Bridging HPC Data Gaps: Novel Tools for GPU Resource Monitoring and Scheduler Emulation

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Walter Jay Ashworth

May 2025

Acknowledgments

I would like to thank Dr. Michela Taufer for being a wonderful advisor and guiding my research with insight and support, and I sincerely thank my committee members, Dr. Jack Marquez and Dr. Catherine Schuman, for taking the time to review and help me refine my thesis.

I am also grateful to the members of the GCLab for fostering a welcoming and enriching environment throughout my time in the lab. Specifically, I would like to thank my seniors, Ian Lumsden, Nigel Tan, and Paula Olaya, for answering my many technical questions and reviewing countless written pieces up until now.

I also thank the AI-BD team at the Pittsburgh Supercomputing Center for their support during my summer internship. From PSC, I would specifically like to thank Julian Uran for mentoring me during my time there.

Finally, I appreciate my collaborators at LLNL from the performance, scheduling, and descriptive expression FRACTALE thrusts, as well as the Flux coffee hour attendees, for their valuable feedback and guidance. Specifically, I would like to thank Tapasya Patki, Daniel Milroy, Mark Grondona, Olga Pearce, Stephanie Brink, Dewi Yokelson, Jim Garlick, and Tom Scogland for guiding me as I learned all things Flux to contribute to the Flux Emulator.

The work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 24-SI-005 (LLNL-TH-2003856).

This material is also based upon work supported by Neocortex, through funding provided by the National Science Foundation under Grant #2005597. This work used on Bridges-2 at Pittsburgh Supercomputing Center through allocation sys890003p from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

Abstract

The evolution of High-Performance Computing (HPC) systems is introducing ever greater complexity in resource management and performance monitoring, creating critical gaps in data availability. With their expanding significance in AI and data-intensive applications, GPUs now serve as pivotal components of HPC workloads. Simultaneously, the push toward exascale computing necessitates increasingly efficient and scalable scheduling policies. This thesis presents two novel tools designed to address the data availability challenges introduced by these ongoing shifts in modern HPC environments. The first tool bridges a crucial gap in per-job GPU resource monitoring for SLURM-managed clusters, which lack this support natively. By enabling detailed post-job analysis of GPU utilization metrics, it allows researchers to identify underutilization issues—ranging from configuration errors to algorithmic inefficiencies—while helping administrators accurately assess resource usage in planning future upgrades. The second tool, the Flux Emulator, builds upon a preliminary prototype and extends the capabilities of the Flux Framework, a cutting-edge resource management and scheduling system tailored for exascale HPC. Through the simulation of historical job workloads, the emulator enables the evaluation of various scheduling policies and strategies without the need for physical cluster resources. By illuminating the effects of different scheduler configurations on performance metrics such as job makespan, it empowers system software developers to refine algorithms and policies for more efficient utilization of emerging exascale systems. By providing detailed GPU resource monitoring within

established scheduling systems and offering a scalable emulator for evaluating multiple scheduling policies, this thesis lays the groundwork for a more data-driven approach to HPC resource utilization and optimization. Together, these tools directly address the critical gaps in data availability and performance insight that arise as HPC environments continue to grow in complexity.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Paradigm Shifting

High-performance computing (HPC) systems are experiencing a paradigm shift due to increasing architectural heterogeneity and the establishment of exascale computing. The shift builds on the use of accelerators (e.g., GPUs) that define an HPC ecosystem, and they are essential for compute-intensive workloads, including scientific simulations and training of artificial intelligence (AI) models. As this shift unfolds, new resource monitoring and system evaluation gaps have emerged, hindering efforts to optimize performance and ensure efficient resource use. This thesis addresses two critical challenges for this evolving landscape: the lack of fine-grained GPU resource monitoring in widely used resource and job management systems (RJMS) such as SLURM [20], and the need for scalable, realistic testing environments to evaluate modern scheduling policies on next-generation HPC platforms like those managed by the Flux Framework.

1.2 GPU Monitoring for SLURM Jobs

As GPU computing becomes increasingly embedded in scientific and engineering workflows, researchers and system administrators face a growing need for transparent

and precise monitoring of GPU usage per job. However, despite their widespread adoption, major scheduling systems such as SLURM do not natively provide the capability to attribute GPU utilization to specific jobs. This lack of granularity hampers performance analysis and debugging efforts, particularly for researchers who rely on pre-existing GPU-accelerated libraries or third-party tools.

This gap exacerbates resource planning challenges for system administrators: underutilized GPUs are difficult to detect, and queues appear artificially long, prompting costly hardware expansions that may not address the root inefficiencies. Recognizing this critical bottleneck, we developed a tool that seamlessly integrates with existing software infrastructures, such as Open XDMoD [13], to collect detailed GPU usage data tied directly to individual SLURM jobs. Our minimally invasive solution enables retrospective analyses of completed workloads, empowering users to diagnose inefficiencies ranging from poor load balancing to suboptimal GPU code execution while giving administrators a robust data foundation to inform capacity planning and system optimization.

1.3 Flux Performance through the Flux Emulator

Beyond resource monitoring, the transition to exascale introduces unprecedented complexity into HPC scheduling. Modern resource managers must support hierarchical, heterogeneous systems that combine CPUs, GPUs, and emerging resource types like Rabbit nodes [18]. The Flux Framework [1], developed to meet these demands, introduces a scalable and highly modular approach to job scheduling and resource management, tailored for contemporary platforms such as the LLNL El Capitan supercomputer [17]. However, validating and refining scheduling policies in such systems is not trivial. Testing directly on production hardware is often impractical due to limited access, high operational costs, and the risk of disrupting mission-critical workloads. Moreover, traditional simulators often fall short, limited by their

RJMS [12] or requiring labor-intensive modeling efforts to reflect the intricacies of ever-evolving HPC architectures [11] [16] [4].

To mitigate these challenges, this thesis presents the Flux Emulator, a novel tool that extends the Flux Framework by simulating job workloads using historical traces and the native scheduling logic of Flux itself. Using actual Flux components rather than abstracted models, the emulator provides more realistic insights into how the proposed scheduling policies will perform under representative workloads. This approach eliminates the need for extensive manual modeling, reduces the risk of non-reproducibility in test scenarios, and enables the controlled evaluation of scheduling strategies without impacting production systems. The Flux Emulator provides a crucial bridge between experimental simulation environments and real-world HPC systems, ensuring that scheduling policies are rigorously tested and validated before deployment at scale.

1.4 Thesis Contributions

The contributions of this thesis are twofold to fill the gaps that hinder the optimal utilization of modern HPC systems. We specifically address the data monitoring gap by:

- Designing and implementing a tool and methodology to collect and analyze per-job GPU utilization data on SLURM-managed clusters, addressing limitations in native resource tracking frameworks.
- Validating the effectiveness of this solution through real-world experiments and proposing a user-oriented interface for visualizing and interpreting GPU resource utilization data.

