IFlux: Intent-aware storage tiers & software scheduling for HPC systems

Hariharan Devarajan, *Member, IEEE,* Vanessa Sochat, *Other,* Daniel Milroy, *Member, IEEE,* Thomas Scogland, *Member, IEEE*

Abstract—Modern High Performance Computing (HPC) workloads exchange datasets at high velocity to enable data-driven science. HPC systems employ hardware and software I/O accelerators to enable efficient data exchange. However, domain scientists and software engineers must manually request specific hardware accelerators and manually integrate software accelerators into their workloads, significantly hindering their adoption. Recently, scientists have proposed I/O intents, defined as "why", "what", and "how" users or applications perform certain I/O operations, that can be used to automatically configure storage systems for the user. We designed an intent-driven scheduler plugin called IFlux that uses the workload's I/O intents to assist the HPC system scheduler to automatically and dynamically allocate hardware and software accelerators. In this work, IFlux advances the current state-of-the-art in three key aspects. First, IFlux automatically and efficiently maps different I/O intents to various hardware and software accelerators with a throughput of 3.5M intents per second with a scheduler overhead of 0.02%. Second, IFlux provides a scheduler workflow that uses a scheduler's job specification enhanced with intents to allocate the required storage accelerators, deploy necessary middleware software, and enable workloads to use the allocated hardware and deployed software transparently with a runtime overhead of 0.01%. Finally, IFlux speeds up popular benchmarks such as the IOR and DLIO Benchmarks, which represent six classes of workloads from simulation, data analytics, and artificial intelligence, by up to 147× for specific use cases. In conclusion, IFlux enhances existing HPC schedulers with an intent-driven approach to speed up large-scale HPC workloads such as the 1000 Genomes workflow and Megatron Deepspeed by up to 3.12× on an institutional cluster.

Index Terms—intent-aware scheduling, I/O and compute co-scheduling, automatic scheduling, portable scheduling

1 Introduction

Modern scientific and engineering workloads, including traditional simulations, data analytics, and artificial intelligence (AI), utilize data-driven techniques to advance science by exchanging large amounts of data in a coordinated fashion. These data exchanges, with varied I/O requirements, often dominate workload runtimes and require special support to run efficiently on modern High-Performance Computing (HPC) systems. HPC systems support different I/O requirements from these workloads by providing requirement-specific hardware and software I/O accelerators to speed up the data exchange rate [1], [2]. These I/O accelerators have demonstrated that data exchange within scientific workloads can be accelerated by several folds [1], [3], [4] on large-scale HPC systems.

HPC systems provide two broad types of accelerators for optimizing the data exchange time within scientific workloads: hardware accelerators and software accelerators. Hardware accelerators [1] are near-compute, faster storage devices designed to provide low-latency and high-bandwidth access to the data. Typically, these storage devices are limited in capacity and are intended for temporary data that needs to be moved to and from the Parallel File System (PFS) for persistence [4]. On the other end of the spectrum, scientists propose software accelerator solutions [1], [3], [4] that utilize a portion of computing resources such as main memory and hardware accelerators to transparently optimize either data access within scientific workloads or data exchange within scientific workflows. These software accelerators are manually integrated within scientific workloads [1], [3]–[5] by running their data management service, linking the software to the workload, and configuring the software accelerator using environment variables. These accelerators have demonstrated that they can significantly improve their targeted data exchange behaviors on large-scale HPC systems.

System schedulers such as the Flux Framework [6] depend on users manually specifying the required hardware accelerators and utilizing the appropriate software for their workloads. This process is challenging for users due to three types of complexity. First, users must handle *workload complexity* where different parts of the workload require different I/O accelerators that can optimize their data exchange [7], [8]. Second, users must handle *software complexity* where every software accelerator is optimized for specific workload behaviors and needs to be carefully integrated for optimal performance [9], [10]. Finally, users must handle *hardware complexity* where every hardware accelerator is utilized appropriately with the correct interface for optimal workload performance [1], [11], [12]. Choosing the correct I/O accelerator for each part of the workload would require user expertise across multiple hardware and software accelerators, which is unrealistic to expect. Additionally, the users would have to manually adapt their code to new future systems as the availability of hardware and software accelerators across different systems changes. This complexity in achieving portability and optimal performance dictates the need for an automated solution that can enhance existing schedulers to intelligently deploy the required hardware accelerators, and the user application needs to support all the dynamically integrated

software accelerators for different parts of the workloads.

In recent years, I/O intent has emerged as a practical way to express workload behavior from the user's perspective [9]. Intents can be authored manually or extracted automatically. A *user intent* captures "why", "what", and "how" of an I/O operation: why the data is exchanged, what file or file set is involved, and how it is accessed. In IFlux, intent is not a new low-level API; it is a scheduler-facing contract linking semantics to measurable file behavior. The "why" distinguishes goals such as checkpointing, training-data reuse, intermediate exchange, and visualization; the "what" captures file/set size and lifetime; and the "how" captures access mode, sharing, and interface. This lets the mapper choose different accelerators for semantically different files even when calls use the same POSIX or MPI-IO interface. For example, a file-per-process checkpoint can be buffered in node-local NVMe with asynchronous flush, while a collective MPI-IO file of similar size can be mapped to UnifyFS to aggregate node-local capacity across ranks. We use I/O intents for scheduling decisions to identify the required I/O accelerators for the workload.

We propose IFlux, an intent-driven scheduler plugin that selects I/O accelerators for different parts of a workload. IFlux's novelty is a scheduler-level integration of three capabilities usually handled separately: per-file intent capture, joint hardware–software accelerator selection, and automatic deployment plus runtime redirection from the scheduler allocation. Prior intent systems mainly optimize within an already chosen backend, and prior burst-buffer/data-management systems typically require manual stack and tier selection. IFlux moves this choice into the scheduler and evaluates mapper, deployer, executor, and end-to-end effects separately. To reduce the workload complexity, IFlux uses an enhanced job specification incorporating the workload's I/O intents per-file that captures the complex I/O behaviors from workloads. To reduce accelerator complexity, IFlux uses an automated mapper and an execution framework. The IFlux Mapper decides on the appropriate hardware and software accelerators to support complex workload requirements defined by I/O intents. The IFlux Execution Framework interfaces with the hardware accelerators and deploys them on-demand for workloads to utilize them. Additionally, the framework has a middleware library that transparently optimizes the workload's data exchange to utilize the deployed software and hardware accelerators. The middleware examples used in our evaluation include SCR for rank-local checkpoints, UnifyFS for collective and shared namespace access over node-local storage, DYAD for producer-consumer data exchange, HFetch for prefetching read-only or read-mostly files, and HReplica for replicating repeatedly read file sets. The IFlux middleware library is the runtime component that selects and configures these tools based on the mapper's placement schema, so applications can benefit from the chosen software accelerator without embedding accelerator-specific setup code in each workload. Compared to state-of-the-art solutions, IFlux accelerates the Megatron Deepspeed Large Language Model (LLM) workload and 1000 Genome's workflow by 3.12× and 2.85×, respectively. The contributions of this work are:

1) Design of the *IFlux execution framework* that dynamically and transparently maps workloads to I/O accelerators available on HPC systems.
2) Design of the *IFlux mapping engine* that maps I/O intents to hardware and software accelerators to optimize data exchange in scientific workloads.
3) Evaluating IFlux with benchmarks and real workflows to optimize data exchange by 3.12× on HPC systems.

Additionally, we observe from our experiments the following five insights for IFlux's transformative potential in optimizing performance in scientific computing and deep learning environments.

1) **Significant Speedups for Simulation Workloads:** IFlux can provide incredible performance improvements, achieving up to 472× speedup for collective I/O operations and 14.2× for file-per-process I/O compared to Lustre, showcasing its effectiveness in optimizing simulation workloads.
2) **Improved training efficiency for Deep Learning Workloads:** IFlux automatically selects appropriate hardware and software accelerators, allowing deep learning workloads to train up to 15× faster for reading datasets and 10.8× faster for checkpointing models.
3) **Cost Efficiency with Minimal Overhead:** The IFlux Mapper's low-cost infrastructure enables efficient allocation of resources while maintaining minimal runtime overhead, making it an attractive solution for transparently enhancing I/O performance.
4) **Data Exchange Performance Improvement:** By utilizing multiple software and hardware accelerators for varying access patterns, IFlux can increase data exchange performance by up to 2.5× in applications like DSS Montage, indicating its versatility across complex workflows.
5) **Automation of Optimizations:** IFlux's ability to automatically select ideal software and prioritize the right hardware accelerators maximizes performance while reducing the burden on users, thereby streamlining workflow configurations and enhancing overall efficiency.

2 Background and Related Work

The complexity of optimally running scientific workloads on large-scale HPC systems stems from two factors: specialization of I/O accelerators and complexity of usage for users. We explore these two aspects in the current state-of-the-art literature and identify key research that partially supports these topics.

2.1 Specialization of I/O Accelarators

HPC systems use both hardware and software I/O accelerators to improve data exchange performance for large-scale scientific workloads, and users often must request these accelerators explicitly at job submission time. Representative hardware accelerators include node-local SSDs on Frontier, node-local IBM burst buffers on Summit, shared DataWarp burst buffers on Cori, and configurable Rabbits burst buffers on ElCapitan. Representative production software accelerators include UnifyFS [4], VeloC [4], Scalable Checkpoint and Restart (SCR) [13], Proactive Data Containers [3] (PDC), DYnamic and Asynchronous Data Streamliner [5] (DYAD), and Hermes [1]. These accelerators

are specialized for different data-exchange behaviors: node-local NVMe burst buffers are effective for random-access buffering and asynchronous flush; shared burst buffers such as DataWarp target large parallel access across nodes; VeloC and SCR target checkpoint/restart; UnifyFS targets collective I/O in simulations; Hermes targets data-analytics workflows; and DYAD targets AI workloads by leveraging local storage for inter-node file exchange, including producer-consumer workflow exchange and deep-learning data shuffling. This specialization shows that no single accelerator fits all workload situations.

Some solutions in the cloud, such as Gfarm [14] and Pwrake [14], aim to reduce data movement between tasks by placing them close to each other in cloud infrastructures. However, these solutions cannot be applied in an HPC ecosystem. A more recent work proposed DFMan [9] that looks at different data exchange patterns and selects the appropriate hardware accelerator for the workload. However, DFMan is an offline scheduling policy that matches specific data exchange behaviors to hardware accelerators, which limits the scope of workload optimization. To the best of our knowledge, no solution can unify both software and hardware I/O accelerators for scientific workloads.

