

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-839070

Mimir: Extending I/O interfaces for expressing \\User Intent for Complex Workloads in HPC.

H. Devarajan, K. Mohror

August 22, 2022

37th IEEE International Parallel & Distributed Processing Symposium
St. Petersburg, FL, United States
May 15, 2023 through May 19, 2023

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Mimir: Extending I/O Interfaces to Express User Intent for Complex Workloads in HPC

Hariharan Devarajan, Kathryn Mohror
Lawrence Livermore National Laboratory
hariharandev1@llnl.gov, kathryn@llnl.gov

***Abstract*—The complexity of data management in HPC systems stems from the diversity in I/O behavior exhibited by new workloads, multistage workflows, and the presence of multitiered storage systems. This complexity is managed by the storage systems, which provide user-level configurations to allow the tuning of workload I/O within the system. However, these configurations are difficult to set by users who lack expertise in I/O subsystems. We propose a paradigm change in which users specify the intent of I/O operations and storage systems automatically set various configurations based on the supplied intent. To this end, we developed the Mimir infrastructure to assist users in passing I/O intent to the underlying storage system. We demonstrate several use cases that map user-defined intents to storage configurations that lead to optimized I/O. In this study, we make three observations. First, I/O intents should be applied to each level of the I/O storage stack, from HDF5 to MPI-IO to POSIX, and integrated using lightweight adaptors in the existing stack. Second, the Mimir infrastructure supports up to 400M Ops/sec throughput of intents in the system, with a low memory overhead of 6.85KB per node. Third, intents assist in configuring a hierarchical cache to preload I/O, buffer in a node-local device, and store data in a global cache to optimize I/O workloads by 2.33×, 4×, and 2.1×, respectively. Our Mimir infrastructure optimizes complex large-scale workflows by up to 4× better I/O performance on the Lassen supercomputer by using automatically derived I/O intents.**

***Index Terms*—Mimir, user intent, I/O extension, multi-stage intent, Intent Infrastructure.**

I. Introduction

Modern HPC workloads that generate, process, and store large amounts of data [1] are known as data-intensive workloads. These workloads consist of individual applications [2], such as simulation, analysis, and artificial intelligence (AI) applications, and collections of applications that execute cooperatively in a workflow [3], such as simulations coupled with big data or AI analytics. Additionally, the presence of new storage hardware devices, where additional levels of memory and storage are stacked in a hierarchy, provides users low latency and high throughput. This diversity in workloads and storage hardware has led to the high configurability of storage software to cater to the different requirements of individual workloads [4]. While setting workload-specific configurations for optimizing I/O can yield significant performance benefits, determining which parameters to set and to what values is a complex task [5]. As most HPC storage-system software utilizes POSIX as its default I/O interface for workloads, some middleware libraries [6], [7] use the intent API (i.e., `fadvice`) provided by POSIX to define I/O behavior and internally configure their system. A user's I/O intent is defined as "why" and "how" the user performs I/O operations. The "why" explains the reason or domain logic behind performing I/O, whereas "how" gives resulting I/O characteristics. In the context of the POSIX interface, the `fadvice` API supports a few common OS file-system-specific user intents [8] such as sequential data access, random data access, the intention of data to be accessed in the near future, and data no longer needed. These intents are used within the kernel file system implementation to manage OS cache efficiently [9].

Describing I/O intent for HPC workloads is a complex task due to the diversification of HPC workloads, multistage workflows, and multi-tiered hierarchical storage. As `fadvice` was designed for a single process accessing a local file system, it is insufficient in the context of distributed workloads running on large-scale storage systems for three main reasons. First, HPC workloads have diversified from traditional simulations to data-analytic and artificial-intelligence workloads [10]. This diversification has led to new I/O behavior that must be supported by HPC storage systems [5]. Second, we are seeing the emergence of workflows with multiple stages that can include workloads with differing I/O behavior [3]. The presence of workflows in HPC systems leads to changing I/O intent during workflow execution. For instance, a piece of data might be written by only one application and read-only by another. These changes in intents (the "how" part) need to be communicated to the storage system at runtime in a distributed fashion. Finally, modern HPC environments contain multiple storage systems with different characteristics [6]. These systems are often selected based on the characteristics of I/O behavior within the workload.

The complexity of modern HPC workloads and storage systems dictates the need for a new intent-management layer that can help users define I/O intent in complex environments. To reduce the complexity of defining I/O intent, we propose a new set of intent APIs, called Mimir API, to systematically extend the existing I/O interfaces used in HPC systems. The main philosophy behind Mimir API is to provide context-driven APIs at each layer of the software stack. These APIs are mostly hidden from the end user and integrated into the software stack by I/O library developers. This strategy enables users to transparently pass I/O intent at the correct software layer instead of defining them together with no contextual organization. If we do not segregate the APIs we could create an explosion of intent APIs, which would be harder for the

system to manage. For instance, we define parallel I/O-based intent only at the MPI-IO layer and dataset-based intent for HDF5 only. Therefore, if a workload uses HDF5 with MPI-IO, both intents are defined at an appropriate level and passed to the underlying storage system.

We developed the Mimir API to capture user I/O intents at different workload levels and communicate those intents to the underlying storage system. To make APIs more practical, we provide the following features for the three main stakeholders of the library: first, for HPC users, a collection of tools to extract user I/O intent automatically; second, for HPC storage software, Mimir infrastructure and adaptors to cohesively interface with the existing software stack and capture intents with low overhead and high throughput; third, for storage-system designers, a collection of use cases that illustrate the use of intent within modern storage systems. To demonstrate the benefits and applicability of the Mimir API, we built a hierarchical cache prototype that uses I/O intent to perform I/O optimizations for the workload. Finally, the evaluations demonstrate that the Mimir infrastructure can achieve a high throughput of 400M ops/s for managing intents and accelerate Pegasus [11] workflows by up to 5× better performance. The key contributions of this work are

1) *Design of Mimir API* for capturing intents for different I/O interfaces, job configurations, software stacks, and data used in HPC workloads;
2) *Design of Mimir infrastructure* for efficiently defining, passing, and managing intents for HPC workloads; and
3) *Demonstration through use cases* of how I/O intents can benefit modern storage systems.

II. Background and Related Work

In the last decade, data-intensive HPC workloads have significantly diversified from applications in cloud environments [10]. The primary examples of these workloads are high-performance data analytics and AI. These workloads often support traditional simulations to increase the accuracy or reduce the cost of modeling within the simulation workloads [12]. This diversification of workloads in HPC environments has resulted in diversification of user I/O intent and highly configurable storage-software stacks.

A. Diversity of User Intent for I/O

Modern high-performance data-analytics workloads perform embarrassingly parallel algorithms to access and perform data transformations [13]. We demonstrate user intent for some of the popular workloads in HPC from analytics, artificial intelligence, and multistage workflows. More than the "why" (which provides contextual information), we focus on the "how", as it describes I/O behavior.