We address the gap in scheduling evaluations by:

- Enhancing and extending the Flux Emulator to support modern Flux Framework implementations, including the integration of the Fluxion Scheduler [14] for advanced scheduling policy evaluation.

- Demonstrating the effectiveness of the emulator through key use cases, showcasing its ability to analyze makespan reduction and improve resource utilization efficiency in realistic HPC scenarios.

1.5 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we outline SLURM and Flux, which are the foundational resource managers for the thesis. Chapter 3 delves into our analysis of SLURM execution monitoring, highlighting existing shortcomings in GPU monitoring and our corresponding solution. Chapter 4 details our Flux Execution modeling, focusing on contributions from the Flux Emulator's design. Chapter 5 discusses our experimental evaluation of the Flux Emulator, demonstrating its minimal overhead, capacity to test various policies, and effectiveness in replicating real system executions. Chapter 6 ends this thesis with a summary and future work.

Chapter 2

SLURM and Flux Overview

2.1 SLURM in a Nutshell

SLURM is a resource and job management system (RJMS) with a centralized design consisting of three daemons, `slurmctld`, `slurmd`, and (optionally) `slurmdbd` [8], which each provide a set of unique services. First, `slurmctld` is the primary controller daemon and is used for management tasks such as job management and resource pool scheduling [7]. There is usually only one instance of `slurmctld` within a cluster, but multiple instances can be deployed for fault tolerance. `slurmctld` does not run any jobs, and instead executes them remotely on independent `slurmd`. Second, `slurmd` is a daemon that is used to run programs on a cluster and is controlled by requests from `slurmctld`. There is usually one instance of `slurmd` on each compute node of a cluster. `slurmd` will respond to periodic requests from `slurmctld` for the status of the node (e.g., CPU load, memory usage) and job execution progress. Finally, `slurmdbd` is a daemon that collects and stores historical data on how jobs are scheduled and executed. `slurmdbd` is usually installed on its own node, separate from `slurmctld` and `slurmd`. The collection of these historical data is called accounting and is one of the primary focuses of Chapter 3.

SLURM provides two commands for users to interact with data recorded using `slurmdbd`: `scontrol` and `sacct`. Users employ `scontrol` to view or modify the configuration and state of actively running jobs, and it cannot be used with jobs that have ended. Then, users apply `sacct` to directly query historical job data from the accounting database. After a job is completed, `sacct` is able to provide the user with new metrics that are not accessible through `scontrol` (e.g., CPU time, exit codes, and start/end timestamps).

2.2 Flux in a Nutshell

Flux is a RJMS characterized by its modular design, allowing service deployment through reactive programming. Flux's structure revolves around three essential components: the Flux brokers and broker modules, the tree-based overlay network (TBON), and the key-value store (KVS). A Flux instance constitutes the operational environment for the Flux components, typically executing on a collection of resources, whether an allocation or a full cluster. As shown in Figure 2.1, this graphical representation depicts how the Flux components function cohesively in a Flux instance. A Flux instance consists of one or more Flux brokers that configure an overlay network to facilitate distributed communication. Flux components engage in communication through this configuration. Within Flux, various services, termed Broker Modules, are provided. These include the Job Manager, Resource Manager, Job Executor, along with Schedulers like the Fluxion and sched-simple schedulers, which are prominently addressed in this study. Users also have the flexibility to design broker modules (i.e., ad hoc services tailored to specific requirements).

2.2.1 Flux Brokers and Broker Modules

Brokers are deployed on some subset of resources (e.g., a single node of a cluster or a single core) managed by the Flux instance and used to communicate with other

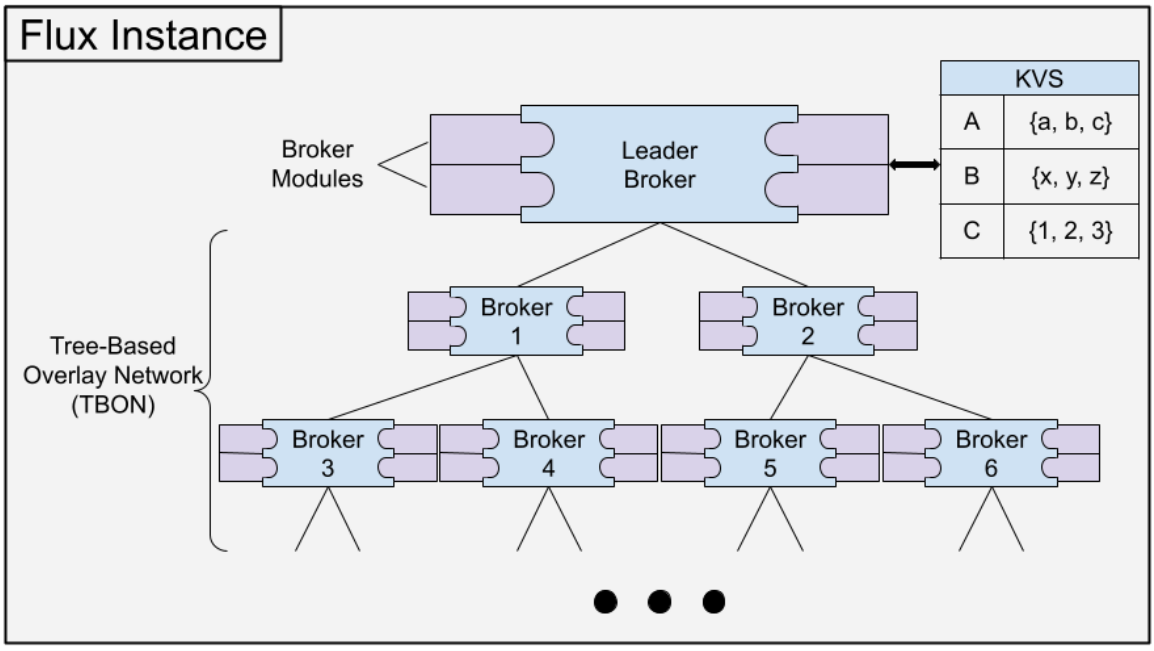


Figure 2.1: Key Components of the Flux Framework

brokers. Flux brokers use the ZeroMQ messaging library to facilitate communication between other brokers.

Flux Broker Modules plug into the brokers and utilize the functionality of Flux brokers. In essence, they provide the services that make Flux useful. In this work, we target five existing broker modules that are included with Flux: the job manager module, the resource module, the job exec module, the sched-simple scheduler module, and the Fluxion scheduler module.