2.2 Complexity for the User

Scientific and engineering workloads have diverse data exchange use cases that require special care in modern HPC systems [7]–[10]. For instance, the Megatron Deepspeed [15], [16] workload is a Large Language Model (LLM) that benefits from efficient cache locality optimization by, for example, using the DYAD software with node-local burst buffers, or from asynchronous I/O optimization via SCR library for optimizing model checkpointing. Another example from HPC workflows, 1000 Genome [17] is a large-scale complex workflow where 34% of accesses are read-only, 34% are independent reads, and 32% are shared file access. In this case, the read-only optimization requires stage-in into local shared memory, independent reads from some processes may take advantage of dynamic prefetching by DYAD, and shared file access can be optimized using UnifyFS. Currently, most solutions in the literature utilize multiple hardware accelerators but a single software accelerator to support limited modes of data exchanges [1], [3], [4]. However, as shown in the previous section, no one software accelerator can fit all data exchange use cases.

Therefore, the complexity of I/O accelerators and the user's workload demand a dynamic solution that utilizes high-level information from these data exchange use cases and automatically selects the optimal hardware and software I/O accelerators for different parts of the workload.

3 IFLUX

We designed the IFlux scheduler plugin to interface with the Flux Framework (Flux), a state-of-the-art modular and hierarchical resource management and scheduling framework [6]. Flux features directed graph-based resource representation and scheduling, making it well equipped to schedule both traditional (e.g., CPUs, memory, and GPUs) and non-traditional (e.g., I/O, network) resources [6].

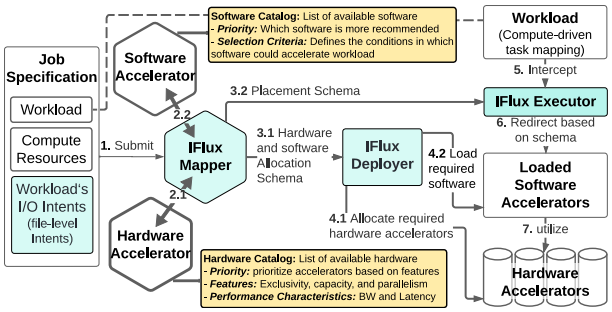


Fig. 1. IFlux runtime environment and software stack in our prototype. Flux allocates compute and scheduler-visible storage from an intent-aware job specification; the mapper consumes intent and accelerator-catalog YAML to emit allocation and file-placement schemas; and the execution framework deploys UnifyFS, DYAD, or SCR while a GOTCHA-based C++ executor transparently redirects POSIX, STDIO, MPI-IO, and Python I/O at runtime.

However, like other schedulers, Flux requires specification of workload resources to make scheduling decisions.

We designed the IFlux scheduler plugin to simplify the user's efficient use of hardware and software I/O accelerators for scientific workloads. To achieve this, IFlux has three main design goals. First, IFlux needs to consider the dynamicity of I/O behaviors within scientific workloads and map these behaviors efficiently to I/O accelerators. Then, IFlux needs to automatically interface with available software accelerators on a cluster and deploy them as a part of the job allocation process. Finally, IFlux needs to transparently utilize I/O accelerators without changing the workload's code. These design goals assist users in building portable code that can automatically optimize themselves for different HPC systems.

At a high level, as seen in Figure 1, IFlux has three parts: a) enhanced job specification with I/O intents, b) IFlux Mapper: a heuristic-driven mapping engine, and c) an execution framework with IFlux Deployer and IFlux Executor to deploy and utilize software accelerators respectively. Together, these three parts allow modern HPC schedulers to automatically allocate hardware accelerators, deploy software accelerators, and transparently utilize these accelerators at runtime. These parts are currently implemented to work with Flux. In Flux, users pass the intent artifact through the IFlux plugin as a command-line input when submitting the job. The plugin extends the Flux job specification with the intent path, runs the IFlux Mapper and deployment-planning components inside the Fluxion scheduler path, and then asks Fluxion to match the job against the storage resources required by the generated allocation schema. Thus, Flux sees an enriched resource request before scheduling and can co-schedule compute and storage resources together. Slurm does not provide an equivalently rich descriptive job specification for this workflow. Therefore, the Slurm path uses a user-space wrapper around `salloc` and `sbatch`: the wrapper accepts the intent artifact, runs the IFlux components before the allocation request is submitted, translates the resulting allocation schema into concrete Slurm storage-resource requests, and then invokes `salloc` or `sbatch` with those requests.

The intent-aware job specification enhances the task

specification by including I/O intents as an intent list or an external artifact. For each task (MPI or single process), we maintain a per-file intent that describes the data exchange behavior for that task with that file. This enhanced job specification is submitted to the IFlux Mapper (Figure 1 arrow 1). IFlux Mapper is a mapping engine that utilizes the job's I/O intents, available hardware accelerators (arrow 2.1), and available software accelerators (arrow 2.2) to map each file to different combinations of the accelerators. The algorithm for mapping is described in Section 3.3. Each accelerator has a satisfiability condition that checks the characteristics of I/O intent and accelerator to choose a valid combination. Additionally, the mapping can prioritize different accelerators based on performance features, such as bandwidth and latency, to choose different accelerators based on workload requirements. The satisfiability condition and accelerator's performance features are present in the accelerator catalog. Finally, the mapper outputs an allocation schema (arrow 3.1) and a file placement schema (arrow 3.2) to the execution framework's IFlux Deployer and IFlux Executor, respectively. The deployer interfaces with all hardware and software accelerators (arrows 4.1 and 4.2) and deploys them. The executor transparently links to the workload (arrow 5) and utilizes the accelerators chosen for the data exchange of specific files within the HPC system (arrows 6 and 7).

The rest of the section discusses the intent-aware job specification in Section 3.1, the structure of accelerator catalogs in Section 3.2, the mapping algorithm in Section 3.3, and the design of IFlux execution framework in Section 3.4.

3.1 Intent-aware Job Specification

The scheduler allocates resources and runs jobs from an input job specification. IFlux extends this specification with workload I/O intents [9]. These intents enable workload-aware software that helps users use system accelerators efficiently. The intent-aware specification carries file-level I/O intents for each job (Table 1). **Technical details:** Existing scheduler descriptions expose CPU, memory, GPU, and coarse storage requests, but not the file-level I/O behavior needed to choose an I/O accelerator. The main challenges are: (1) representing multiple I/O behaviors in one workflow without one manual annotation per rank or file, (2) preserving scheduler portability, and (3) keeping the specification small enough for the scheduling path. IFlux addresses this by attaching task-level and global file-intent objects to the job specification, supporting path-pattern and file-set intents, and storing intents as scheduler-visible YAML referenced by the task environment. If t is the number of tasks, i the number of intent objects, and l the total matched path-pattern length, validating and loading intents costs $O(t+i+l)$ time and $O(i+l)$ space. Because file-set intents summarize repeated rank-local files, i typically scales with distinct I/O behaviors rather than process count. Hardware and software accelerators target specific I/O patterns. Our evaluation shows that file-level patterns are sufficient to capture workflow dynamicity. By file-level dynamicity, we mean that a task can expose multiple simultaneously valid I/O behaviors through different files or file groups. This does not claim file-level intents are the finest possible representation; finer intents, such as block regions, are possible and explored by systems such as Mimir [9]. IFlux uses file/file-set granularity because our middleware optimizations operate at that scope: SCR for checkpoint files, UnifyFS for shared/collective files, DYAD for producer-consumer files, HFetch for prefetching, and HReplica for replication of files or file sets. In this evaluated configuration, block-level intents add detail these middleware choices cannot directly exploit. IFlux also does not assume one application-wide behavior: it records per-file intents and aggregates them at process, workflow, and campaign scopes. This exposes inter-file patterns (for example, many small files with shared lifetimes or repeated reads across a file set), inter-process patterns (producer-consumer or collective sharing), and inter-job patterns (repeated dataset reuse across campaigns). Large small-file collections are represented as file-set intents with count, aggregate size, average object size, sharing mode, and common lifetime, allowing the mapper to distinguish metadata-heavy small-file access from a few large sequential streams. This is why we report intent classes for Montage, 1000 Genomes, UNet3D, and Megatron rather than one label per workload. These behaviors are central to selecting appropriate hardware and software accelerators. Key hardware-selection features include access pattern, file size, and process sharing. For example, large sequential accesses are usually better on distributed PFS, while small irregular accesses are often better on node-local storage. Conversely, process sharing, inter-node access, and interface are often more important for software-accelerator selection. For instance, producer-consumer style process file sharing is more efficient with DYAD [5], [18], [19]. In contrast, collective I/O from several processes is more scalable on UnifyFS [4]. These specific features of each hardware and software accelerator are encoded as satisfiability conditions within the IFlux Mapper.

TABLE 1
The file-level intent that is attached to the job specification.

File-Intent Behaviors	Description
Access Pattern	Types of access pattern from workload: sequential, strided, or random.
File size	Size of file in bytes.
Name	Name of the file.
Workload Type	Behavior classification for workload: write-only, read-only, WORM, or RAW
Process File Sharing	How do processes access file: file-per-process, shared, or collectively.
Priority	The important of the file.

File-level intents include two mapping controls: (a) intent scope and (b) prioritization. The specification supports two scopes: *task-level intents* for specific tasks and *global intents* that apply across tasks. Global intents capture cross-task behavior and enable workflow-wide optimization. Each file-level intent can be prioritized by behavioral attributes (for example, I/O count and size) or performance attributes (for example, I/O time, latency, and bandwidth). Prioritization is optional. If users provide no preference, IFlux defaults to performance, defined as reducing application wall-clock time. Preference changes the order in which hardware and software catalog entries are applied to each file intent. Under performance preference, IFlux ranks hardware by

measured bandwidth and latency for the requested access pattern, and ranks software by expected benefit for that file class. Under persistence preference, it ranks resilient/recoverable devices and middleware above volatile devices, even when volatile devices have higher peak bandwidth. For example, performance preference may try shared memory or NVMe first for temporary data, while persistence preference may first try a recoverable burst buffer, PFS-backed allocation, SCR-managed checkpoint path, or explicit stage-out before a volatile cache. If a user preference is inaccurate or suboptimal, IFlux still enforces hard constraints (capacity, sharing scope, interface support, persistence, software availability) and falls back to performance-based ranking when preferred mappings violate constraints. Users set prioritization in the job specification to optimize the workload objective they care about. This helps the mapper prioritize critical file accesses over less critical ones.