The common I/O intent of these workloads is that they iteratively traverse the data to converge on a solution for their respective analytical algorithms. A key difference in the I/O intent of these workloads is that the data they consume are accessed with different I/O semantics. For example, BD-CATS [14] operates on a shared HDF5 file using MPI-IO semantics; KMeans [15] reads points from files, often divided by the number of tasks in the algorithm; and linear-regression [16] models often consume large CSV data of floating-point values. AI workloads consume different kinds of large HDF5 datasets. In terms of I/O intent (the "why" using "how"), CosmoFlow [17] wants to consume multiple HDF5 files using MPI-IO semantics, as each file represents a sample size of 32 MB. By contrast, Cosmic Tagger with UNet (Cosmic Tagger) [18] and distributed flood-filling networks (DFFNs) [19] want to consume large HDF5 files using h5py APIs and stripe the file in memory. Finally, we look at large-scale workflows such as Multiscale Machine-Learned Modeling Infrastructure [20] (MuMMI), Montage [11], and 1000 Genomes [21] workflows. These contain multistep applications with a collection of simulation, analysis, and AI stages. These workflows contain several unique I/O intents. First, the intent of I/O changes as the workflow goes through different stages. During the simulation stage of MuMMI, for instance, the application generates checkpoints using write-heavy I/O operations; whereas during the AI stage, the workload trains the AI model on the simulation data over several epochs using an iterative read pattern.

As shown above, the diversity of workloads diversifies I/O intent and makes it more dynamic during a workload's lifetime. This diverse I/O intent dictates the need for a new intent infrastructure that transmits user intent to storage to enable workload-aware I/O optimizations.

B. Configurability of HPC storage-software stacks

The storage-software stack provides user-level configurations to support workload diversity and often directly affects the performance of workloads on HPC systems [12]. This section explores user-level configurations present in parallel file systems (PFS) and middleware-I/O libraries.

In the case of PFS, each software solution provides a configuration based on its architecture. For example, a) GPFS allows users to configure the ROMIO driver to optimize for independent data accesses [22]; b) the Intel MPI-IO enables applications to select the number of aggregators and limit the number of processes for collective buffering [23]; and c) the DAOS storage system enables users to configure multiple tiers of storage and page-cache size [24], [25]. In the case of middleware-I/O libraries, each library provides optimization-specific parameters for users to set to optimize I/O for their workload—for example, buffering I/O middleware software [6], [7] allows configuration of parameters such as the buffer size of tiered buffering resources, placement policy, element eviction policies, etc. The main drawback of this configurability is that, currently, users have to manually configure the storage system for their workload. Some recent autotuning work [4], [26] aims to explore this configuration space based on extracted behavior. These works do not use a standard set of characteristics as proposed by Mimir APIs. In this context, we propose a change in the paradigm, so that users define their I/O intent (the "how" part) and storage systems configure themselves (even with autotuning) according to the intent

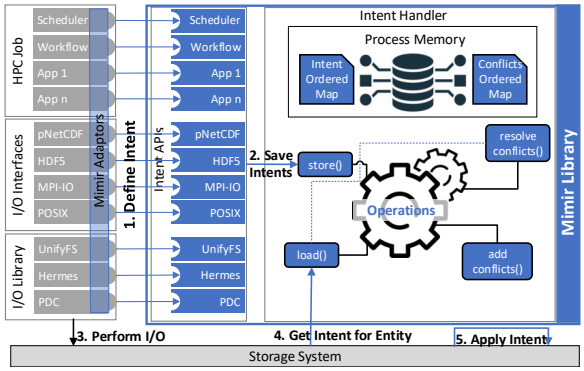


Figure 1: At the core of the high-level view of the system, we enable intent capturing and storing using the Mimir library that defines a collection of APIs, along with a management layer for user intent. We define a collection of adaptors to enable transparent integration with existing software for user intent.

provided. This would complement many middleware solutions, such as ADIOS XML and Hermes Configurator, as they can now define intents that will be converted into configurations as required by these libraries.

III. Mimir

Enabling user intent within the HPC software stack requires software support at all layers of the I/O stack. To capture these user intents, we present the Mimir library to define APIs and management routines within the system. To automate the capture of intent for user convenience, we extend the Vani tool suite [5] by offering a collection of tools for extraction of I/O intent from user workloads. Additionally, for cohesive integration of user intent within existing system software, we propose a collection of Mimir adaptors that extend existing I/O interfaces and software to accommodate user intent without additional software dependencies. Finally, we demonstrate a collection of use cases to assist system designers in enabling user intent as part of the system architecture. Through this work, we envision a transformation in the modern HPC software stack such that system software focuses on both I/O interfaces and user intent to enable workload-aware optimizations for generalized storage-system software.

At a high level, the Mimir library is a system-level software that consists of core intent-based APIs that capture user intent in the system. This system view is depicted in Figure 1. The Mimir API is defined for different entities in an HPC system. Mimir API is defined for each layer of the software stack in the context of a job running in an HPC system (Figure 1). An entity is defined as hardware, software, and data component in the system. At the scheduler level, we define the API to capture allocation information such as the number of nodes, memory per node, CPU/GPU cores per node, mounted storage systems, and runtime bound for the job. Similarly, we define the API for software elements such as workflow managers (e.g., Pegasus [11] and MaestroWF [27]), I/O interfaces (e.g., HDF5 [28], MPI-IO [29], and POSIX), and even middleware I/O libraries (e.g., UnifyFS [30], Hermes [6], and SCR [31]). By segregating APIs for each software layer, we achieve the principles of API modularity and separation of concern for our system design for user intent. Internally, the Mimir library contains a collection of management routines and efficient data structures to simplify the adoption of the intent-based ideology for storage-system software. Specifically, the library handles the ingestion and searching of user intents within the system, and the system software queries the library for applying intent in real time. The library is designed to share the runtime with the workload's executable and the storage-system client. This design enables the user to transparently pass intents to the storage-system client.

As an example, intent is defined by the user and consumed by the Mimir library through the APIs (Figure 1, arrow 1.). The intent is saved in process memory through efficient data structures using the library IntentHandler routine called `store()` (Figure 1, arrow 2.). The user then performs I/O through system interfaces such as POSIX, which is captured by the storage system (Figure 1, arrow 3.). The storage system, in turn, queries the Mimir library for attached intent for an I/O entity (Figure 1, arrow 4.). Examples of I/O entities are files, objects, and write/read operations. The library retrieves all intents attached to an I/O entity after resolving any storage-specific conflicts. The storage system uses the resolved set of intents and applies storage-system-specific optimizations or configurations for the workload (Figure 1, arrow 5.).

Users have multiple mechanisms through which they can define I/O intent in the workloads. At the lowest level, workloads can link to the Mimir library directly and pass user intent with the application code. This linking is extremely flexible, as users can configure different scopes of intent, set priorities, and dynamically change I/O intent during an application's runtime. However, for existing applications, we recommend our Vani tool suite to extract workload intent dynamically and pass it to the software stack using the LD_PRELOAD mechanism. With this mechanism, users need not change any application code, but they will have to profile their application and extract the I/O intents using our tools. The intents will be applied by the storage system globally, without fine-grained control over the intent lifetime. For easy deployment as system software, we provide a collection of adaptors that can be integrated within existing I/O interfaces such as HDF5. Using the adaptors, users need not include the Mimir library explicitly. These three mechanisms cover many use cases that HPC users can employ to express I/O intent to the underlying storage system. Each approach has its merits; the selection would depend on user preference or the HPC system's deployment.