The job manager orchestrates the life cycle within Flux through a state machine [5]. It also records transitions between job states in KVS, and it is extensible through plugins known as jobtap.

The resource module maintains an inventory of resources available in the Flux instance, provides this information to other modules, and verifies that resources exist. The resource inventory within the resource module is known as a resource set [6].

The job exec module is responsible for launching jobs on scheduled resources. It will handle requests for job execution from the job manager and respond to those requests when a job is completed.

The sched-simple scheduler module is included in Flux for less complex scheduling tasks. Sched-simple follows a priority-based scheduling policy with a node-based resource representation that is similar to SLURM. Sched-simple does not include any advanced scheduling techniques, such as backfilling.

Finally, Fluxion [14] is the primary scheduler for Flux. Fluxion moves away from node-centric representations of resources as is used in traditional HPC schedulers by modeling resources using a directed graph. This graph-based representation of resources allows for more effective scheduling of accelerators and dynamic resources such as the "Rabbit" [18] nodes on El Capitan. This resource representation is an open area of research. In addition to its unique resource representation, Fluxion includes the option to use one of four different queuing policies: first-come-first-serve (FCFS), conservative backfilling, easy backfilling, and hybrid backfilling.

2.2.2 Tree-based Overlay Network (TBON)

In a Flux instance, the Tree-based Overlay Network or TBON organizes brokers into a hierarchical network, positioning a 'leader broker' at its apex. This arrangement allows the TBON to manage the routing of communications among brokers. The configuration of the TBON is established at the start of a Flux instance.

2.2.3 Key-value Store (KVS)

The Key-Value Store or KVS serves as a broker's cache, utilizing a key-value data architecture. This cache might, for instance, contain a description of system resources. Every broker possesses its own local KVS. These local caches are transmitted up the hierarchy from child to parent brokers, culminating in the Leader broker, which maintains a comprehensive version of the KVS. When there is a need to exchange

data between two distinct brokers, they access the necessary information via their mutual parent broker by utilizing the TBON.

2.2.4 Reactive Programming

Reactive programming represents a specialized form of event-driven programming. Here, alterations in data states can induce code execution that spread throughout the application. In the context of Flux, it facilitates the coordination of interactions among broker modules. The operation of reactive programming in Flux is based on three key elements: watchers, remote procedure calls (RPC), and the Flux reactor. Watchers are pivotal software units that monitor data states, activating when these states meet specific criteria. Furthermore, RPC facilitates state communication among different broker modules. Lastly, Flux's reactor manages the code responses whenever watchers are triggered.

2.3 Flux in Action through a Job Life Cycle

Flux manages the job life cycle, from submission through completion, through its main components and broker modules. This life cycle can be comprehensively described in six distinct steps. Refer to Figure 2.2 for a detailed depiction of the job lifecycle within Flux, highlighting the specific components and broker modules active throughout each stage of the process.

Step 1: Flux is initialized by employing the resource module to generate a resource description from a system configuration file. This description is then saved in Flux's internal key-value store (KVS). Subsequently, the scheduler retrieves this stored description from the resource module.

Step 2: At the point of job submission, the user initiates their task by entering commands in Flux, such as `flux run` or `flux submit`. Upon receipt, Flux directs

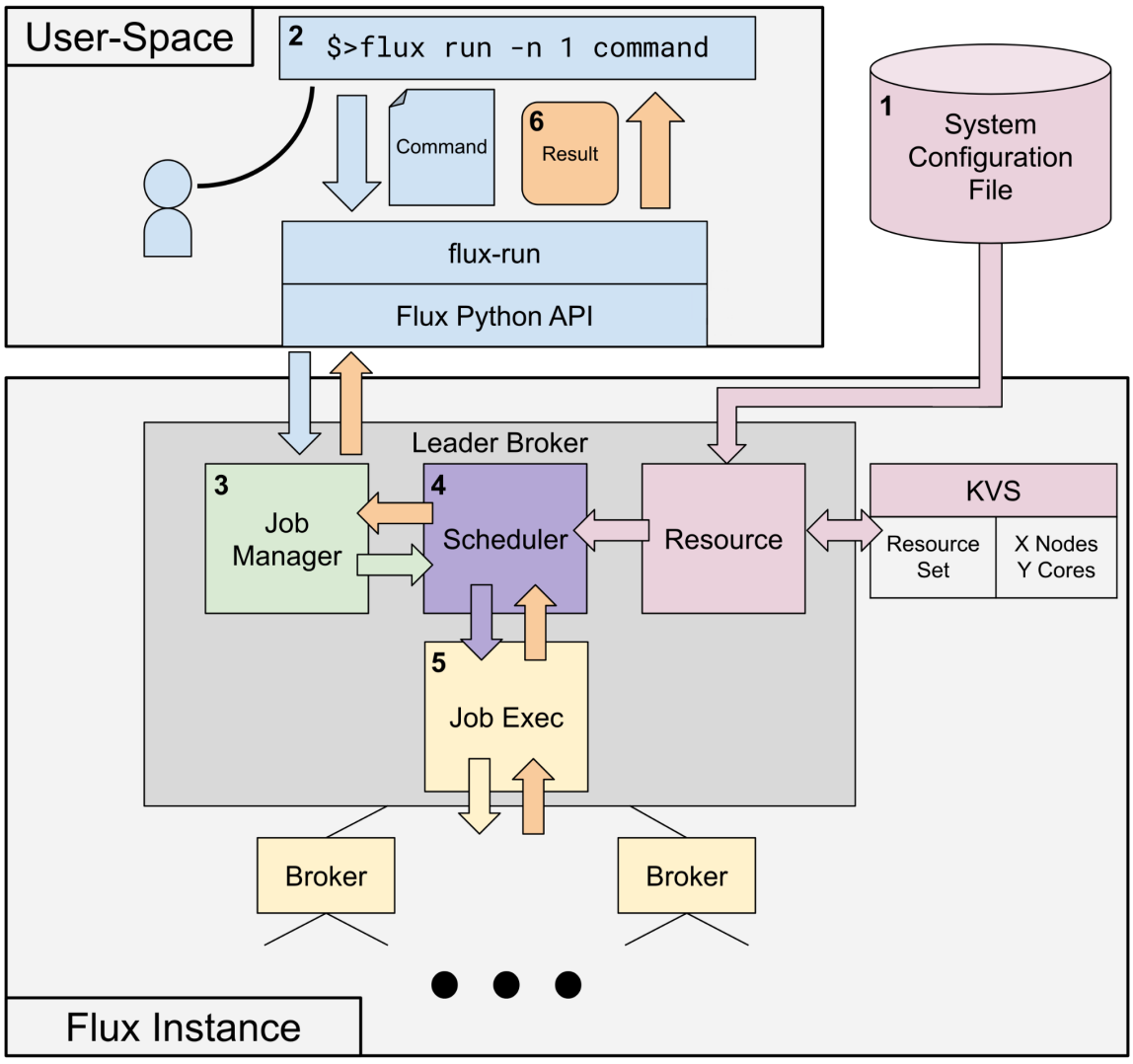


Figure 2.2: The job life cycle in Flux.

these tasks to the job manager, responsible for managing and coordinating the entire lifecycle of the job.