Implementation Details: The enhanced specification is modeled on Flux's job specification. Intents are local filesystem files. Although we used a simple orchestration tool for running jobs [6], any scheduler with basic dependency support can be used. In our approach, a YAML workflow defines tasks and dependencies, and I/O-intent file paths are passed to each step as environment variables. We attach task-specific intents to each command (a task in Flux), and place workflow-wide intents in a special task called `intent-workflow`. Intents are stored as a YAML array referenced from the task specification; each object encodes one file-level intent with the behaviors in Table 1. IFlux obtains intents by automatic extraction or manual annotation. Prior work (Mimir and H5Intent) showed extraction from traces and interface-level observations [9], [20]; IFlux follows similar mechanisms. DFTracer, Recorder, and Darshan traces provide file paths, interfaces, operation types, process sharing, sizes, and temporal order, which are translated into file-level intents [9], [20], [21]. When profiling data is unavailable, or users need domain-specific hints, intents can be manually annotated as in Mimir [9]. Users need not annotate every physical file: one file-set intent can match a pattern such as `chkpt/rank-*.pt`. YAML intent objects include fields such as `access_pattern`, `file_size`, `filename`, `priority`, `process_access_pattern`, and `procs_accessing`. For larger workloads, the same representation summarizes file sets through path patterns and aggregate attributes. Serialized YAML objects are typically hundreds of bytes, so file-set intents keep metadata proportional to distinct behaviors rather than ranks, and large MPI jobs do not require one intent per process unless behaviors are truly distinct.For MPI workloads, the mapping unit remains the file or file set, independent of subcommunicator decomposition. At runtime, each process accesses either a unique file or a shared file; IFlux tracks this per-file relationship. Each intent records interface, access pattern, sharing mode, number of participating processes, and the process set sharing the file. Collective MPI-IO, independent MPI-IO, POSIX-over-MPI, HDF5 file views, and per-subcommunicator access are captured as distinct `interface` and `sharing` attributes on affected files. This lets IFlux recognize high-level patterns such as file-per-process checkpoints, shared collective files, and producer-consumer files without depending on internal communicator structure.

3.2 Accelerator Catalogs

The accelerator catalogs are an indexed database of hardware and software accelerators available on a cluster. Each record contains behavioral and performance features.

Technical details: The scheduler cannot make intent-aware allocations unless it knows which hardware tiers and software accelerators exist, what patterns they support, how much capacity they provide, and how they are deployed. The main challenges are cross-system heterogeneity, changing availability, pattern-dependent performance, and separating hard semantic constraints from performance preferences. IFlux addresses this with one normalized record per accelerator containing behavioral attributes, performance measurements, satisfiability predicates, deployment actions, and teardown actions. If h is the number of hardware accelerators, s the number of software accelerators, r the number of rule predicates, and b the number of benchmark profiles per accelerator, catalog loading and indexing costs $O((h+s)\log(h+s)+r)$ time and $O(h+s+r)$ space at scheduling time; periodic discovery costs $O(h+s+b(h+s))$ offline and is outside the critical job-submission path. For example, a node-local record includes capacity per user, controller type, controller parallelism, byte addressability, maximum bandwidth, and minimum latency. A UnifyFS record includes properties such as mount point, satisfiability condition, and deployment requirements. The mapper uses these fields to prioritize accelerators and map file-level intents. Indexing enables efficient sorting and lookup during mapping. Catalogs also include deployment, initialization, and teardown actions used by the execution framework at runtime. A satisfiability condition is a Boolean rule over intent attributes and accelerator attributes. For example, node-local NVMe satisfies a file-per-process checkpoint when `sharing=file-per-process`, `persistence=stage-out`, and requested capacity is less than available per-node NVMe. UnifyFS satisfies collective access when `sharing=collective` or `interface=MPI-IO` and the job spans multiple nodes. DYAD satisfies producer-consumer exchange when one task writes a file that another task reads and the file lifetime is intermediate. SCR satisfies model checkpointing when files are rank-local, write-dominated, and persistent after stage-out. These rules make the mapping explicit and explain the source of performance improvement: IFlux selects the accelerator whose semantics match the file behavior before using measured performance to break ties.

Implementation Details: Catalogs are extracted periodically by a `cron` job that discovers accelerators and collects behavioral and performance characteristics. This job keeps the system view current as software becomes available and hardware degrades. For hardware discovery, it scans system mount points and detects raw node-local devices (for example, shared memory and NVMe), scheduler-allocated resources (for example, IBM Burst Buffers and HPE Rabbit Nodes), and network mount points (for example, parallel and network file systems). For software discovery, it searches a predefined accelerator list across environment modules and Spack installations. After discovery, it

Input : I_{all}: all file-level intents, H_{all}: full hardware catalog, S_{all}: full software catalog, P: priority policy
Output: C_p: ordered candidate placements, C_a: ordered candidate allocations

$I_p \leftarrow prioritizeIntents(I_{all}, P)$
$H_p \leftarrow prioritizeHardware(H_{all}, P)$
$S_p \leftarrow prioritizeSoftware(S_{all}, P)$

Function *IFluxMapper*(I_p, H_p, S_p)**:**
  for $i \in I_p$ **do**
    $matched \leftarrow false$
    for $h \in H_p$ **do**
      if $h.available$ **then**
        $ok, placement, s_{sel} \leftarrow MapIntent(i, h, S_p)$
        if ok **then**
          $C_p.add(i, placement)$
          $C_a.add(i, h, s_{sel})$
          $matched \leftarrow true$
    if *not* $matched$ **then**
      $C_p.add(i, default_pfs(i))$
  return C_p, C_a
Function *MapIntent*(i, h, S_p)**:**
  if $satisfy(i, h)$ **then**
    $placement \leftarrow place(i, h, None)$
    return $true, placement, None$
  for $s \in S_p$ **do**
    if $satisfy(i, h, s)$ **then**
      $placement \leftarrow place(i, h, s)$
      return $true, placement, s$
  return $false, None, None$

Algorithm 1: High-level intent-aware mapping prioritizes intents and catalogs, attempts native placement first, then software-assisted placement, emits ordered candidates, and falls back to default PFS.

gathers behavior and performance features. Behavioral characteristics come from predefined rules by accelerator type, while performance comes from I/O benchmarks such as IOR [7] and DLIO Benchmark [22]. The benchmark profile is pattern-specific rather than a single device number: IOR samples read/write, sequential/random, shared-file/file-per-process, and representative transfer sizes, while DLIO samples small-file training reads and checkpoint writes. Catalog extraction runs periodically and can also be triggered after a system maintenance event or when monitoring detects sustained deviation from the cataloged bandwidth. Shared tiers such as the PFS are treated as capacity-rich but contention-sensitive resources; their catalog records include measured bandwidth under the current sampling window and are updated conservatively so that IFlux does not overfit to a transient idle period. Discovered accelerators and their characteristics are stored as one YAML file per system. By default, accelerators are indexed and prioritized by performance characteristics.

3.3 IFlux Mapper

The IFlux Mapper selects per-job hardware and software allocations from three inputs: intent-aware job specification, hardware catalog, and software catalog. **Technical details:** The mapper solves selection of a valid, high-performing combination of hardware tier, software accelerator, and placement action for each file intent without manual accelerator enumeration. Main challenges include combinatorial search across intents and accelerators, capacity constraints, accelerator-specific semantics, mixed read/write phases, and incomplete intents. IFlux addresses this with prioritized candidate generation: it sorts intents and catalogs by policy, checks hard satisfiability before performance preferences, and emits ordered candidates plus conservative PFS fallback for unmapped/invalid intents. Candidates are passed from most to least preferred so the scheduler can choose the best feasible option under resource availability and user-imposed time limits. For i intents, h hardware choices, s software choices, and c as maximum satisfiability-check cost, worst-case mapping time is $O(ihsc)$. If k valid candidates are emitted, the mapper stores placement schema, allocation schema, and availability state in $O(k + h + s)$ space, with $k \leq ihs$. The heuristic engine has three phases: prioritization, mapping, and output. In prioritization, it sorts all three inputs using user-defined job priorities. The mapper intentionally evaluates candidate mappings after sorting. Thus, the candidate order is not arbitrary: it reflects the selected policy from most preferred to least preferred. User preferences are applied to all three mapper inputs: they sort the intent list, the hardware catalog, and the software catalog before mapping begins. Therefore, the order in which users provide intents does not determine the final mapping order; IFlux reorders intents according to the selected preference, such as expected performance impact, footprint, persistence, or another user-specified priority. The same preference also changes which storage accelerators and software accelerators are tried first for a job. For example, a performance preference sorts hardware and software by measured bandwidth or latency for the relevant pattern. A job-characteristic preference instead ranks candidates by semantic fit to intent traits such as I/O size, sharing semantics, access pattern, and read/write role: large sequential read-mostly files favor high-throughput shared tiers and prefetch-capable services, small random files favor low-latency node-local tiers, file-per-process checkpoint files favor SCR-like paths, and shared or collective files favor shared-namespace services such as UnifyFS before generic storage choices are considered. The mapping phase runs a greedy search over prioritized inputs against available cluster accelerators and returns allocation and placement schemas for runtime execution.

The mapping algorithm is shown in Algorithm 1. The algorithm uses collection variables I_p, H_p, and S_p for prioritized lists and element variables i, h, and s inside the loops. It iterates over $i \in I_p$ and $h \in H_p$, then evaluates $s \in S_p$ through $Map(i, h, s)$ to emit ordered candidates. Hardware availability for an intent is constrained by capacity, scope (node-local or shared), and intent parallelism requirements. Once prioritized hardware h is identified for intent i, the $Map(i, h, s)$ function first checks native satisfiability $satisfy(i, h)$ and then checks software-assisted satisfiability $satisfy(i, h, s)$ when needed. Native support is checked first to minimize overhead, because software abstraction can add redirection cost [4]. Software acceleration is considered when prioritized hardware cannot be used

natively. For example, node-local tiers cannot natively accelerate multi-node collective I/O, but UnifyFS can provide transparent collective acceleration over node-local storage. If no candidate is found for an intent i, the algorithm adds $default_pfs(i)$ to C_p as the fallback storage choice. IFlux uses intents for allocation planning (capacity and scope), deployment planning (stage-in/stage-out), software setup (startup parameters), runtime execution (file-redirection rules), optimization planning (prefetching/caching), and validation (conservative PFS fallback for incomplete or inconsistent intents). If an intent maps to no accelerator, IFlux leaves that file on the PFS and records the failed rule for later validation. Mixed read/write patterns are represented as WORM, RAW, or separate read and write intents when phases have different lifetimes or sharing modes. Matched mappings produce the allocation and placement schemas returned by the mapping phase. Algorithm complexity is $O(i \times h \times s)$, where i is the number of intents, h the number of hardware choices, and s the number of software choices.

