in Mathematics at the University of Greifswald, advised by Dr. Jürgen Eichhorn. His research is focused on scientific computing and the modeling and effective and efficient life cycle management of large-scale scientific and engineering data across platforms, from mobile and embedded devices to supercomputers and cloud environments.