Step 3: To validate the job, the job manager employs a state machine for orchestration. Once the job has been validated, the job manager proceeds to dispatch an allocation request concerning the job to the scheduler.

Step 4: The task's scheduling involves its placement into the scheduler's priority queue. Should the necessary resources currently be in use, the task will remain in a waiting state until those resources are free. Once the resources are liberated, the scheduler will assign them accordingly and inform the job manager of the successful allocation. Following this, the task is transferred to the scheduler's running queue.

Step 5: Execution of the task involves a process where the job manager dispatches a request to the job execution module, which proceeds to initiate the task using the designated resources. The task is then carried out to its full completion or until a timeout occurs, depending on what happens initially.

Step 6: Upon job completion, the results are promptly delivered to the user by Flux. Subsequently, the job execution process informs the job manager that the task has reached completion, and it's necessary to release the allocated resources. Following this, the scheduler proceeds to free up these resources, making them available for subsequent tasks.

Chapter 3

Monitoring SLURM Executions

3.1 Limitations of XdMoD and SLURM

NSF-funded ACCESS HPC clusters, such as Pittsburgh Supercomputing Center's Bridges-2 and Neocortex [2] [3], use Open XdMoD as a primary tool for resource monitoring. However, XdMoD lacks per-job GPU usage tracking, a generalized method for analyzing resource usage over time for specific jobs, and suffers from a one- or more-day delay between job execution and data availability. This problem is further compounded because XdMoD's data ingestion method (Figure 3.1) depends on allocation data from the SLURM accounting database. However, SLURM accounting does not provide any way to collect historical GPU allocation data outside of the number of devices allocated. Consequently, it is not possible for XdMoD to determine how GPUs are used by jobs, despite having access to GPU utilization at node level through Performance Co-Pilot (PCP), a program used to collect time series data from nodes in a cluster.

This gap forces users, often domain scientists, to develop their own resource tracking methods, which can lead to the underutilization of GPUs. A common example of this is when a user requests multiple GPUs but only uses a single one due to a misconfiguration in their software. Without the ability to easily view their resource

usage, these users will launch many jobs with wasted GPUs. This underutilization, which can account for up to 15% of allocated GPU hours [9], results in longer queue times for GPU nodes. Without the ability to track GPU underutilization, system administrators are prompted to expand existing clusters.

3.2 Two-Phase Approach for Improving GPU Monitoring

We address this GPU monitoring gap in current ACCESS computing centers through a two-phase approach. In the first phase, we develop a data ingestion method to gather per-job GPU data using readily available software tools and an additional live index parsing script, which provides critical insights into everyday underutilization issues. In the second phase, we visualize data collected using our prototype data ingestion method and notify users of low utilization to make users and admins more aware of poor resource usage. We plan to integrate this model into XdMoD and extend it to the broader community with an emphasis on ACCESS systems. We are piloting our model on Bridges-2 to test its reliability, evaluate its effectiveness, and make necessary adjustments before advocating for broader implementation.

3.2.1 Phase I: Data Ingestion Layer

In the first phase, depicted in Figure 3.2, our data ingestion method leverages DuckDB for data aggregation, combining inputs from SLURM accounting, live device index parsing, and PCP for time series data. Our methodology is compared to Jobstats [15], a project with a similar function that collects GPU allocation data through prolog and epilog scripts. Instead of modifying SLURM's prolog and epilog scripts, we actively collect per-job GPU allocation data using SLURM's `scontrol` at fixed intervals, which are stored in our database. This data provides the mapping necessary to determine how individual jobs use their GPUs. Our approach provides three key

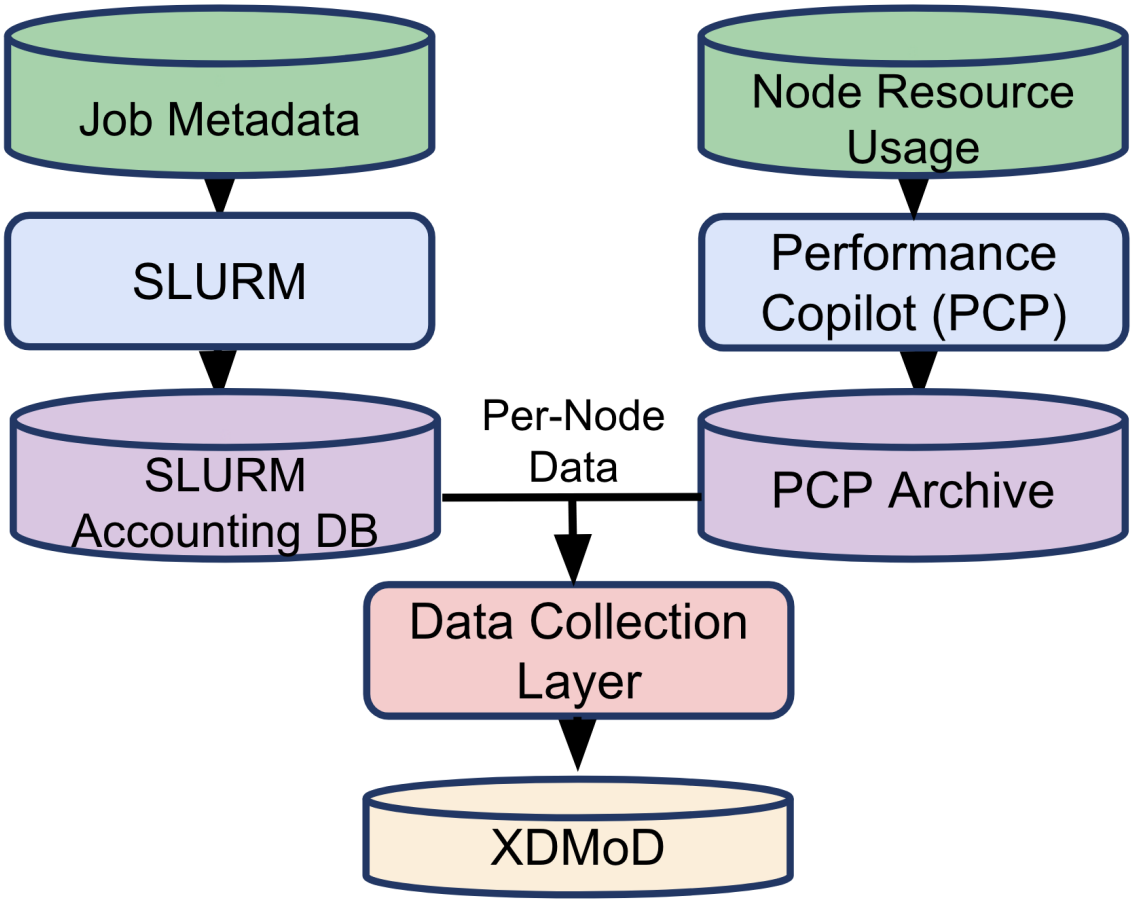


Figure 3.1: Data ingestion with XdMoD.