The mapper outputs two schemas: allocation and placement. The allocation schema lists hardware/software to deploy, and for each accelerator the catalog provides allocation, initialization, and teardown actions. For example, node-local NVMe entries can request device size plus stage-in/stage-out for persistence, while software entries such as UnifyFS can load modules at allocation, start services at initialization, and stop services at teardown. The placement schema provides per-file runtime directives for accelerator selection, mount points, prefetching, and caching. IFlux optimizations depend on middleware present in the system catalog. In our prototype, the catalog includes SCR [13], UnifyFS [4], DYAD [5], [18], HFetch [11], and HReplica [12]; IFlux uses them transparently by selecting, deploying, and configuring them per workload part. Heuristic criteria come from prior I/O and data-management literature and are encoded as intent-to-optimization rules. For example, read-only/read-mostly file sets known before execution map to stage-in or HFetch prefetching when capacity permits; repeated reads of shared datasets map to HReplica-style replication/caching; producer-consumer intermediates map to DYAD to avoid repeated PFS exchange; collective MPI-IO over node-local storage maps to UnifyFS; and rank-local checkpoint writes map to SCR or asynchronous flush through node-local storage. Storage-device selection is driven by cataloged device class, mount point, scope, capacity, bandwidth, and latency across shared memory, NVMe/SSD, burst buffer, and PFS tiers. During execution, data follows placement policy: node-local checkpoint data is captured by SCR or the executor, staged on shared memory or NVMe, and flushed to PFS during computation or teardown, while UnifyFS exposes an aggregated namespace over node-local storage and later drains persistent files to PFS.

Implementation Details: The mapping algorithm is prototyped in Python. Inputs are YAML files processed with Jmespath [9] for portable dictionary querying and manipulation. Satisfiability conditions are stored in YAML and evaluated by the engine (via `eval`) during mapping. Mapper outputs are written as JSON Lines for efficient indexing and parallel processing by the execution framework.

3.4 IFlux Execution Framework

The IFlux execution framework transparently applies placement and allocation schemas produced by the mapper. It enables scheduler-driven runtime optimization without workload source changes. It has two components: IFlux deployer and IFlux executor. The deployer allocates and deploys accelerators; the executor intercepts workload I/O and routes it through deployed accelerators. **Technical details:** The key problem solved by the execution framework is converting a static mapping decision into correct runtime behavior without modifying application source code. The main challenges are coordinating accelerator lifecycle with the scheduler allocation, starting distributed software services, applying per-file redirection at low overhead, preserving persistence semantics, and monitoring failures or capacity pressure. IFlux solves this by splitting execution into a deployer and executor: the deployer allocates, initializes, monitors, and tears down accelerators, while the executor builds a prefix tree from placement rules and intercepts supported I/O interfaces to redirect or optimize file accesses. If a is the number of allocated accelerators, n is the number of allocated nodes, and p is the number of placement rules, deployer orchestration costs $O(an)$ control actions in the worst case and stores $O(a)$ accelerator state. Executor initialization builds the prefix tree in $O(pl)$ time and space, where l is the average path length; each intercepted open or path lookup costs $O(l)$ time, while read/write calls that do not require redirection-specific optimization proceed with constant additional state lookup after the file descriptor is classified. The deployer has four phases: allocation, initialization, runtime monitoring, and teardown. During allocation, it interfaces with the scheduler to reserve required hardware and software accelerators. For hardware, it requests required capacity through scheduler commands; for software, it loads services via environment modules and Spack [6]. During initialization, it mounts hardware tiers and starts required software services. During runtime monitoring, it tracks accelerator health, performance, and availability, and collects statistics for future mapping decisions. The monitoring service records capacity consumption, read/write throughput, operation rate, accelerator health, queue depth when available, stage-in/stage-out time, cache hit rate, prefetch accuracy, and software-service failures. These metrics are not on the critical path of each I/O call; they update the catalog and can trigger future remapping or conservative fallback when an accelerator becomes full, degraded, or unavailable in a dynamic HPC environment. Finally, teardown stops software services and deallocates hardware accelerators at job completion.

The IFlux executor is the runtime component that helps workloads use deployed accelerators transparently. It is implemented as a shared library attached to the workload process. Using GOTCHA [21], it intercepts I/O calls and redirects them to selected hardware/software allocators. The executor has four phases: initialization, redirection, optimization, and finalization. Initialization reads mapper placement data, enables required interface interception, builds the prefix-tree path index, and initializes software accelerators. Redirection matches intercepted files to selected

accelerators. During I/O, optimization applies operations such as prefetching and caching to improve performance. The middleware library contributes the runtime part of the optimization: it reads the placement schema, intercepts file opens and selected I/O calls, rewrites paths or initializes software-specific handles, and applies per-intent optimizations without changing workload source code. This hides hardware mount points, software-specific environment variables, service endpoints, and data movement commands from users, reducing the manual tuning required across the I/O stack. Finalization disables redirection, stops software services, and clears executor internal data structures.

Implementation Details: The IFlux Deployer is implemented in Python and interfaces with hardware accelerators, software accelerators, and the scheduler. It uses Python `mpssh` to orchestrate distributed service initialization and teardown. The IFlux executor is a C++ shared library with multiple interface wrappers, including POSIX and STDIO from Python and C workloads, to efficiently utilize accelerators. These scripts and libraries are integrated with `ctest` in our repository.

4 Evaluation

We assess IFlux's impact on scheduling by examining its key components: IFlux Mapper and IFlux Execution Framework. Then, we explore the scalability and performance benefits of IFlux using community-recognized benchmarks such as IOR [7] and DLIO Benchmark [22] to represent a variety of data-intensive workloads in HPC systems. Finally, we evaluate the performance benefit of IFlux on real scientific workloads against state-of-the-art solutions such as UnifyFS [4] and Mimir [9] running on large-scale HPC systems.

4.1 Methodology

4.1.1 Hardware and Software

We run our experiments on an institutional cluster, comprising 121 nodes with a 48-core AMD Rome processor, 256GB of memory, 1.5 TB NVMe SSD, and eight AMD MI50 GPUs each. We run the benchmarks and scientific workloads using Python 3.9.12, OpenMPI 4.1.2, and UCX 1.15.0. The Python packages included pydftracer 1.0.4, torch 2.2.1, h5py 3.10.0, and Numpy 1.26.0. This software setup enables multiprocessing and distributed DL training with MPI, alongside UCX communication for the PyTorch workload. In this setup, we include five system software: Scalable Checkpoint and Restart (SCR) Library [13], UnifyFS [4], DYAD [5], [18], and Algorithms prefetching and caching from HFetch [11] and HReplica [12]. Additionally, to trace I/O calls and workload-level functions, we utilized DFTracer version 1.0.4, which supports C, C++, and Python workloads. All the test cases shown are wrapped using the `ctest` testing framework to enable easy reproducibility of the results.

4.1.2 Workloads

We use a collection of I/O benchmarks, namely, IOR [7] and DLIO Benchmark [22], to represent the I/O behaviors of large-scale scientific workloads, such as simulation, data analytics, and deep learning (DL). IFlux transparently optimizes these workloads without any changes to these workloads except for the DL workloads, where a new data loader and checkpointing strategy implementation is dynamically loaded into the workload. Additionally, we test IFlux against four real scientific workloads, such as Montage [23], 1000 Genomes workflow [17], UNet3D DL training [24], and Megatron Deepspeed LLM [15], [16]. We compare our solution against the current execution of these workloads over Lustre and a state-of-the-art optimized version that uses a single hardware and software accelerator.

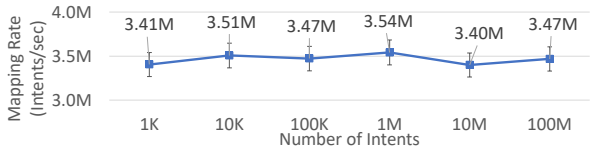


Fig. 2. IFlux Mapper has a throughput of 3.5M intents per second, which would map 1K events in 280 µs.

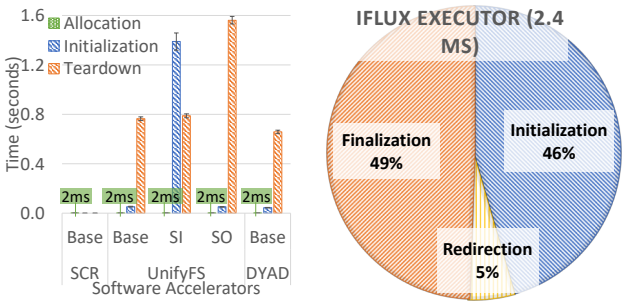
4.1.3 Metrics

To understand the capability and performance benefit from IFlux, we use three metrics: a) Operations per second, measuring the throughput of operations in the system; b) Time, measuring the runtime of an operation or the whole workload; and c) I/O Bandwidth: measuring the rate of bytes transferred to the storage system as perceived by the workload. Additionally, we ran each test 15 times and reported the average and the standard deviation for every test.

4.2 Performance of IFlux Mapper

The IFlux Mapper assists the scheduler in utilizing intents and the accelerator catalogs to select the appropriate hardware and software accelerators for a given workload. Therefore, the mapper's performance determines the increase in scheduling time by IFlux for a workload. The mapper improves scheduler throughput because it converts accelerator selection into indexed rule evaluation over compact intent and catalog records instead of launching probes, parsing application traces, or testing candidate accelerators at job submission time. The technical problem fixed by the mapper is the combinatorial and user-driven selection of hardware tiers, software accelerators, and per-file placement. IFlux reduces this problem by precomputing accelerator behavior/performance catalogs, sorting candidates once per job policy, evaluating satisfiability predicates, and emitting allocation and placement schemas that can be consumed directly by the execution framework. To evaluate this, we run a micro-benchmark using a catalog of eight accelerators available on an institutional cluster, which has three hardware and five software components, including a parallel file system and native hardware access. The benchmark runs the IFlux mapper with varied intents from 1K intents to 100M intents. For these tests, we calculate the rate of processing intents as intents per second. The results are presented in Figure 2 with the x-axis depicting the number of intents passed and the y-axis showing the mapping rate in intents per second.