A. Mimir API

Mimir API is the core interface for defining I/O intent within the system. We argue that distributed systems need a richer semantics to express I/O intents than provided by the `fadvice` API in POSIX. For performance and portability across different types of workloads, the API is natively compiled as a shared library written in C++ with extensions to C, Python, and bash programming languages. The API is a facade that passes intents to `IntentHandler`.

```
1. int main(int argc, char* argv[]) {
2.      /* init program structures*/
3.      ...
4.      using namespace mimir;
5.      MimirHandler i_file_h1, i_file_h2;
6.      FileIntent fi1(name="file.dat", type = INDEPENDENT_FILE,
                       size_mb = 4.0, current_device = PFS,
                       io_amount_mb = 4.0, write_dist._0_4KB = 1.0);
7.     assert(file_begin(fi1, i_file_h1) == MIMIR_SUCCESS);
8.     int fd = open("file.dat", O_WRONLY | O_CREAT, 0660);
9.     for(int i = 0; i < 1024; ++i) {
10.         assert(write(fd, write_buf, 4096) == 4096);
11.    }
12.    assert(close(fd) == 0);
13.    /* Computations */
14.    ...
15.    FileIntent fi2(name="file.dat", type = READ_ONLY,
                      size_mb = 1024.0, current_device = PFS,
                      io_amount_mb = 512.0, write_dist._1_4MB = 1.0);
16.    assert(file_begin(fi2, i_file_h2) == MIMIR_SUCCESS);
17.    int fd = open("file.dat", O_RDONLY);
18.    for(int i = 0; i < 4; ++i) {
19.        assert(read(fd, read_buf, 1048576) == 1048576);
20.    }
21.    assert(close(fd) == 0);
22.    assert(file_end(i_file_h1) == MIMIR_SUCCESS);
23.    assert(file_end(i_file_h2) == MIMIR_SUCCESS);
24. }
```

Figure 2: In this example for managing the intent scope, lines 8–10 see only file intent `fi1`, whereas lines 17–21 see both `fi2` and `fi1` with `fi2` at a higher priority than `fi1`.

1) API Semantics: In the library, Mimir Intent APIs are defined under the `mimir` namespace to prevent conflicts in APIs and easy extension to different languages. The APIs are designed with epoch semantics to enable I/O intent at different scopes of the application code. Here, each intent is active within an epoch—that is, each intent has a `begin`: marking the application of intent and an `end`: marking the removal of intent within the system. For instance, we can apply file-level intent to the whole application or just a couple of operations. This way, users have complete flexibility to instruct the underlying storage system. The user's intent is defined based on the entity of the system. If the user uses the LD_PRELOAD mechanism to load intents, they are applied at the start of each application load and removed when the application exits. This is achieved using GCC constructor and destructor for the Mimir library. All intent structures are passed/maintained within the system as a reference to ensure a minimal memory footprint of intents. Each API call returns a handler that maintains the state for the user to end the epoch at a later time.

2) Scope/Visibility: User intents are visible local to the process and are stored in process-heap memory (to minimize footprint) making it globally accessible by all threads of the process. `IntentHandler` maintains the order of intent in a last-in, first-out manner (implemented using a stack data structure) to efficiently maintain scopes of different intents in the system. We discuss an example of POSIX code (Figure 2) where the application opens a file, writes the data, and then reads the data back with a different granularity of I/O. The user marks the file access as *Independent* using lines 6–7. Therefore, all I/O operations on *file.dat* see the file intent `fi1` and the storage system knows that the user is performing I/O with

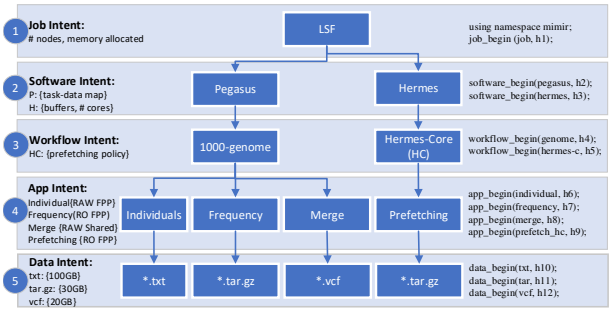


Figure 3: At each level of the software stack, we capture the user's I/O intents and compound them for the storage system. Generally, closer to the applications, users have more high-level logic in their applications, whereas lower down on the stack this information becomes closer to the bytes written/read.

4KB granularity and the total amount of I/O. The application subsequently reads the file back at a later point. Here, the user defines an intention to only read the complete file with 1MB granularity (line 15–16 file intent `fi2`). Here, all operations performed between lines 17–21 see the file intent `fi2` first and then `fi1` next. If the storage system that implements these file intents marks that it has a conflict between read-only intents and independent intents, then `IntentHandler` will prioritize `fi2` over `fi1`, as it has closer scope toward the operations in lines 17–21. When the file system sees `fi2`, it can apply prefetching optimization with correct stride and count (1MB and 1024 respectively) to match the reading by the application. Finally, the user can define a custom priority of intents in the system for use in the conflict-resolution algorithm.

3) Multilevel example of using Mimir API: The Mimir API, defined for each software-stack level, can be combined into a comprehensive intent-configuration file. All intents are applied in the scope of a job allocated in an HPC system. We present an illustrative example of a multistep workflow running on the HPC system (Figure 3). At the highest level, we have the resources allocated by a scheduler such as Slurm [32] or Flux [33]. Here, the intent APIs capture the job's intent, such as number of allocated nodes, amount of memory allocated per node, number of GPUs allocated per node, number of CPUs allocated per node, type of network (IB/TCP), node-local burst buffer (mount point, controller, and block size), parallel file system (mount point, stripe size, and stripe count), and total job time. Once the job is allocated, the workload executes the Pegasus workflow manager (PWM) [34] with Hermes [6] (using LD_PRELOAD) at level 2. Here we can capture intent for software such as PWM and Hermes (e.g., PWM:task-data map describing the dependency of task and dataset, job-aggregator describing the job which can act as process aggregators in the workflow, data-configuration, num_tasks deployed on the system, num_tasks_per_node deployed on the system, and scratch dir, Hermes:number of buffers, buffer capacities, block size, block count, cores per node required, and RPC threads). Similarly, at level 3, where the workflow executes, we can capture domain-specific information from the 1000-genome workflow, such as the length of genome sequences accessed, number of

samples, dataset size, number of populations to be sequenced, and population size. This domain-specific information is one of the instances of high-level intents defined by the user. This information can be utilized in the storage system to predict the amount of I/O expected, format to be utilized to store this information based on data type [35], or compression algorithm to be used based on data type [7]. At level 4, we have individual applications executed over the allocated resources. Each application has its I/O pattern with a file at level 5. For instance, the `Individual` app accesses a collection of text files with a read-after-write behavior, file-per-process pattern, small 4KB transfer size, and POSIX interface. Also, the Hermes core employs prefetching threads that load all tar files required by the node with large transfer sizes of 1MB shared across all Hermes cores. At each level, we have a utility (executable or library) that automatically extracts or captures through APIs the user intents and builds a JSON file with them. The storage system loads this JSON file during client initialization.