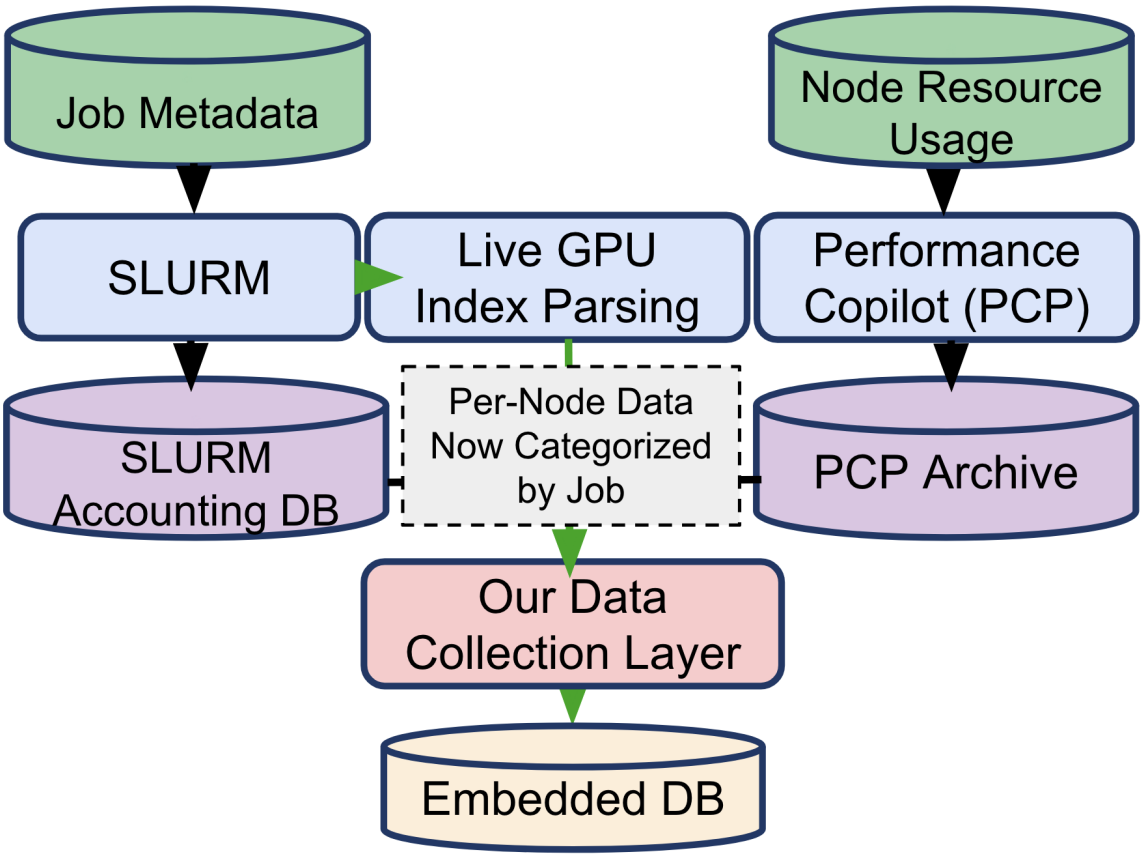


Figure 3.2: Our Model of Data Ingestion (Phase I).

advantages over Jobstats. First, it offers a simpler and more flexible data ingestion model through the use of a single Python script. Second, since we do not modify the prolog and epilog scripts, we can reduce administrative burden. Finally, we leverage the same underlying data collection software as the XdMoD team (i.e., SLURM accounting and PCP), ensuring a smooth rollout without additional system tools.

3.2.2 Phase II: User-Facing Platform

In the second phase, which is still in progress, we design a prototype of a user-facing platform to display per-job resource data in interactive plots and notify users of underutilization (Figure 3.3). In figure 3.4, we show the key features of this prototype, including the interactive usage plot 3.4a and a matrix 3.4b for users to quickly view the utilization of resources in their jobs. This platform provides actionable insights to correct configurations, optimize framework settings, and improve resource management. Initial findings from phase one give us confidence that our user-facing platform can produce noticeable improvements in system GPU utilization and queue wait times, significantly enhancing overall system efficiency.

3.3 Performance Insights from GPU Utilization Data

We assess the performance of our data ingestion method by creating a data set of GPU jobs on Bridges-2 to identify patterns of poor utilization among users. From 9,224 unique jobs collected between June 18, 2024 and July 9, 2024, we analyzed 5,717 completed jobs that ran for over 30 seconds. Using this data set, we identify patterns of inefficient device utilization by calculating the average device usage per job, focusing on jobs where the average usage was under 5%. This calculation considers both the average GPU utilization and the total time the device was allocated. We sort users according to their total number of underutilized device hours, as shown in