The results show that the algorithm's mapping rate is, on average, around 3.5M intents per second across different numbers of intents. As most scientific workloads running on several hundreds of nodes would have intents in the 1K



(a) Deployer (b) Executor

Fig. 3. IFlux Deployer takes 53 ms for init and .76 sec for teardown, and the executor can redirect I/O operations with an average cost of 100 µs.

- 10K range [9], we expect an average added cost of 280 µs to 2 ms on scheduling time for the scientific workload. The additional cost (approximately 0.02%) is negligible in comparison with a typical scheduling cycle time (including queue wait time and resource allocation time) of tens of seconds to minutes.

Findings 1: The IFlux Mapper's low cost comes from evaluating pre-indexed catalog rules and compact per-file/fileset intents; this enables schedulers to infer the required hardware and software accelerators without adding a measurable delay to normal queueing and allocation time.

4.3 Performance of IFlux Execution Framework

IFlux's runtime performance depends on the cost of deploying I/O accelerators by IFlux Deployer and the runtime overhead of the IFlux Executor. Therefore, we design two tests that measure the cost of both these components of the execution framework. The first test measures IFlux's cost of deploying software accelerators that are present on an institutional cluster. Specifically, we test the deployment routines for Scalable Checkpoint and Restart (SCR) library, UnifyFS, and DYnamic and Asynchronous Data Streamliner (DYAD). Additionally, we test the stage-in (SI) and stage-out (SO) functionality of the UnifyFS software accelerator. Specifically, we stage in or out 4 GB file per node. In this test, we deploy the software accelerators on 32 nodes and measure the time taken for the software accelerator's allocation, initialization, and finalization. The results are presented in Figure 3(a) where the x-axis represents different software accelerators and the y-axis represents the time taken in seconds.

We observe that the time taken for allocation, which includes loading environment modules for each of these accelerators, takes 2 ms on average. The initialization and finalization cost for SCR is zero as it is only a library that is linked with the workload. For UnifyFS and DYAD, the initialization cost without stage-in (SI in the figure) is 45 to 53 ms. UnifyFS's stage-in operation loads a 128 GB dataset in parallel and takes only 1.38 seconds on node-local storage from Lustre file system. Similarly, the teardown phase of the deployment for UnifyFS and DYAD without stage-out (SO in the figure) is 0.65 to 0.76 seconds. Like stage-in, UnifyFS's stage-out flushes the data into a PFS in parallel, taking only 1.56 seconds.

Findings 2: IFlux can efficiently deploy software accelerators to optimize workload's runtime with minimal cost.

The second test measures the runtime overhead of the IFlux executor by measuring the initialization, redirection, and finalization costs. Specifically, we measure these costs using a benchmark that performs 10K tests. In each test, we initialize, open a file (IFlux performs redirection), perform a 1MB write, and then finalize. During the workload's execution, the initialization and finalization phase occurs once per test per process, and the redirection phase occurs for every file open call. We use POSIX API over Lustre file system to open file. The results are presented in Figure 3(b).

We observe that across 10K runs, the average cost of the IFlux executor, which includes initialization, redirection, and finalization is 2.4 ms. The bulk of the time is taken by initialization and finalization routines, which is about 1.15 ms on average. The redirection cost is smaller (around 122 µs) due to the prefix tree-based implementation to compare and redirect I/O operations. As compared to POSIX file open, which costs ≈500µs, this is 22% of the open cost. As redirection only occurs during the opening of a file, this cost is amortized over many read-write operations (each costing 2.5ms).

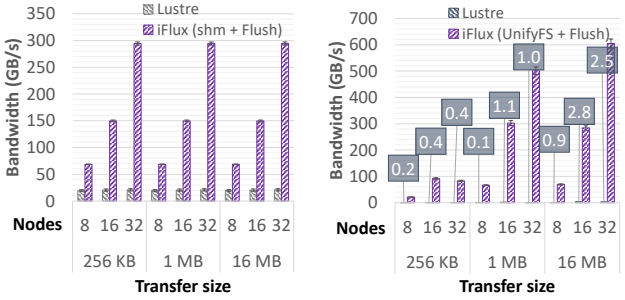
Findings 3: IFlux executor incurs a minimal runtime overhead (0.01%) on the workload to utilize storage accelerators.

4.4 IFlux with IOR Benchmark

To test the effectiveness of IFlux in a controlled environment, we evaluate it with different configurations for IOR that emulate different classes of workloads: simulation and data analytics. To emulate simulation workloads that checkpoint their state periodically, we configure IOR to perform independent file accesses [7], [25] (file-per-process pattern with POSIX interface) and collective I/O accesses [25], [26] (using MPI-IO interface). To emulate data analytics workloads that have read-heavy access patterns, we configure IOR to perform independent reads as seen by map-reduce style analysis [27], [28] (file-per-process pattern with POSIX interface) and collective reads as seen by scientific clustering algorithms [25], [29] (using MPI-IO interface).

For each of these tests, we tested the workload with a 1 GB block size per process to avoid OS caching, and varied different transfer sizes popularly used in HPC workloads: 256 KB, 1 MB, and 16 MB. Additionally, we scaled our test from eight to 32 nodes with 48 processes per node. In these cases, to emulate large-scale data-intensive runs, we assume the shared memory can only capture 25% of the data per node, and the rest can fit into the node-local NVMe. Also, all files have the same priority for scheduling, as the distribution of I/O is uniform. Finally, IFlux uses node-local accelerators but it flushes all the data to Lustre for a fair comparison. Figures 4 and 5 show the results from the four tests, with the x-axis showing the different transfer sizes with the number of nodes and the y-axis showing the Bandwidth in GB/s.

The simulation results (Figure 4) show that Lustre performs 10× faster for file-per-process access pattern than for collective I/O. For IFlux, it automatically allocates and utilizes different storage accelerators for each case. In the case of file-per-process write (Figure 4(a)), IFlux allocates the shared memory and node-local NVMe to capture the I/O and then asynchronously flushes that to the parallel file system. This is possible as IFlux Mapper knows the file size for each process from the I/O intents supplied in the job specification. Therefore, it can determine how many



(a) File-Per-Process Write (b) Collective I/O Write

Fig. 4. IFlux accelerates IOR Benchmark's write-heavy simulation workloads by up to 14.2× for file-per-process and 471× for collective I/O.

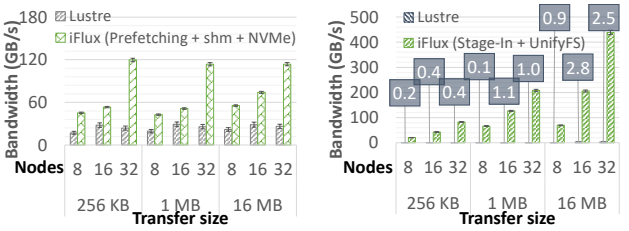
files can fit into shared memory and automatically spill the rest into node-local NVMe. This overall usage of node-local storage leads to a performance improvement of 3.5× to 14.2× with more significant improvement for larger scales as the performance of node-local devices scales linearly as compared to Lustre.

In the case of collective write (Figure 4(b)), IFlux allocated hardware accelerators, such as shared memory and node-local NVMe, and software accelerator, UnifyFS. IFlux configures UnifyFS to use shared memory and then split its data into node-local NVMe. At runtime, the IFlux executor configures UnifyFS to use internal block size based on I/O intents and redirects all I/O operations to UnifyFS, achieving an aggregate bandwidth of 604 GB/s for 16 MB transfer size for 32 nodes. Without IFLux, the internal block size in UnifyFS defaults to 256 KB, which would be fine for the 256KB case but would allow smaller accesses for 1MB and 16MB cases. Again, similar to the file-per-process, the scalability of the IFlux solution through UnifyFS is linear with a larger number of nodes. Overall, we see a speedup of 81× to 471× for collective I/O write operations with UnifyFS, which uses node-local accelerators and asynchronous flushing to Lustre.

The result from the analysis workload (Figure 5) shows that Lustre is 10× faster for file-per-process than collective I/O, similar to the simulation workload results. For IFlux, it again allocates and utilizes different accelerators for these workloads. In the case of file-per-process read (Figure 5(a)), IFlux allocates the shared memory and NVMe for all the files but performs a dynamic prefetching on the opening of the files. This increases the file's open time but significantly reduces the read time for all consequent operations. This benefit is evident from the figure where the overall IFlux's performance is 1.8× to 5× more than that of Lustre. The read performance improvement is not as large as the write improvement because, in an I/O benchmark, asynchronous prefetching cannot effectively hide the I/O cost as there is no computation.

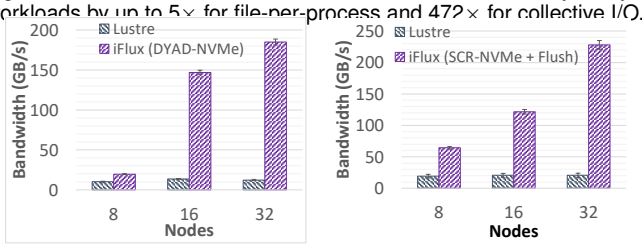
In the case of collective read (Figure 5(b)), IFlux allocated similar accelerators to the collective write case. However, it instructs UnifyFS Service to stage in the data during initialization. This allows the workload to increase I/O bandwidth for read operations, resulting in a bandwidth of 438 GB/s in the best case. The bandwidth of Lustre for collective operations is lower than for file-per-process. Therefore, IFlux achieves a speedup of up to 472× over Lustre.

Findings 4: IFlux can automatically choose the appropriate



(a) File-Per-Process Read (b) Collective I/O Read

Fig. 5. IFlux can accelerate the IOR Benchmark's read-heavy analysis workloads by up to 5× for file-per-process and 472× for collective I/O.



(a) Reading Pipeline (b) Model Checkpointing

Fig. 6. IFlux can accelerate DLIO Benchmark for dataset reading by 15× and checkpointing by 10.8× as compared to Lustre.

accelerators for simulation workloads and provide a speedup of up to 472× for collective I/O operations and up to 14.2× for file-per-process I/O operations as compared to Lustre.

4.5 IFlux with DLIO Benchmark

To test the effectiveness of IFlux for deep learning workloads in a controlled environment, we evaluate it with different configurations for DLIO Benchmark that emulate the two types of data access within deep learning training: dataset reading and model checkpointing.