In this example, we have high-level I/O intents that need to be passed by the user and others that can be extracted using I/O calls directly. The I/O intents that can be extracted directly are extremely useful in making this work practical. However, the user or system must pass high-level intents using Mimir APIs. Examples of high-level intents are scheduler job intents that are provided in the job script, such as the number of nodes, and memory allocated, or by the batch scheduler, such as node-local burst buffer mount point, parallel file system mount point, and network characteristics of compute nodes. These intents are vital for storage systems to determine the optimization parameters such as buffer capacities, number of inter-node threads for handling workload scale, and the presence of node-local devices for increasing buffering resources. Additionally, this information can be used by middleware I/O libraries such as UnifyFS and Hermes to determine the number of cores per node needed for handling workload scale and allocated node-local devices. Once these middleware libraries use this information, they pass on their allocated information as I/O intents to the storage system, which can then look at all the intents to calculate the overall resources allocated by them. Similarly, intents from PWM, such as task-data relationships, are very hard to capture directly from I/O calls [36] but are vital to capture producer-consumer patterns so that the storage system can employ caching optimizations.

B. Intent infrastructure

Within the Mimir library, `IntentHandler` is the critical component for intent infrastructure and management. `IntentHandler` isolates different intents within the system based on entity type. Within POSIX, intents are defined as integers with no other information. This is insufficient as we saw in the Mimir APIs. The `IntentHandler` manages a global list of intents with many attributes within a singleton instance, local to the process. Its algorithmic design prioritizes retrieval operations over store operations, as we expect users to define/undefine intent once, but the storage system will retrieve them multiple times, depending on the process/thread. At a high level, `IntentHandler` has two primary responsibilities: a) *store and find intents*—a fundamental task of storing intents from users and retrieving them for the storage system, and b) *conflict management for intents*—a particular task that storage systems can use to identify intent conflicts before applying them. Essentially, conflicts for each entity and intent type are added during the startup of the storage system and resolved at runtime based on user intents. For each of these responsibilities, we next discuss the internal data structure and algorithm employed for efficient management of intents within the system.

1) Storing and finding intents: Intents are stored in memory using an unordered_map of entity to an ordered set of intents (i.e., `unordered_map<EntityKey, set<Intent>*>`). As the intents are global within a process, inter-thread write access requires thread-safe implementation of the data structures. By default, this locking is disabled within the library and can be enabled based on client access patterns. The complexity of store operation within the handler is $O(log(N))$, where N is the number of intents on a given entity. This is because of the insert operation stores the intents in decreasing order of priority. However, the complexity of the find operation within the handler is $O(1)$, due to the `unordered_map` lookup.

2) Storing and resolving intent conflicts: Conflicts between intents are defined once by the storage system based on the implementation of intents within the system. These conflicts are stored in memory using `unordered_map` of intent to an ordered set of intents. As in the management of intents, conflicts are stored globally within a process with similar performance characteristics. The complexity of adding conflicts within the handler is $O(log(N))$, where N is the number of conflicts on a given intent due to insert operation. By contrast, the complexity of resolving conflicts is $O(N)$, where N is the number of conflicts on a given intent, based on iteration over all the conflicts in a given intent.

C. Mimir adaptors

Mimir adaptors are transparent mechanisms to integrate intent within the existing software stack, enabling existing I/O software like HDF5 or MPI-IO to expose intent-based APIs that allow users to interact with only one interface (i.e., their traditional I/O interface) and use Mimir implicitly behind the scenes. Mimir acts as a middleman to ease communication between the user and the storage system. *Mimir does not apply the intent provided by the user, but only passes it efficiently to the underlying storage system.* We provide a working example of the flow of data and intent within a system using Mimir adaptors (Figure 4). The application creates a new HDF5 file and passes a file-access property called `H5P_FPP` for file-per-process (FPP) intent. This property is created as a part of the Mimir adaptor integration within HDF5. Internally, the Mimir adaptor is implemented as an HDF5 virtual object layer (VOL) which applies/removes intents and calls the native HDF5 call (HDF5 in the figure). The VOL first stores the intent on the Mimir library using the Mimir API and then invokes storage-system APIs to create the HDF5 file. On

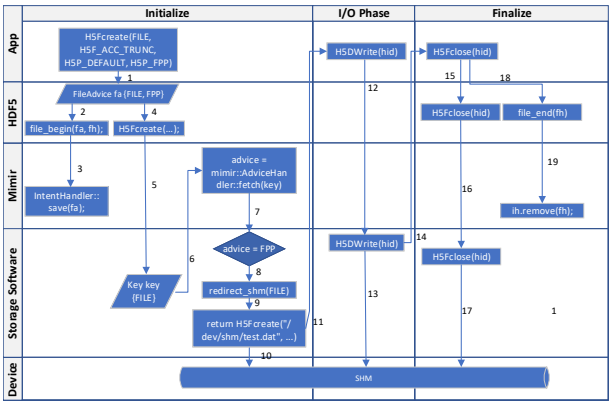


Figure 4: The application, through HDF5 Adaptor, transparently performs both I/O and intent operation (using Adaptor) in its VOL layer. The storage system queries the Mimir library in a decoupled way and applies intent-based optimizations.

receiving file creation requests, the storage system queries the Mimir library to check if the intent is present in the system. As the system finds the FPP intent, it redirects the I/O to shared memory (arrow 10) and returns to the application. The application now performs a write operation on the file id returned from the HDF5 library, which is automatically redirected to shared memory (arrow 13). Once the application finishes the I/O, it calls H5Fclose API, which internally ends the intent epoch for the file using the Mimir API (arrow 19). This example demonstrates how a simple FPP intent from the application can result in I/O optimization using shared memory for a file within the storage system.

D. Automatic extraction of intents

User intents (especially the "how") can be automatically extracted using an extension to the Vani tool suite [5]. Previously, the Vani tool focused on characterizing I/O behavior in workflows. In this work, we extended Vani to extract entity-level I/O intent and include other entities in the system, such as the job, software stack, and data. This extension to Vani enables capturing the behaviors of the batch scheduler, different scopes of the workflow (application, processes, threads, and I/O phase), middleware libraries, high-level I/O libraries, and storage system software such as burst buffers and parallel file systems, and finally, data itself as a whole or individual file/object characteristics. We aggregate the characteristics of different entities into simple high-level I/O intents such as FPP patterns, collective I/O patterns, temporary file access patterns, and stage-in/stage-out patterns. The extended suite consists of `JobUtility` and `Analyzer` tools that extract information from the workloads and build a JSON file of user intents. We attach a Recorder profiler [37] during the workload execution, which traces the workload for CPU function calls, GPU function calls, MPI communication calls, and multilevel I/O information. After workload execution, we run our `JobUtility` tool to extract job intent (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, format of files in the dataset, and size of each file). Recorder and utility logs are collected and stored in a global file system such as GPFS. After job execution, we use the `Analyzer` in the tool suite to convert the Recorder logs into Parquet format. This is necessary because Recorder logs are in row-major format, and filtering and aggregation operations in memory are highly inefficient for this format. The parquet file stores the trace data in a column-major format that can be processed using the DASK tool in an out-of-core fashion [38]. Also, the DASK library is used within the `Analyzer` to extract the workload's user intent and generate a JSON file of all user intents.

E. Use cases

Through Mimir, we want to encourage system designers to think about I/O in the context of the captured user I/O intent. We present several theoretical use cases to help system designers build I/O and intent APIs cohesively in the system. In these use cases, we commonly ask users to define "intent" (i.e., how they perform I/O) and not "storage configuration" (i.e., how they should configure storage for their I/O). In all cases, we observe that we can infer configuration based on the intent supplied by the user. To cover a wide range of system designers working on different software-stack layers, we present use cases of software examples from high-level I/O libraries, low-level I/O interfaces, and state-of-the-art middleware I/O libraries. These use cases may help guide designers to extend existing I/O software to incorporate user intent cohesively.