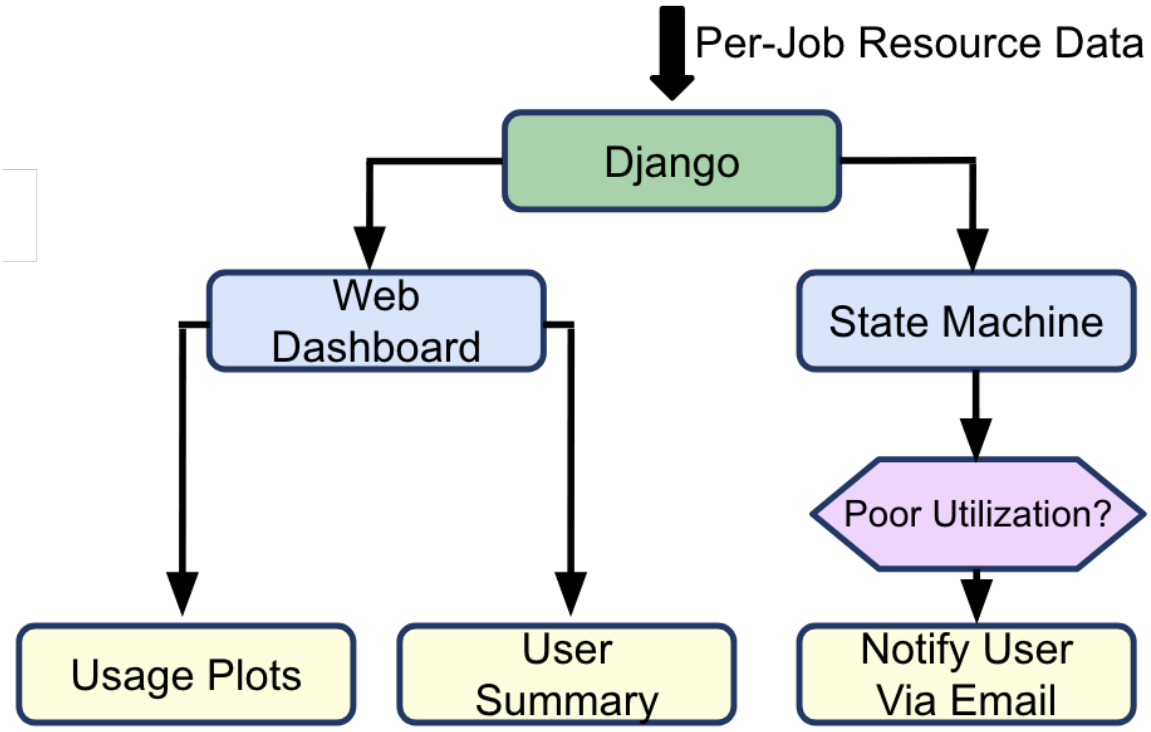
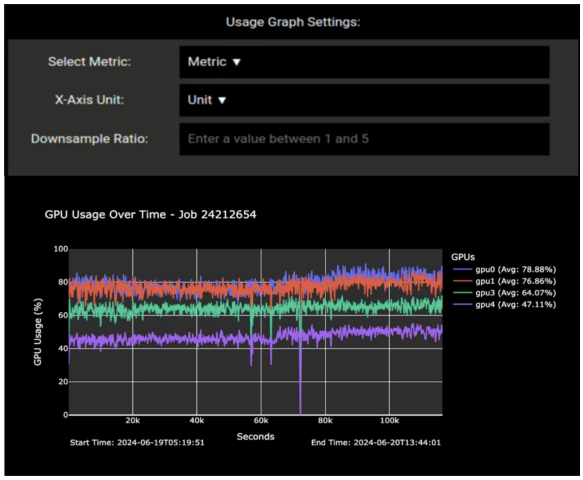


Figure 3.3: Model for our user-facing platform

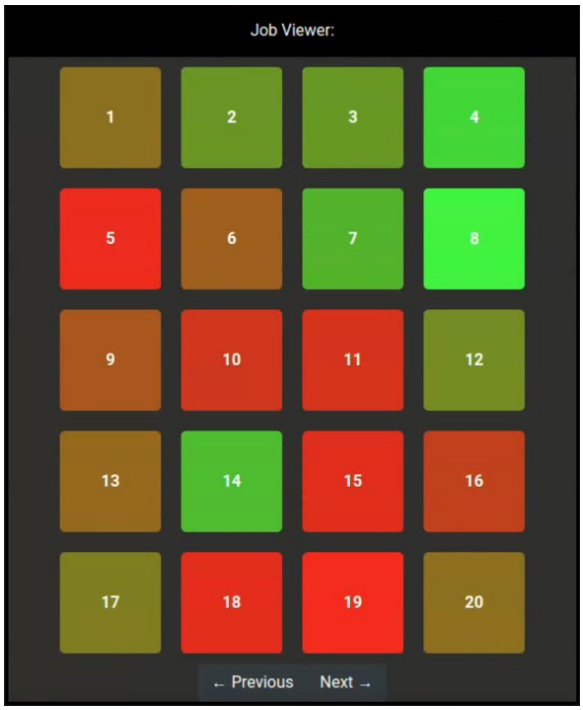
Figure 3.5. Furthermore, we classify device utilization into four levels: 0% (no usage), below 33% (low usage), 33% - 66% (moderate usage), and above 66% (high usage).

Our analysis reveals that, of 377 unique users, only 10 users account for more than 97% of the underutilized device hours. Considering the small number of users contributing to underutilization, we investigate the batch scripts of the 10 users and identify three common issues among them:

- **Batch configuration oversight**: Only a small subset of allocated GPUs is used due to omitted lines in batch scripts.
- **Framework misconfigurations**: User programs do not distribute tasks across multiple GPUs as intended because users input incorrect settings.
- **Low GPU utilization on many jobs**: Multiple concurrent jobs with less than 5% GPU usage could run on a single GPU instead of being spread across an entire node.



(a) Interactive Usage Plot



(b) User Job Explorer

Figure 3.4: Key features from our prototype user-facing platform

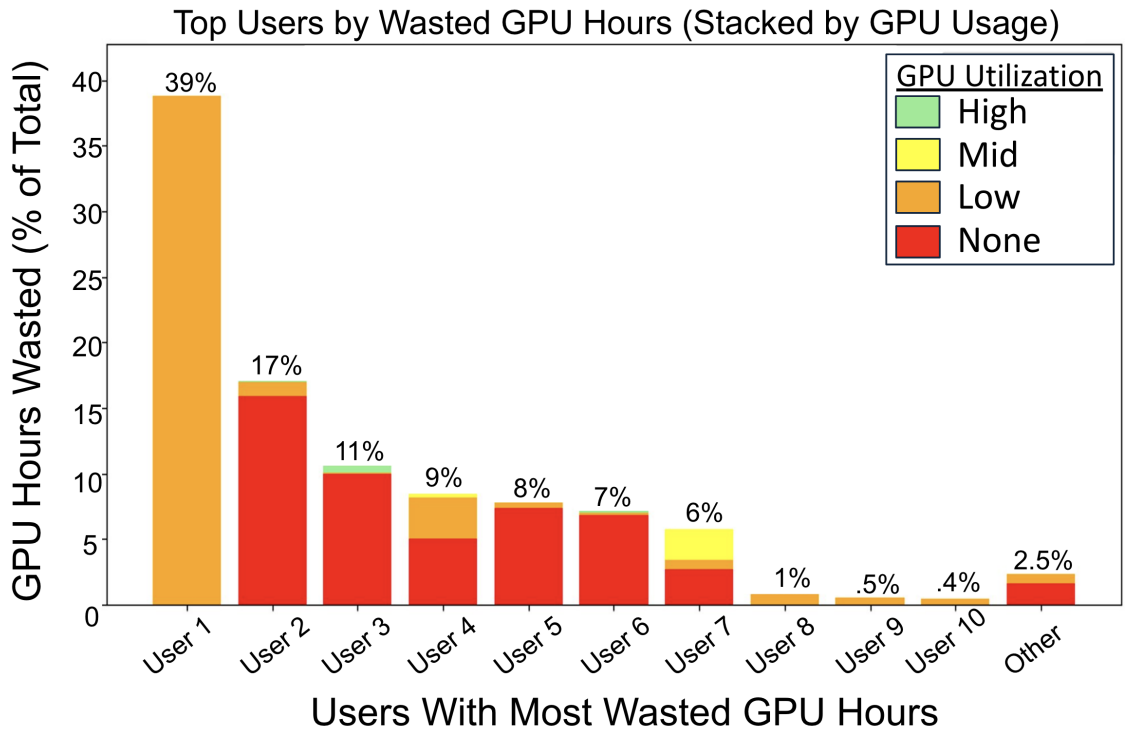


Figure 3.5: Distribution of inefficient time utilization on devices.