To emulate deep learning (DL) training's dataset reading, we configure the DLIO benchmark to run the input reading pipeline similar to the UNet3D model [24], which accesses 168 files with one sample per file. Each sample has an average size of 140MB with a standard deviation of 48 MB. We run this training for twenty epochs with a batch size of four samples. In contrast, we emulate deep learning checkpointing using DLIO Benchmark's Megatron Deepspeed workload [15], which checkpoints its model every thousand steps. It performs a data parallel checkpoint where every process checkpoints its own model during a data parallel training stage. Each checkpoint saves the optimization state, model parameters, and layers with an aggregated checkpoint size of 2.2 GB per process. We measure the time taken to perform I/O operations from the POSIX layer using the DFTracer tool [21]. We show the results from this test in Figure 6, with the x-axis showing the number of nodes and the y-axis showing the bandwidth in GB/s.

The read performance for DL training (Figure 6(a)) shows that Lustre provides a bandwidth of 10 to 14.5 GB/s. For IFlux, the read-only workload with data sharing across multiple processes is mapped to the DYAD software with node-local NVMe. For DYAD, the IFlux deployer runs the DYAD Service module within the Flux and loads the DL workload with DYAD's data loader. With this, IFlux achieves an I/O bandwidth of 185 GB/s with 32 nodes. This improves the DL training performance by 15× as compared

to Lustre. The checkpointing performance of DL training (Figure 6(b)) shows that Lustre gives a bandwidth of 21 GB/s, similar to the file-per-process write performance from IOR. In contrast, IFlux uses the Scalable Checkpoint and Restart (SCR) library to store the checkpoints on node-local NVMe and then asynchronously flush to Lustre during model computation. This allows the model checkpointing to achieve a bandwidth of 228 GB/s with IFlux, making it 10.8× faster than Lustre.

Findings 5: For DL workloads, IFlux can automatically select hardware and software accelerators for multiple phases, allowing workloads to be up to 15× faster for reading and 10.8× faster for checkpointing compared to Lustre.

4.6 IFlux with Scientific Workloads

To evaluate IFlux's impact on real-world HPC workloads, we tested it for four exemplar scientific workloads: Montage workflow, 1000 Genomes, Unet3D DL training, and Megatron Deepspeed LLM. These are large-scale, complex workloads that exchange large datasets, and optimizing them has a huge impact on the scientific community.These experiments also evaluate whether file-level intent granularity is sufficient for the optimizations IFlux applies. Across the four workloads, the mapper only needs to distinguish file and file-set classes, such as read-only inputs, repeated dataset reads, rank-local checkpoints, shared collective files, and producer-consumer intermediates, because the selected middleware libraries act on those same units. The evaluation therefore does not claim that block-level intents are unnecessary for every possible storage optimization; rather, it shows that file-level intents provide enough information to choose and configure the file-scoped middleware optimizations used by IFlux.

4.6.1 Digitized Sky Survey (DSS) using Montage

The Montage mosaic [23] 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 2° of the sky along with the galactic plane, where each mosaic image is .5° by .5° in size and has an overlap of .25° 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 Lustre and Mimir. We strongly scale the workload from eight to 64 nodes and compare the different classes of behaviors within the workload.

Intents: The workflow-level file-access intent captured 6039 processes with 48 tasks per node. At the application level, the per-file intent contains the total I/O, size of the file, access pattern (e.g., sequential, random, or strided), process access pattern (e.g., file-per-process, collective I/O, or shared), and distribution of operations. The workflow-level file-access intent contains independent file accesses in comparison with producer-consumer files. Specifically, we observe that the workflow spends 90% of its I/O time in reading and writing intermediate files (i.e., data produced by one process is consumed by another process but not used thereafter), 7% of files are read-only (e.g., mostly configuration files and initial fits file for building

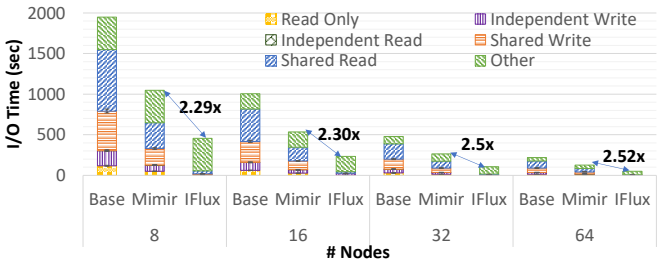


Fig. 7. IFlux uses various accelerators for different parts of the Montage workflow and optimizes it by 2.5× compared to Mimir and 4× compared to Lustre file system.

the mosaic), 12% are file-per-process writes (e.g., final mosaic and intermediate auxiliary files which are not used), and 70% are shared access on files (mostly from producer consumer files). The "Other" category in the figure contains metadata-oriented accesses that do not map to the optimized read-only, independent, shared, or producer-consumer intent classes used in the Montage analysis. This includes metadata accesses on Montage table files, logging files such as `stdout` and `stderr`, and file-existence probes such as `stat` and `access` calls. These accesses would not be optimized and would be served by the Lustre file system. This feature allows IFlux not to have effects on new and unexplored intents and allows the system to identify new intents that need to be identified and optimized.

The results for Montage are presented in Figure 7, with the x-axis showing the number of nodes along with the solution tested and the y-axis showing the I/O time in seconds. We reference Figure 7 before interpreting the trends so the reader can first identify the plotted workloads, baselines, and intent categories. We compare Montage with Lustre (Base in the figure) and Mimir as a Baseline. Lustre reduces the I/O time from 1949 to 218 seconds as we scale 64 nodes. Mimir optimizes different files based on their intents by placing them into different storage accelerators [9]. This results in a speed-up of 2.3× over Lustre.

For IFlux, the IFlux Mapper assigns files with different intents to a specific combination of I/O accelerators. The read-only files are optimized using data stage-in into node-local NVMe; independent writes are optimized using shared memory buffering with asynchronous flushing; independent reads are optimized by prefetching data into node-local NVMe; and finally, shared reads and writes are optimized using DYAD as they follow consumer-producer patterns. The performance improvement therefore comes from different concrete decisions for different intent classes, not from the presence of IFlux alone: stage-in removes repeated read traffic from Lustre, buffering hides write latency behind computation, prefetching moves future read data into faster local storage, and DYAD bypasses repeated PFS exchange for producer-consumer files. As the time for deployment of these solutions is a few seconds, we only show the runtime performance with the IFlux executor. We see that read-only files are optimized by 15.3×, independent files by 3.4×, independent reads by 15.2×, and shared file accesses by 41× on average. This makes IFlux 2.3× to 2.5× faster than Mimir.

Findings 6: Using multiple software and hardware accelerators for different access patterns can increase data exchange performance for DSS Montage by up to 2.5×.

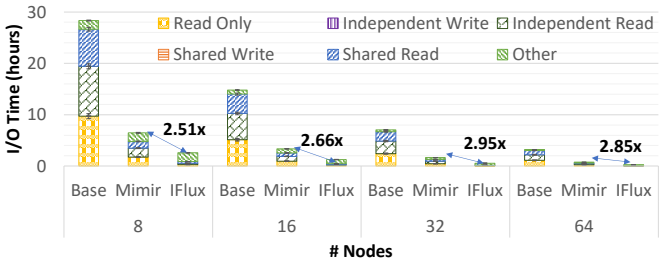


Fig. 8. IFlux can appropriately choose accelerators based on intents and optimize the 1000 Genome workflow by up to 3× compared to Mimir.

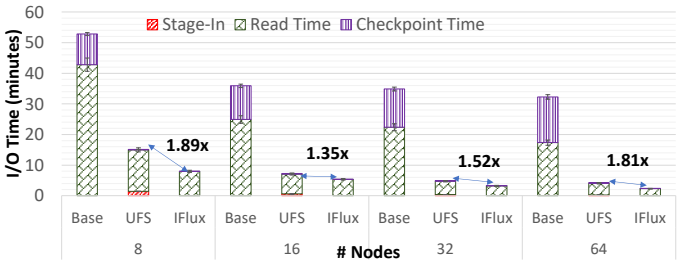


Fig. 9. IFlux optimizes the Unet3D DL workload by 1,89× compared to UnifyFS by using DYAD and SCR with node-local storage.

4.6.2 1000 Genome workflow

The 1000 Genome workflow [17] with the Pegasus workflow manager 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 the mutations are). It then performs a few analyses, including plotting. We run the workflow over Lustre and Mimir. We strongly scale the workload from eight to 64 nodes and compare the different classes of behaviors within the workload.

Intents: The file-level intents are captured across 1736 tasks. The application level per-file I/O intents show that the *individuals* application performs 70TB, and the *frequency* application performs ≈500GB of read and write operations. Read operations dominate I/O, with 72TB of I/O on reads and only 100GB of I/O on writes, indicating that the workload across many tasks repeatedly reads the same files. In general, 34% of the files are only read by the workflow, 34% of the files are accessed using the file-per-process pattern (i.e., independent read or write to files), and multiple tasks of the workflow access 32% of the files at the same time. As the reads dominate the overall I/O time, we expect the read intents to have a larger impact on optimization. Therefore, we prioritize read-heavy files over write-heavy for optimization.

The results for the 1000 Genome workflow are presented in Figure 8, with the x-axis showing the number of nodes along with the solution tested and the y-axis showing the I/O time in seconds. We compare the workflow with Lustre (Base in the figure) and Mimir. Similar to the Montage workflow, Lustre reduces the I/O time from 28 to 2.3 hours as we scale the workload from eight to 64 nodes. With Mimir, different read-heavy intents are mapped to higher level of the hardware accelerators which reduces the workflow time for 64 nodes to 47 minutes. This results in a speedup of 4× - 4.4× over Lustre.

For IFlux, the IFlux Mapper assigns files with different intents with a specific combination of I/O accelerators. The read-only and the independent read files are staged into node-local NVMe for the workload and replicated across all the compute nodes. This is possible as the workload repeatedly reads the same data for running the genetic algorithm (i.e., 72 TB of read I/O from 100 GB of dataset). The shared files are read by multiple processes and are optimized using UnifyFS, as many processes read them concurrently. This results in an average speedup of 6.6× for read-only, 6.6× for independent read files, and 4.8× for shared read files. This results in an overall speedup of 2.5× to 3× for the workflow.

Findings 7: Multiple intents can be optimized by the same I/O accelerator mechanism and provide speedup to the workload like independent reads and read-only files. Also, the parallelism of shared reads determines the Mapper's choice of software accelerator between DYAD and UnifyFS.

4.6.3 UNet3D DL training

UNet3D is a network for volumetric segmentation in medical imaging, using a 20-layer deep convolutional network to learn from 10,240 sparsely annotated images (140 MB each, totaling 1.36 TB in NPZ format) [24]. Utilizing PyTorch for data parallel training, the model processes batches of four images across six threads per GPU over twenty epochs with checkpointing, scaling from eight to 64 nodes with eight GPUs each. We test the workload by reading the dataset from Lustre and from the pre-staged dataset in UnifyFS.