1) High-level I/O libraries (MPI-IO and HDF5): For MPI-IO, user intent such as read-only file access, collective access-pattern information (aggregators, granularity, and strides), processes participating in collective I/O, the synchronicity of operations, buffers allocated, aggregators assigned, data properties (types, memory layout, serializations, etc.), and transfer size distribution information can help the library define parallel-I/O semantics efficiently. For instance, MPI-IO can disable file-locking semantics internally [22] when users specify access to read-only files. With this optimization, the storage system doesn't maintain expensive locks for cases with no conflicts. As another example, the MPI-IO library can optimize GPFS based on collective access-pattern information by auto-configuring GPFS for data-sieving optimization (i.e., `romio_ds_write` or `romio_ds_read`). This optimization can reduce random access to the file by reading large continuous chunks and discarding unwanted data in memory. The storage system can control the tradeoff between the amount of access I/O and PFS bandwidth with access-pattern information from the user. For HDF5 [39], user intent such as memory layout of the dataset (row/column major), chunking information (size, cache, and stride), file access patterns (sequential/random, stride, and count), distribution of values in datasets (i.e., uniform/gamma/normal), storage layout (contiguous or chunked), and metadata ops vs. data ops can help the library define scientific I/O-level semantics for the storage system. For instance, HDF5 can perform data sieving and read-ahead optimizations on the file by using the memory-layout information of the dataset [40]. This information will enable the whole data to be brought

proactively by the storage system, as it knows that HDF5 will perform all the I/O defined by the hyperslab [41]. Another example is the HDF5 library, which can configure Lustre FS based on chunking information [40]. Additionally, chunking granularity can define cache line size by optimizing libraries such as Hermes to build an efficient cache for HDF5 files. Finally, information about user query behavior vs. data operations can enable the restructuring of the HDF5 file effectively—for example, by splitting the HDF5 metadata and data and cache metadata information to optimize its access.

For HDF5 using MPI-IO-VFD can combine I/O intents from both layers for the storage system. For instance, collective I/O semantics from MPI-IO and chunking information from HDF5 can accurately inform the storage system about the data access pattern to be expected from the workload. This is a hard problem to solve as the I/O calls issued by high-level I/O libraries often do not match the I/O pattern seen by the storage system. Another example of using intents between layers is that if the HDF5 file was accessed in read-only mode, then MPI-IO can disable file-locking semantics internally. This will inform the storage system to use lock-free file access on the servers as the HDF5 file has only read-only file access.

2) Low-level I/O interface (POSIX): For POSIX, users can provide file-level intents such as a file-per-process file, producer-consumer file, file access patterns (sequential/random/stride and count), file sharing pattern (e.g., shared file pointer), metadata vs. data operations, and write-only file to describe the access intent of the file to the file system. For instance, if a file is accessed only by a single process (i.e., FPP), the POSIX interface can buffer the file in the local node to increase the bandwidth of the I/O operations. This intent implementation [6] requires the status and capacity of current buffers in the node, which the storage system can acquire from job configuration-level intent. Another example is that if a file is accessed in a shared mode by a collection of producers and consumers, we could utilize distributed cache such as Hermes [6], PDC [42], and UnifyFS [30] to employ node-local devices for caching and sharing using RDMA [43] communication semantics. Finally, in the case of write-only file access, we can perform asynchronous writes without the need for consistency checks and ensure write completion during the exit of the application only. This optimization is often used to overlap compute with I/O within the application, but the intent can enable this optimization for the storage system.

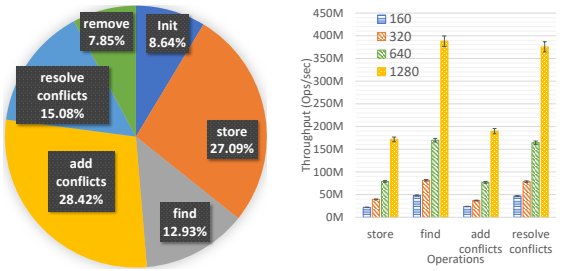
3) Middleware I/O libraries (Hermes and UnifyFS): For Hermes [6], users can provide the buffers types (NVRAM, NVMe, and memory) and the Hermes data-access pattern (non-mutatable, compaction, and data-migration between processes and nodes) to enable the file system to understand I/O transformations performed by Hermes. In the context of Hermes, the size and mount point of available buffers in the system can help Hermes pre-allocate buffers used for optimizing I/O operations for the workload. In the storage-system context, Hermes collects multiple application requests into larger buffers and flushes asynchronously to the parallel file system. This information on data-access patterns from Hermes can assist the storage system in correctly optimizing the workload. A similar use case exists for UnifyFS [30], in which users can define read/write patterns (e.g., node-local data sharing and read-only accesses), enabling UnifyFS to use the appropriate consistency semantics for I/O. UnifyFS shows that loosening consistency semantics, when possible, helps improve I/O performance for the workloads [44].

IV. Evaluation

To evaluate the benefit of managing user intent, we perform comprehensive experiments to test the internal capabilities of the Mimir infrastructure. We build a prototype hierarchical storage cache over a parallel file system (PFS) to demonstrate the potential impact of passing user intent to storage software. We implemented this prototype because no modern storage systems or middleware libraries support intent-based auto-configuration. In this evaluation, we use GPFS as the baseline PFS, then use our prototype hierarchical cache without Mimir (HC in figures), and finally our prototype enhanced with Mimir Intents (HC-M in figures). Using this hierarchical cache w/wo Mimir, we evaluate the benefit of passing I/O intents for different workloads using our micro-benchmark. Finally, to demonstrate the use of the complete system under large-scale workflows, we evaluate Mimir with complex workflows such as Montage and 1000-genome with PWM on a large-scale HPC system. We run these tests five times. The variance in the data is noted in the figures.

1) Hardware: We run the workloads on the Lassen supercomputer at Lawrence Livermore National Laboratory (LLNL) [45]. 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 PiB IBM Spectrum Scale file system (also known as GPFS). Individual Lassen nodes consist of two IBM POWER9 CPUs (IBM AC922 servers) with 256GB of system memory and four Nvidia Volta V100 GPUs with 64GB of HBM2 memory.

2) Implementation of a hierarchical cache: The cache works as a middleware library with a client that attaches to the executing processes through LD_PRELOAD and a distributed server that is deployed along with the PWM's pegasus-mpi-cluster module. The cache unifies node-local devices using RPC over RDMA protocol employed by thallium-rpc. The cache supports multiple hierarchical storage accelerators such as memory, node-local burst buffers, and shared burst buffers. For Lassen, we use two storage accelerator layers (i.e. the shared memory (/dev/shm) and local tempfs (/tmp)), backed by the GPFS as the final flushing destination. The cache configures its placement policy, cache-eviction policy, and prefetching schemes based on the user intent passed using Mimir. Specifically, it supports the following four optimizations. First, *FPP user intent*: The hierarchical cache utilizes node-local devices. Second, *Producer/consumer (PC) / Shared-file user intent*: The hierarchical cache uses a shared cache layer over node-local devices with RPC communication. Third, *Read-only (RO) / Input File (IF) user intent*: The hierarchical cache preloads the files asynchronously into node-local devices at the process initialization. Finally, *Temporary*



(a) Anatomy (b) Performance

Figure 5: For the Mimir infrastructure prioritizing find operations over store operations results in 2× more performance for retrieval operations.