These findings reveal that users experiencing the first two issues are likely not using any form of resource monitoring because such problems are easily solvable. Users who fall into these easily solved categories account for over 57% of all wasted GPU hours during the observation period. As a result, we are confident that the introduction of our data ingestion method and a user-facing platform would significantly reduce GPU underutilization.

3.4 Lessons Learned and Next Steps

Our solution impacts both system administrators and end users. It fills a crucial gap in existing tools like XdMoD by enabling timely per-job GPU usage monitoring and analysis on ACCESS systems. Our contributions include a flexible data ingestion method that avoids altering SLURM configurations and utilizes existing PCP infrastructure, and the design of a user-facing platform with notifications and

a web dashboard for improved resource management. Early results are promising, showing potential for reducing GPU job wait times, enhancing system utilization, and optimizing existing resources to support future growth.

In the future, we will finalize the implementation of our user platform to measure the system's impact on underutilized device hours and queue times. We plan to pilot our model on Bridges-2 so that we can test reliability, evaluate effectiveness, and make necessary adjustments. Once refined, we can integrate our solution into Open XdMoD, thereby extending it to other ACCESS systems.

Chapter 4

Modeling Flux Executions with the Flux Emulator

4.1 Limitations of the Existing Flux Emulator

The Flux Emulator originally existed as an extension of the Fluxion scheduler. It was used to study the introduction of network bandwidth and runtime prediction into scheduling policy [19]. The existing Flux Emulator migrated from Fluxion to flux-core in 2019. However, the implementation was largely undocumented and unmaintained. Since this migration, flux-core was updated to v0.71 while the emulator only supported flux-core v0.14. Additionally, Fluxion support was never reintroduced to the emulator. As a result of these limitations, the emulator has not been used since 2019.

To account for more than five years of Flux's development, we make a number of changes to upgrade the Flux emulator. First, the new emulator maintains the same functionality as the previous version from the user's perspective, but it now includes integrated support for the Fluxion scheduler. Second, we enhance input verification to be stricter (e.g., type-checking input to prevent Flux errors). Third, due to the introduction of the resource module, we modify the process

of emulating resources by generating a Flux resource set, storing it in the KVS, and restarting the resource module and scheduler for the changes to take effect. Fourth, we update synchronization methods between the emulator and Flux to accommodate modifications to job event reporting and the underlying scheduler libraries. Specifically, we adjust the internal loop logic within the emulator to shift from job state reporting to job transition reporting. Additionally, we utilize a common scheduling library, `libschedutil`, shared by Fluxion and sched-simple, to extract scheduler information. Because the relevant section of `libschedutil` was refactored, we adapt our implementation accordingly. Finally, we revise the code to reflect other minor refactoring changes between Flux versions. We remove no functionality in this updated emulator.

4.2 Redefining the Flux Emulator for Modern HPC Systems

The Flux Emulator is designed to emulate job execution within Flux by reusing, extending, and preserving Flux's core components. With the modifications to the previous emulator complete, we detail the changes that we make to Flux through the Flux Emulator. First, we replace the Flux submission commands (e.g., `flux run`, `flux submit`) with the `flux emulator` command that uses the Flux Python API to track jobs through their life cycle and interact with various components of a Flux instance. Second, on startup, the `flux` emulator command overrides the resource module's resource set with one obtained from the user. This allows users to emulate the scheduling of jobs on any number of resources. Third, sim-exec replaces job-exec, and it works with the Flux Emulator to fast-forward over the job's runtime. Finally, we make enhancements to the job manager (job-sync additions) that allow the emulator to track the scheduler's state and job start times. To support job

manager enhancements, we introduce a callback mechanism to Fluxion to determine if it is busy or idle.

4.3 Flux Emulator in Action through a Job Life Cycle

To further explain how the Flux Emulator works, we revisit the six steps of Flux's job life cycle and explain how the emulator modifies these steps. Figure 4.1 depicts the job life cycle in the Flux Emulator and the components involved.

Step 1: In the initialization step, the emulator generates a fake resource set and stores it in the KVS. Then it restarts the resource module and the scheduler. The resource module ingests the fake resource set, and the scheduler acquires it from the resource module. At this stage, the emulator also replaces job exec with sim exec.

Step 2: In the submission step, the user submits a list of historical job traces and a desired resource configuration to Flux using the command `flux emulator`. The emulator establishes communications with the job manager through the Flux Python API. The emulator then processes the job traces and submits them to Flux.

Step 3: The job validation step proceeds as normal except the job manager blocks the emulator until a job has started and the scheduler is idle.

Step 4: In the scheduling stage, jobs are scheduled using the fake resource set. Whenever all jobs are scheduled or the available resources become full, the scheduler notifies the job manager that it is idle, and the job manager unblocks the emulator.

Step 5: In the job execution stage, the job manager sends a request to sim exec to run the jobs. When notified that the scheduler is idle, sim exec skips time to the end of the job with the shortest runtime. Sim exec does nothing until time is skipped. At that point, the emulator uses sim exec to send a signal to Flux that the job is finished and that resources need to be released.