Intents: There are two types of file-intents: intents for training and intents for model-checkpointing files. For training files, we have read-only intents with a normal distribution of transfer size (based on the sample size of the dataset) with a mean sample size of 145MB and a standard deviation of 68MB. Once the sample is read, the sample is resized into a 2MB image for training. For checkpointing, the AI workload checkpoints the model parameters every five epochs. The average size of the model is 498MB and is saved using PyTorch save API, which saves the tensor data as a dictionary. Overall, the workload spends the majority of its time (80%) doing reading and rest on checkpointing. Therefore, the IFlux infrastructure prioritizes dataset reading over checkpointing while configuring the space on storage accelerators on the HPC system.

The results for the UNet3D DL training workload are presented in Figure 9, with the x-axis showing the number of nodes along with the solution tested and the y-axis showing the I/O time in seconds. We compare the workflow with Lustre and UnifyFS as a Baseline. Lustre reduces the overall I/O time from 52 to 32 minutes when scaled to 64 nodes. UnifyFS optimizes both the DL reading and checkpointing by 4.1× and 68×, respectively. This is because UnifyFS stages the dataset in parallel into node-local NVMe and optimizes reads. Additionally, it stores all checkpoints on NVMe with asynchronous transfer to Lustre, further improving I/O performance.

For IFlux, the IFlux Mapper assigns the files used for reading to DYAD with shared memory accelerators and

checkpointing to SCR with shared memory as the cache. This smarter selection improves the read time by 1.4× and checkpointing time by 7.2× over UnifyFS. Although both UnifyFS and DYAD can optimize the DL training's dataset reading, DYAD maximizes the locality of the sample and improves training performance. Overall, IFlux optimizes the UNet3D workload by 1.35× to 1.89× over the UnifyFS solution.

Findings 8: Choosing the ideal software for the workload is critical to maximizing its performance (by up to 1.89×). Also, the system should automatically drive these optimizations and not burden users to configure them manually.

4.6.4 Megatron Deepspeed LLM

Megatron is a large, powerful transformer developed by the Applied Deep Learning Research team at NVIDIA [15]. Microsoft has a version of Megatron integrated with its DeepSpeed [16] framework. In this case, Microsoft developed efficient, model-parallel (tensor, sequence, and pipeline), and multi-node pre-training of transformer-based models such as GPT, BERT, and T5 using mixed precision. In this workload, we use the 0.9.5 version of the Megatron Deepspeed and run the pre-trained model for GPT. We strongly scale the workload with eight to 64 nodes with 8 GPUs per node. The application runs with one epoch and 32K iterations, with checkpointing occurring every thousand steps. At each iteration, the workload reads 160 samples.

Intents: The I/O intent for the training dataset is a single intent with 268 million tokens. Each token is 2KB in size and is accessed using the memory mapping (i.e., `mmap`) interface. The token reading during training uses metadata files, which contain sample offsets and sample sizes, and this data is used to load the token. All these I/O operations use `mmap` and, therefore, use OS Page Cache infrastructure to move data asynchronously between the compute node and storage. The checkpointing phase consists of three types of I/O intents: optimization state, layer parameters, and model hyperparameters. Each intent has its own characteristics defined by the number of tensors stored, the size of each tensor, the dimension of each tensor, and the number of files. The values for these characteristics define the "how" of the intents based on LLM parameters such as number of layers, size of pipeline parallelism, size of tensor parallelism, and GPUs per node. Additionally, the megraton deepspeed checkpoints the model parameter per process. This means each process in the LLM has its own checkpoint files with its own parameters and file size. For example, the model's hyperparameters are ≈1MB but the optimization states are ≈2.2GB.

The results for the Megatron Deepspeed workload are presented in Figure 10, with the x-axis showing the number of nodes along with the solution tested and the y-axis showing the I/O time in seconds. We compare the workflow with Lustre and UnifyFS as a Baseline. Lustre changes the overall I/O time from 132 to 148 hr when scaled to 64 nodes. I/O time increases because while reading time is reduced, the checkpoint stage weakly scales and increases the overall time. UnifyFS optimizes both the reading and checkpointing by 1.25× and 3.1×, respectively. Like UNet3D, UnifyFS uses node-local NVMe to buffer the checkpointing data.

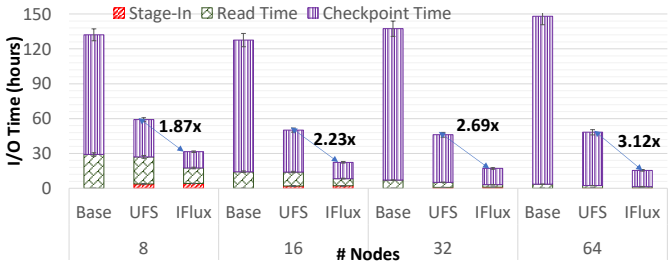


Fig. 10. IFlux optimizes Megatron Deepspeed by 3.12× compared to UnifyFS by utilizing DYAD and SCR accelerators.

The IFlux Mapper assigns the files used for reading to UnifyFS with NVMe accelerators and checkpointing to SCR with shared memory as the cache. A faster hardware accelerator is used as 80-95% of time is dominated by checkpointing. This smarter selection improves the read time by 1.2× and checkpointing time by 2.7× over UnifyFS. Again, both UnifyFS and DYAD can optimize the DL training's dataset reading. However, UnifyFS works more efficiently on datasets with multiple samples per file as compared to DYAD, which is optimized for a single sample per file. Overall, IFlux optimizes the Megatron Deepspeed workload by 1.87× to 3.12× over the UnifyFS solution.

Findings 9: Prioritizing the correct hardware accelerators for important files can maximize I/O performance for workloads.

4.7 Discussion

To make the discussion easier to follow, we organize it into six focused points: system merits and limits, dynamic allocation behavior, complexity reduction, application-specific tuning guidance, sensitivity to inaccurate or incomplete intents, and portability cost across HPC systems.

The main merit of IFlux is that it moves accelerator selection, deployment, and redirection from user scripts into the scheduler path while preserving per-file specialization. This gives users portability across systems with different storage tiers and allows one workflow to use multiple accelerators in one run. The main limitation is that IFlux can only optimize intent classes, interfaces, and accelerators present in its catalogs; unknown behaviors fall back to the PFS until profiling or user feedback adds a new rule.

Dynamic resource allocation is handled at two time scales. Before the job starts, the mapper chooses accelerators from the current catalog and the scheduler-visible resource state. During execution, the deployer monitors health, capacity, and performance, and the executor can conservatively fall back to the PFS or avoid additional caching when an accelerator becomes full or degraded. Future work will extend this to live remapping within long-running jobs.

IFlux reduces I/O-stack complexity by hiding four manual steps: requesting the right storage resource, starting the right software service, configuring application-visible paths or environment variables, and moving data between temporary and persistent tiers. Users describe the file behavior once through intents or profiling-derived annotations; the mapper and executor translate that description into hardware, file-system, middleware, and data-movement actions.

Application-specific tuning remains important, but IFlux turns it into rules rather than one-off scripts. Sequential

file-per-process checkpoints favor SCR or node-local NVMe with asynchronous flush; collective MPI-IO favors UnifyFS over node-local devices; read-mostly training datasets favor stage-in, prefetch, or DYAD depending on file granularity; producer-consumer workflow intermediates favor DYAD; and mixed RAW/WORM phases are split into separate read and write intents when the best accelerators differ.

IFlux's benefit depends on intent accuracy, so we state what happens when an intent is inaccurate or incomplete. A wrong intent makes an optimization either ineffective or, in the worst case, incorrect. It is ineffective when the accelerator no longer matches the true behavior: declaring a file read-mostly enables prefetching, but if the file is actually write-heavy, the prefetch is wasted and performance falls back toward the underlying storage. It is incorrect when a wrong annotation violates an accelerator's semantic contract: declaring a file written once and never updated may laminate it under UnifyFS, so a later write can cause a cache-coherency problem. IFlux limits the first case through conservative PFS fallback for unmapped or invalid intents, and the second through hard satisfiability constraints checked before selection; still, a confidently wrong intent can pass both checks, so automatic and accurate intent capture, from tracing or annotation, remains a first-class requirement rather than an optional hint.

Because we evaluate a single cluster with a fixed accelerator set, we also discuss porting cost. Building the catalogs is largely one-time and cheap: most systems expose one or two storage accelerators with two broad semantics, node-local and shared-across-nodes, and the software is limited and packaged as modules, so a `module` plus `spack` search discovers what is available. For example, Frontier and Tuolumne provide UnifyFS and SCR, while Tuolumne and Corona provide SCR and DYAD. Maintaining the performance catalog through periodic benchmarking is costlier but off the submission path: facilities such as ALCF and OLCF already run IOR weekly during off-hours for health monitoring, and IFlux reuses those measurements to build the performance profiles. The satisfiability rules are the most portable part, since they follow hardware semantics and software capabilities rather than a specific machine: they are unchanged between architecturally similar systems, and a new device mainly adds options. For instance, Tuolumne's rules, despite its added Rabbit accelerator, do not differ from Frontier's; Rabbit only provides more candidate mappings.

The key findings highlight the significant advantages of the IFlux system in optimizing workload performance through efficient accelerator management. The IFlux Mapper's low cost allows schedulers to automatically infer the necessary hardware and software accelerators, leveraging throughput that far exceeds that of traditional HPC schedulers. IFlux can deploy software accelerators to enhance runtime with minimal additional cost and incurs only slight overhead when utilizing storage accelerators through the Flux executor. It can automatically select appropriate accelerators for simulation workloads, achieving remarkable speedups of up to 472× for collective I/O and up to 14.2× for file-per-process I/O compared to Lustre. For deep learning workloads, IFlux optimally chooses hardware and software accelerators across multiple phases, resulting in performance enhancements of up to 15× for reading and 10.8× for checkpointing. Moreover, using varied accelerators for different access patterns can boost data exchange performance for applications like DSS Montage by 2.5×. IFlux's ability to optimize multiple intents simultaneously and its focus on selecting the ideal software for workloads can enhance performance by up to 1.89×. Additionally, prioritizing suitable hardware accelerators for critical files maximizes I/O performance, demonstrating the system's potential to automate optimizations without imposing manual configurations on users.