File (TF) user intent: The hierarchical cache evicts the file's cache line on the intent's epoch end. For evaluations, the cache consists of a client and a server. The server is loaded along with the *pegasus-mpi-cluster* job, which spawns and coordinates workflow processes in the workflow. By contrast, the client is loaded with the workflow processes that perform I/O.

3) Workloads: We use micro-benchmarks to showcase different performance metrics on the designed components. Additionally, we employ two scientific HPC workflows using PWM, Montage (a mosaics-building tool used in astrophysics), and 1000 genome (a workflow that identifies mutational overlaps to find potential disease-related mutations).

4) Tools: We use the Vani tool suite to automatically extract user intent from Pegasus workflows.The tool is used with both workflows to extract user intent automatically at multiple levels of the software stack associated with a job and is stored in a JSON. This JSON file is loaded during *pegasus-mpi-cluster* job initialization within the workflow. When the workflow processes perform I/O, our hierarchical cache captures I/O calls and redirects them to the cache server for applying user intent.

A. Performance of Mimir infrastrasture

The Mimir infrastructure is responsible for managing user intent within the system. It comprises standard routines that enable efficient exchange of intent between the user and storage system. To evaluate its effectiveness, we test the relative cost of operations within the system to observe the time complexity of routines within the infrastructure. Additionally, we measure the overall throughput of operations in the infrastructure to ensure it is not a bottleneck in modern HPC systems. In all these tests, we observe a memory footprint of 6.84 KB per node on the Lassen supercomputer with thousands of intents stored by the Mimir library.

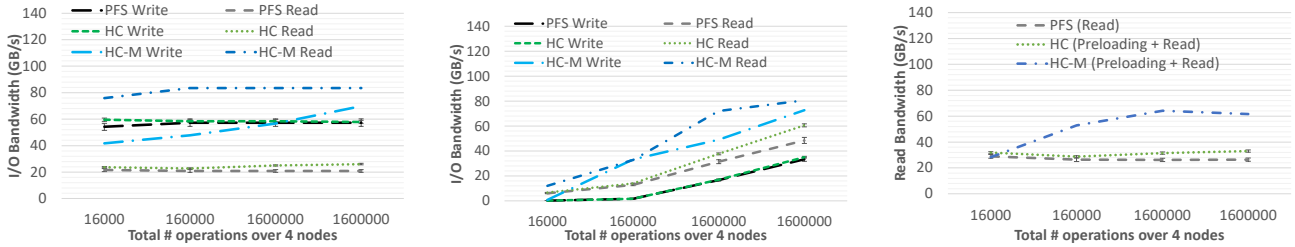
1) Anatomy of managing intents: To ensure that search operations are significantly more than store operations, we create a micro-benchmark that runs on 32 compute nodes with 40 processes on each node. Each process performs a million operations of every routine supported by the Mimir infrastructure. We calculate the relative cost of each operation on the system and plot a pie chart (shown in Figure 5(a)). We observe that initialization (*init* in the figure) is the least-expensive operation, due to the use of the singleton pattern for `IntentHandler`. Storing intents (`store()`) and adding conflicts (`add_conflicts()`) into the system has the most cost, because we store the intent in order of priority (with $Olog(N)$ complexity). The find operation (`find()`), which is the most frequently used in the system, has a low cost (2.25× faster than the `store()` operation). Finally, we see that the cost of resolving conflicts in the system is more because the complexity is $O(N)$. Here, we test a million conflicts, which is why the cost is high. For our real workflows, the N is in the tens scale which makes the cost of resolving conflicts close to `find()` operations.

2) Performance of intent operations: To ensure the Mimir infrastructure does not bottleneck modern storage systems, we create a micro-benchmark that measures the throughput of operations supported in Mimir infrastructures. We run our micro-benchmark on multiple nodes using weak scaling on 4, 8, 16, and 32 nodes with 40 processes per node to test this performance. The benchmark performs 10K operations per process, and we calculate the overall parallel throughput of the system. The results are shown in Figure 5(b). The x-axis depicts different workload scales for each operation. The y-axis shows the throughput in operations per second. We see that all operations scale well as we increase the number of processes, because intents are stored locally to the process. We also observe that the find- and resolve-conflicts operations achieve a maximum of 370-390M Ops/s and the store- and add-conflicts operations achieve 165M Ops/s. This throughput suggests that Mimir infrastructure will not become a bottleneck for storage systems and user applications, as I/O operations achieve throughput in thousands ops/sec for node-local burst buffers (the fastest storage accelerators).

B. Effects of intent-aware optimizations

HPC workloads contain different intents for each file/operation within the system. To understand the effects of optimizing I/O based on intents, our micro-benchmark isolates different cases of user intents and measures the effects of optimization on I/O calls. We demonstrate three main use cases prominent in HPC workloads: a) a file-per-process intent, in which every process performs a read-after-write access pattern on its own file, b) a producer-consumer intent, in which half the processes write data and the other half reads the data, and c) a read-only intent, in which every process reads an existing file from the parallel file system. For each case, the baseline runs the micro-benchmark over GPFS and the optimized case runs the micro-benchmark with our hierarchical cache, which has access to node-local shared memory and tempfs. In these tests, each process in the benchmark performs 100, 1K, 10K, and 100K I/O operations with a transfer size of 4KB with 160 processes (on four nodes). The x-axis shows the total number of operations performed; the y-axis shows aggregate I/O bandwidth achieved across all ranks in GB/s.

1) File-per-process intent: We observe a steady bandwidth of 56GB/s for writes and 21GB/s for reads from GPFS as we increase the number of ops (Figure 6(a)). This high bandwidth for writes is because OS caches the writes and asynchronously moves them to the PFS. However, as we do more I/O beyond



(a) File-per-process intent. (b) Producer/Consumer intent. (c) Read-only intent.

Figure 6: Passing I/O intents to HC can optimize I/O by 1.2×–4×. Additionally, the variance in I/O performance is lower than PFS because the HC uses node-local devices that are not affected by other jobs in the system.

the size of the OS cache, the write cache is eventually thrashed and therefore the reads suffer in performance. In the case of our hierarchical cache (HC), we notice the improvement in performance is about 10-15% when we do not use Mimir Intents. This is because GPFS performs aggressive caching already and therefore a custom cache with LRU evictions can maximize the usage of cache but is limited. However, when we pass I/O intents to our HC (HC-M), the cache now knows that this file will be read back, it maintains the cache till the read operation executes, rendering the speedup on reading operations much higher. Specifically, we can get up to 1.22× speedup on write operations and 4× on reading operations if we instruct the HC about independent files using Mimir.

2) Producer/consumer intent: We observe an increase in bandwidth of I/O operations for GPFS as we increase the number of operations (Figure 6(b)). Specifically, we see that on the GPFS, the write bandwidth increases from 160MB/s to 33GB/s and the read bandwidth increases from 6GB/s to 48GB/s. Again similar to the previous case, HC improves the I/O performance by 8-12% without having the knowledge of intents. This is because the consistent hashing algorithm within HC is not aware of which processes will consume the data. When we pass I/O intents to HC, with more operations, the benefit of a write cache helps the benchmark, and hence it gets to around 73GB/s for 100000 ops. The same is true for read operations. This write-read bandwidth is achieved owing to the locality-aware hashing algorithm used within the HC by using I/O intents (HC-M) to distribute data into the four nodes. Thus most operations are close to the reader process. The HC using PC intent can optimize write operations up to 2.3× and read operations by 1.34–2.1×.