5 Related Work

Prior systems have explored storage-tier selection and data management for HPC, including burst-buffer file systems, hierarchical buffering, data staging, and workflow-aware placement. DataWarp, BurstFS, Gfarm/BB, UnifyFS, GekkoFS, Hermes, DataSpaces, PDC, Data Elevator, Stacker, UniviStor, HFetch, HReplica, and HCompress each improve some part of the deep memory and storage hierarchy by exposing faster temporary storage, managing data movement, or applying optimization algorithms such as prefetching, replication, and compression [1], [3], [4], [11], [12], [14], [30]–[34]. These systems demonstrate the value of storage tiers, but they generally require the user or workflow developer to choose the data-management layer and configure it for the workload. IFlux is complementary: it uses intent-aware scheduling to select among these kinds of hardware and software accelerators and deploy them as part of the job allocation.

Mimir is the closest prior intent system and is the source of our intent vocabulary [9]. Together with H5Intent [20], it illustrates intent use at lower software layers: Mimir operates at the POSIX I/O layer and H5Intent operates at the HDF5 library layer. These systems optimize behavior within their layer, but their accelerator choices are bounded by the resources already assigned to the job. IFlux differs in scope and control point: it applies intent in the scheduler, combines intent records with cluster accelerator catalogs, and selects plus co-schedules both hardware tiers and software accelerators based on cluster-wide availability. It then emits allocation and placement schemas that deploy and use those accelerators at runtime. Thus, Mimir and H5Intent help express and tune intent within I/O libraries, while IFlux decides which system resources and software services should be allocated for those intents.

Intent-aware networking and cloud scheduling also show that high-level intent can guide resource management, but these systems usually target network policies, cloud placement, or offline workflow scheduling rather than HPC storage accelerators [35]–[38]. IFlux contributes a scheduler-integrated design for online HPC jobs where file-level intent, hardware tiers, software accelerators, deployment cost, and runtime redirection are considered together.

6 Conclusion

This work presented the IFlux scheduler plugin that efficiently uses I/O intents to dynamically and transparently allocate, deploy, and utilize I/O accelerators for scientific workloads. IFlux is inexpensive and introduces low execution overhead, enabling rapid inference and deployment

of required hardware and software accelerators for the target workload. We demonstrate that the IFlux Mapper can efficiently map file-based I/O intents to multiple hardware and software accelerators at a rate of 3.5M intents per second. The high throughput ensures that the mapping overhead is negligible in comparison with scheduling. In our tests based on common I/O benchmarks such as IOR and DLIO Benchmark, IFlux chooses suitable accelerators for simulation workloads and provides speedups of up to $472\times$ for shared files and up to $15\times$ for individual file accesses in comparison with Lustre. IFlux tests on large-scale HPC systems using production scientific workloads such as Megatron Deepspeed and 1000 Genomes demonstrate speedups of $3.12\times$ and $2.85\times$, respectively, compared to state-of-the-art accelerators. IFlux automates the complex, arduous, and error-prone manual mapping, configuration, and deployment of I/O accelerators by taking advantage of I/O intents. The combination of automation and intent-driven scheduling provided by IFlux promises to ease user burden and accelerate complex scientific workflows.

In the future, we plan to expand our work to multi-cluster scheduling, where jobs can be placed in different clusters based on hardware and software accelerator availability for optimal data exchange performance. This will allow users to automatically select the most appropriate cluster for their workload based on their preference for performance, availability, and cost.

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 (LLNL-JRNL-869716). This material is based upon work supported by LLNL LDRD 24-SI-005.

References

[1] 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.

[2] H. Devarajan, A. Kougkas, K. Bateman, and X.-H. Sun, "HCL: Distributing Parallel Data Structures in Extreme Scales," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2020, pp. 248–258, iSSN: 2168-9253. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9229595

[3] 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," *AGU Fall Meeting Abstracts*, vol. 2018, pp. IN34B–09, Dec. 2018.

[4] 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.

[5] H. Devarajan, I. Lumsden, C. Wang, K. Georgouli, T. R. W. Scogland, J. S. Yeom, and M. Taufer, "DYAD: Locality-aware Data Management for accelerating Deep Learning Training," in *IEEE/SBC 36th International Symposium on Computer Architecture and High Performance Computing*, Hilo, Hawaii, USA, Sep. 2024.

[6] T. Patki, D. Ahn, D. Milroy, J.-S. Yeom, J. Garlick, M. Grondona, S. Herbein, and T. Scogland, "Fluxion: A Scalable Graph-Based Resource Model for HPC Scheduling Challenges," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2077–2088.

[7] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Nov. 2008, pp. 1–12, iSSN: 2167-4337.

[8] 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.

[9] 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.

[10] H. Devarajan, "TailorFS: An Adaptive file system to support dynamic I/O requirements of HPC workloads," May 2024, original-date: 2022-06-17T05:36:12Z. [Online]. Available: https://github.com/hariharan-devarajan/tailorfs

[11] H. Devarajan, A. Kougkas, and X.-H. Sun, "HFetch: Hierarchical Data Prefetching for Scientific Workflows in Multi-Tiered Storage Environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 62–72, iSSN: 1530-2075.

[12] H. Devarajan, A. Kougkas, and X.-H. Sun, "HReplica: A Dynamic Data Replication Engine with Adaptive Compression for Multi-Tiered Storage," in *2020 IEEE International Conference on Big Data (Big Data)*, Dec. 2020, pp. 256–265.

[13] 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.

[14] O. Tatebe, S. Moriwake, and Y. Oyama, "Gfarm/BB — Gfarm File System for Node-Local Burst Buffer," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 61–71, Jan. 2020.

[15] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," Mar. 2020, arXiv:1909.08053 [cs].

[16] S. A. Jacobs, M. Tanaka, C. Zhang, M. Zhang, S. L. Song, S. Rajbhandari, and Y. He, "DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models," Oct. 2023, arXiv:2309.14509 [cs].

[17] 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

[18] J.-S. Yeom, H. Devarajan, and I. Lumsden, "DYAD: DYnamic and Asynchronous Data Streamliner," Aug. 2024, original-date: 2019-10-03T23:02:42Z. [Online]. Available: https://github.com/flux-framework/dyad

[19] I. Lumsden, H. Devarajan, J. Marquez, S. Brink, D. Boehme, O. Pearce, J.-S. Yeom, and M. Taufer, "Empirical Study of Molecular Dynamics Workflow Data Movement: DYAD vs. Traditional I/O Systems," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2024, pp. 543–553. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10596488

[20] H. Devarajan, G. Heber, and K. Mohror, "H5Intent: Autotuning HDF5 With User Intent," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 2, pp. 108–119, 2025.

[21] H. Devarajan, L. Pottier, K. Velusamy, H. Zheng, I. Yildirim, O. Kogiou, W. Yu, A. Kougkas, X.-H. Sun, J. S. Yeom, and K. Mohror, "DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. Atlanta, GA: IEEE, Jun. 2024.

[22] 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.

[23] M. Rynge, G. Juve, J. Kinney, J. Good, B. Berriman, A. Merrihew, and E. Deelman, "Producing an Infrared Multiwavelength Galactic Plane Atlas Using Montage, Pegasus, and Amazon Web Services," *Annual Astronomical Data Analysis Software and Systems (ADASS)*, vol. 485, p. 211, May 2014, conference Name: Astronomical Data Analysis Software and Systems XXIII ADS Bibcode: 2014ASPC..485..211R. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2014ASPC..485..211R

[24] Kidney, "2019 Kidney Tumor Segmentation Challenge," 2019. [Online]. Available: https://kits19.grand-challenge.org/

[25] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "HACC: Extreme scaling and performance across diverse architectures," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2013, pp. 1–10, iSSN: 2167-4337.

[26] 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

[27] 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.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, Jun. 2010, pp. 143–154.

[29] 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.

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

[31] T. Wang, A. Moody, W. Yu, K. Sato, and K. Mohror, "An Ephemeral Burst-Buffer File System for Scientific Applications," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. BurstFS; 005217WKSTN00, Apr. 2017. [Online]. Available: https://www.osti.gov/biblio/1351607-ephemeral-burst-buffer-file-system-scientific-applications

[32] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - A Temporary Distributed File System for HPC Applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 319–324, iSSN: 2168-9253.

[33] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: an interaction and coordination framework for coupled simulation workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: Association for Computing Machinery, Jun. 2010, pp. 25–36.

[34] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, "Enabling Transparent Asynchronous I/O using Background Threads," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, Nov. 2019, pp. 11–19.

[35] B. K. Saha, D. Tandur, L. Haab, and L. Podleski, "Intent-based Networks: An Industrial Perspective," in *Proceedings of the 1st International Workshop on Future Industrial Communication Networks*, ser. FICN '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 35–40.

[36] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting Diverse Dynamic Intent-based Policies using Janus," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 296–309.

[37] M. Beshley, A. Pryslupskyi, O. Panchenko, and H. Beshley, "SDN/Cloud Solutions for Intent-Based Networking," in *2019 3rd International Conference on Advanced Information and Communications Technologies (AICT)*, Jul. 2019, pp. 22–25.

[38] A. Campanella, "Intent Based Network Operations," in *2019 Optical Fiber Communications Conference and Exhibition (OFC)*, Mar. 2019, pp. 1–3.



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.



Vanessa Sochat is a Principal Computer Scientist in Livermore Computing. Her work focuses on container technologies, scheduling, and HPC and cloud convergence. She maintains the Flux Operator as part of the Flux Framework, and collaborates closely with industry and several open source communities, leading work and initiative to innovate in the space.



Daniel Milroy is a Computer Scientist at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL). Dan holds a BA in physics from the University of Chicago and a PhD in computer science from the University of Colorado Boulder. His research focuses on graph-based scheduling, resource representation, and management for HPC and cloud-converged environments. While Dan's research background is numerical analysis and software quality assurance and correctness for climate simulations, he is currently interested in scheduling and representing dynamic resources for HPC and cloud.



Thomas Scogland Thomas R. W. Scogland is a computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research centers around large scale resource management in heterogeneous systems, the design and development of runtime systems and programming models for scientific applications dealing with heterogeneous and many-core architectures, heterogeneous memory systems, and innovating in the construction of application binaries. He is the research lead and scheduler lead on the flux resource manager project, a developer on the RAJA framework libraries, lead on the CAMP and DESUL library projects, chair of the OpenMP accelerator subcommittee, LLNL WG21 C++ committee representative and participates in numerous other research and development efforts. Tom holds a B.S. in Computer Science from Purdue University, and an M.S. and PhD in Computer Science from Virginia Tech.