3) Read-only intent: We observe a steady bandwidth of 27GB/s from the GPFS as we increase the number of operations (Figure 6(c)). For HC, the data is prefetched on the opening of the file. This enables 25% of the reads to be from memory. However, when we pass I/O intents, HC asynchronously prefetches the file in memory during application startup. This early prefetching allows 80% of the I/O operations to occur from memory directly. As we increase the operations on the file, the benefit of passing I/O intents on HC increases. This increase rises till the memory bandwidth saturates at around 63GB/s. The I/O intent can accelerate the optimization through HC by 2.33×.

In all these cases we use Vani tool suite to extract I/O intents from the benchmark run. Here, we tested passing incorrect intents to the benchmark. The side-effect of incorrect intents resulted in up to 20% performance degradation than the ideal case as described in these cases above. This is because even though the I/O intents are wrong, the HC still places them in a faster tier (though non-optimally). This results in improvement over GPFS but not as fast as it can be.

C. Complex multistage workflow performance

We evaluate the concept of user intent through Mimir infrastructure on complex HPC workflows. Specifically, we evaluate the Montage and 1000 genome with PWM and run over GPFS as the baseline. We trace the workflows and extract user-intent information in JSON format for the optimized version. In these tests, we measure the I/O cost per file, which is categorized into read-only, FPP, and producer/consumer files. Additionally, we include "other" to signify cases that do not fall under these optimizations.

1) Montage: The Montage mosaic engine is used to transform all images in the surveys to a common pixel scale of one second or arc, where all the pixels are coregistered on the sky and represented in galactic coordinates and Cartesian projection. The output of this workflow covers 10° of the sky along with the galactic plan, where each mosaic image is 5° by 5° in size and has an overlap of 1° with neighboring tiles. The workflow uses nine kernels of the mosaic engine in a complex workflow to build these patches of sky. We run the Montage workflow with Pegasus over GPFS.

In this case, we have extracted I/O intents from the PWM and individual applications. From the PWM, we extract the process-task dependency that provides a DAG, the number of tasks (6039 processes), tasks per node (40), data configuration (shared file system), and scratch dir location (/p/gpfs1). From the individual applications, we have information about process distribution between apps with maximum processes for the mDiff application (i.e. 5209 processes). Additionally, we have the distribution of I/O between apps, with mViewer producing the largest file of 1.5GB but based on the distribution of I/O time shows that the workflow spends 90% of I/O time in reading/writing intermediate files. Finally, we have low-level intents present for each file produced/accessed per process as a map to guide data placement with the HC. These I/O intents are utilized along with high-level I/O intents such as 7% of read-only files, 12% of independent writes, 70% shared access on files, scheduler resource allocation (32 nodes

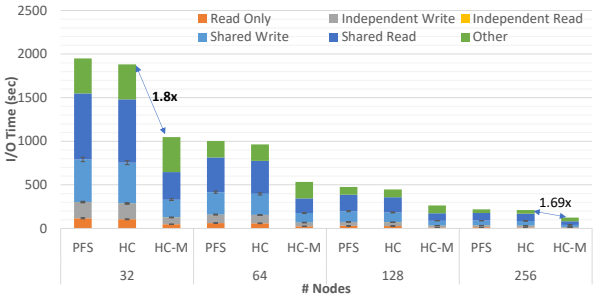


Figure 7: Montage workflow using Pegasus is optimized using the HC with intents by 2.7×-3× for producer/consumer, 2.2× for file-per-process, and 1.8× for read-only files.

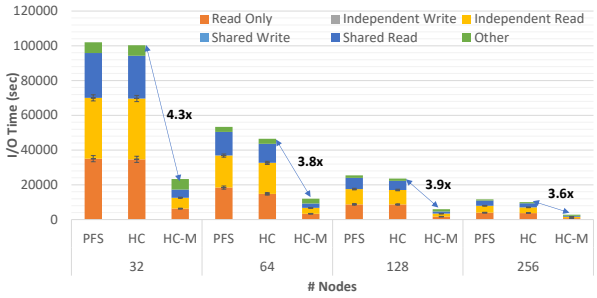


Figure 8: Genome workflow is optimized using the hierarchical cache by 2.7×–5× for producer/consumer files, 3.5× for file-per-process files, and 2.3× for read-only files.

with 40 process per node), and PWM's process-data map to optimize the workflow in our HC. Specifically, the HC combines the process-data map with file-level characteristics to determine which file should be placed on which storage system layer (shared memory, burst buffer, or GPFS).

We see performance improvement, because the amount of work is the same as we increase the number of nodes in both PFS and optimized case (Figure 7). The HC optimizes this workflow by 3-7% due to the better use of aggregate shared memory across all nodes. However, the performance of HC is limited due to cache thrashing as it is not aware of the workload's I/O intent. When we pass I/O intents to the HC (HC-M in the figure) we see that we can get better performance for the workflow. This is because HC using intents identifies how it should optimize each type of file within the system appropriately. Specifically, using I/O intents, HC optimizes 2.7×–3× on producer/consumer files, 2.2× on FPP files, and 1.8× on read-only files. Overall, the workflow achieves 1.8× better performance by passing I/O intents to HC.

2) 1000 Genomes: The 1000 Genomes with PWM project provides a reference for human variation, using the reconstructed genomes of 2,504 individuals across 26 different populations to energize these approaches. This workflow identifies mutational overlaps using data from the 1000 Genomes project to provide a null distribution for rigorous statistical evaluation of potential disease-related mutations. The workflow fetches, parses, and analyzes the project data, cross-matching the extracted data (which person has which mutations) with the mutation's sift score (how bad they the mutations are). It then performs a few analyses, including plotting.

In this case, we have extracted I/O intents from the PWM and individual applications. From the PWM, we extract the process-task dependency that provides a DAG, the number of tasks (1736 processes), tasks per node (40), data configuration (shared file system), and scratch dir location (/p/gpfs1). From the individual applications, we have information about process distribution between different applications: individuals-1632, frequency-62, mutation overlap-56, and others. Additionally, we have the distribution of I/O between apps: *individuals* perform 70TB and *frequency* perform 500GB of read/write operations. In this workflow, the read operations dominate the I/O (72TB of reads vs. 100GB of writes). Finally, we have low-level intents present for each file produced/accessed per process as a map to guide data placement with the HC. These I/O intents are utilized along with high-level I/O intents such as 34% of read-only files, 34% of independent reads, 32% shared access on files, scheduler resource allocation (32 nodes with 40 processes per node), and PWM's process-data map to optimize the workflow in our HC. Specifically, the HC combines the process-data map with file-level characteristics to determine which file should be placed on which storage system layer.

Our observations are similar to those in the Montage workflow, where we see improvement in performance because the amount of work is the same as we increase the number of nodes in both PFS and the HC (Figure 8). Similar to the previous case, the HC without I/O intents can only optimize the workload by 12%. However, the HC with I/O intents optimizes specific files based on user intent extracted and stored using Mimir infrastructure. The workflow is optimized using the cache by 2× for producer/consumer files, 3.5× for FPP files, and 2.3× for read-only files. Overall, the workflow with Mimir achieves 5× better performance over HC without Mimir.

V. Conclusion

In this work, we showcase the importance of I/O intents across different data paths of a workload by evaluating the hierarchical cache (HC) for a variety of workloads. We demonstrate that with I/O intents we can achieve better performance of 2.33× for read-only files, 4× for file-per-process files, and 2.1× for producer/consumer patterns as compared to an HC without I/O intents. Additionally, we demonstrate that the Mimir infrastructure can manage intents with a high throughput of 400 M Ops/s and low memory overhead of 7 KB/node. Finally, we showcase the impact of using I/O intents for complex or large-scale workflows such as Montage and 1000 Genomes using the Pegasus workflow manager resulting in up to 4× better I/O performance for the HPC workflows.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract LLNL-CONF-839070. 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.

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] 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.

[5] H. Devarajan and K. Mohror, "Extracting and characterizing I/O behavior of HPC workloads." Heidelberg, Germany: IEEE, Sep. 2022.

[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] 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.

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

[9] 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

[10] D. Milojicic, P. Faraboschi, N. Dube, and D. Roweth, "Future of HPC: Diversifying Heterogeneity," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, Feb. 2021, pp. 276–281.

[11] E. Deelman, G. Mehta, G. Singh, M.-H. Su, and K. Vahi, "Pegasus: Mapping Large-Scale Workflows to Distributed Resources," in *Workflows for e-Science: Scientific Workflows for Grids*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. London: Springer, 2007, pp. 376–394.

[12] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath, "DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2021, pp. 81–91.

[13] J. Kwon, N. L. Kim, M. Kang, and J. WonKim, "Design and Prototyping of Container-Enabled Cluster for High Performance Data Analytics," in *2019 International Conference on Information Networking (ICOIN)*, Jan. 2019, pp. 436–438, iSSN: 1976-7684.

[14] 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.

[15] A. S. Shirkhorshidi, S. Aghabozorgi, T. Y. Wah, and T. Herawan, "Big Data Clustering: A Review," in *Computational Science and Its Applications – ICCSA 2014*, ser. Lecture Notes in Computer Science, B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, Eds. Cham: Springer International Publishing, 2014, pp. 707–720.

[16] P. Xenopoulos, J. Daniel, M. Matheson, and S. Sukumar, "Big data analytics on HPC architectures: Performance and cost," in *2016 IEEE International Conference on Big Data*, Dec. 2016, pp. 2286–2295.

[17] S. A. Jacobs, B. Van Essen, D. Hysom, J.-S. Yeom, T. Moon, R. Anirudh, J. J. Thiagaranjan, S. Liu, P.-T. Bremer, J. Gaffney, T. Benson, P. Robinson, L. Peterson, and B. Spears, "Parallelizing Training of Deep Generative Models on Massive Scientific Datasets," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2019, pp. 1–10.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," 2016, pp. 265–283.

[20] J. Y. Moon, F. Di Natale, H. I. Ingolfsson, H. Bhatia, and J. R. Chavez, "Multiscale Machine-Learned Modeling Infrastructure RAS," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. MuMMI RAS, Sep. 2021. [Online]. Available: https://www.osti.gov/biblio/1835066

[21] N. Siva, "1000 genomes project," *Nature Biotechnology*, vol. 26, no. 3, pp. 256–257, Mar. 2008, publisher: Nature Publishing Group. [Online]. Available: https://github.com/pegasus-isi/1000genome-workflow

[22] F. Schmuck and R. Haskin, "{GPFS}: A Shared-Disk File System for Large Computing Clusters," 2002. [Online]. Available: https://www.usenix.org/conference/fast-02/gpfs-shared-disk-file-system-large-computing-clusters

[23] "ANL/MCS-TM-234 Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation," 2007.

[24] "Changing the DAOS tier 2 configuration." [Online]. Available: https://help.hcltechsw.com/domino/earlyaccess/admn_daos_t2_config_changes.html

[25] M. S. Breitenfeld, N. Fortner, J. Henderson, J. Soumagne, M. Chaarawi, J. Lombardi, and Q. Koziol, "DAOS for Extreme-scale Systems in Scientific Applications," *arXiv:1712.00423 [cs]*, Dec. 2017, arXiv: 1712.00423. [Online]. Available: http://arxiv.org/abs/1712.00423

[26] 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, Sep. 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3419604.3419775

[27] D. Natale and Francesco, "Maestro Workflow Conductor," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. MaestroWF; 005369MLTPL00, Jun. 2017. [Online]. Available: https://www.osti.gov/biblio/1372046

[28] P. Elena and K. Larry, "HDF5 Data Compression Demystified #2: Performance Tuning," May 2017. [Online]. Available: https://www.hdfgroup.org/2017/05/hdf5-data-compression-demystified-2-performance-tuning/

[29] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, ser. IOPADS '99. New York, NY, USA: Association for Computing Machinery, May 1999, pp. 23–32.

[30] 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

[31] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–11, iSSN: 2167-4337.

[32] "Slurm Workload Manager - burst_buffer.conf." [Online]. Available: https://slurm.schedmd.com/burst_buffer.conf.html

[33] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A Next-Generation Resource Management Framework for Large HPC Centers," in *2014 43rd International Conference on Parallel Processing Workshops*, Sep. 2014, pp. 9–17.

[34] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, May 2015.

[35] H. Devarajan, A. Kougkas, and X.-H. Sun, "An Intelligent, Adaptive, and Flexible Data Compression Framework," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2019, pp. 82–91.

[36] F. Chowdhury, F. Di Natale, A. Moody, K. Mohror, and W. Yu, "DFMan: A Graph-based Optimization of Dataflow Scheduling on High-Performance Computing Systems," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2022, pp. 368–378, iSSN: 1530-2075.

[37] 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.

[38] M. Dugré, V. Hayot-Sasson, and T. Glatard, "A Performance Comparison of Dask and Apache Spark for Data-Intensive Neuroimaging Pipelines," in *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, Nov. 2019, pp. 40–49.

[39] 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.

[40] Y. Ji, C. Yu, J. Xiao, S. Tang, H. Wang, and B. Zhang, "HDF5-Based I/O Optimization for Extragalactic HI Data Pipeline of FAST," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science, S. Wen, A. Zomaya, and L. T. Yang, Eds. Cham: Springer International Publishing, 2020, pp. 656–672.

[41] "Introduction to Parallel HDF5." [Online]. Available: https://confluence.hdfgroup.org/display/HDF5/Introduction+to+Parallel+HDF5

[42] S. Byna, Q. Koziol, V. Vishwanath, J. Soumagne, H. Tang, J. Mu, B. Dong, R. A. Warren, F. Tessier, T. Wang, and J. Liu, "Proactive Data Containers (PDC): An Object-centric Data Store for Large-scale Computing Systems," vol. 2018, pp. IN34B–09, Dec. 2018.

[43] "High Performance RDMA Protocols in HPC | SpringerLink." [Online]. Available: https://link.springer.com/chapter/10.1007/11846802_18

[44] C. Wang, K. Mohror, and M. Snir, "File System Semantics Requirements of HPC Applications," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 19–30. [Online]. Available: https://doi.org/10.1145/3431379.3460637

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