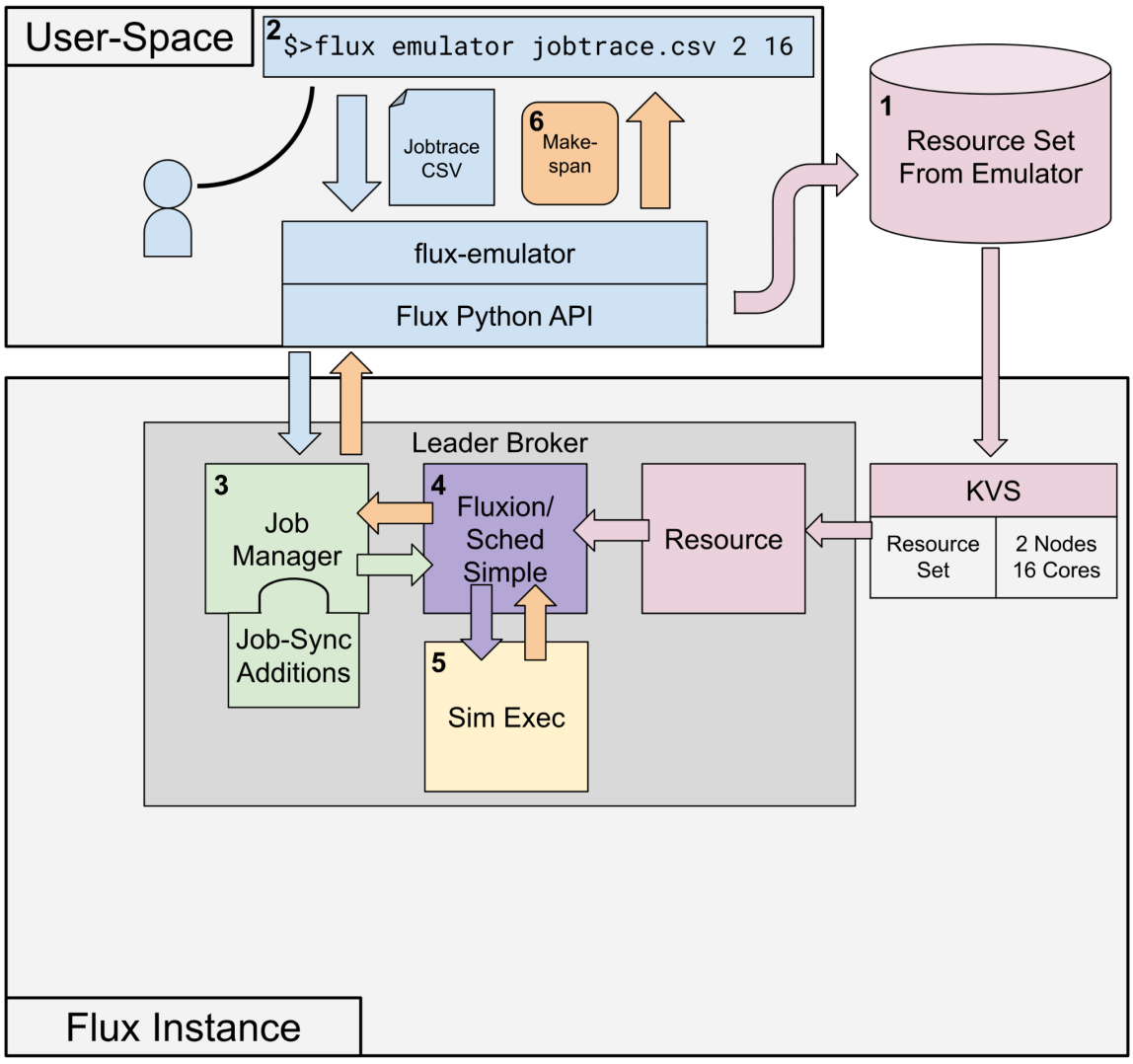


Figure 4.1: The job life cycle in the Flux Emulator

Step 6: Finally, in the conclusion stage, the emulator checks if all jobs are completed. If there are still jobs remaining, the emulator returns to step 4. When all jobs are complete, the emulator outputs statistics about all jobs run including the makespan and used core-hours.

4.4 Lessons Learned and Next Steps

Through updating the Flux Emulator to use the latest version of Flux, we learn that continuous maintenance of code and thoughtful software design are crucial (and often overlooked) when developing research tools. The constant evolution of frameworks like Flux quickly renders tools that use these frameworks obsolete unless overhauled. This process can be very lengthy and results in precious research time being spent rehashing old problems instead of solving new ones.

To facilitate future maintenance of the Flux Emulator, we plan to upgrade it in three key ways. First, we will consolidate the fragmented components of the Flux Emulator into a single jobtap plugin. This allows us to maintain the same level of functionality as the current emulator while providing a clearer and more maintainable code base. Second, we will add in-code documentation for each of the functions used, making it easier for others to understand, maintain, and modify the code in the future. Finally, we will integrate the emulator into existing CI/CD systems found in flux-core, allowing us to quickly track and resolve future code changes that cause the Flux Emulator to break. By applying these changes to the Flux Emulator, we will simplify maintenance and remove the need to overhaul the code each time it is used.

Chapter 5

Experiments with the Flux Emulator

5.1 Testing Overview

To validate and measure the effectiveness of the Flux Emulator, we perform three sets of experiments: correctness tests using the old emulator as a baseline, overhead measurements, and a scheduling policy assessment with comparison to a real system.

We perform our experiments on Tuolumne [10] which has Flux as its system RJMS and the same hardware architecture as El Capitan (with fewer nodes). Additionally, we use Fluxion v0.42.2 and flux-core v0.71.0, both of which are installed on Tuolumne.

5.2 Emulator Correctness Test

In this experiment, we use two tests with predictable results to ensure that the emulator produces the same correct results in three iterations of the emulator: the old emulator, the new emulator using sched-simple, and the new emulator using Fluxion and a first-come-first-serve (FCFS) scheduling policy.

Table 5.1 shows the results for test 1 and test 2 across the three iterations of the Flux Emulator. In test 1 we use 10 jobs, each lasting 6 minutes and requiring 1 node, and we emulate a system with 1 node. The expected result is a makespan of 1 hour and a system utilization of 100%. In test 2 we use 10 jobs, each lasting 6 minutes and requiring 10 nodes, and we emulate a system with 30 nodes. The expected result is a makespan of 0.4 hours and a system utilization of 83.33%. In line with our expectations, we observe that all three iterations produce identical results for both test 1 and test 2. Based on these results, we conclude that the new emulator produces the same correct result as the old emulator with both sched-simple and Fluxion.

5.3 Overhead Measurement

This experiment measures the throughput (jobs/second) of Flux, which is heavily influenced by the performance of Fluxion, to ensure that the emulator does not introduce unnecessary overhead. To do this, we reconfigure Flux within our allocation so that it believes it has 10,000 nodes and submit jobs to Flux using a test program with a run-time of 0.1 seconds. We make two runs of this experiment using Fluxion alone and Fluxion with the emulator: one with 1,000 jobs and one with 10,000. Furthermore, we repeat each test 20 times to account for variance.

Table 5.1: Comparison of correctness between the old and new emulator.

Test Case	**Emulator**	**Scheduler**	**Makespan (hours)**	**System Utilization (%)**	**Job Execution Order**
Test 1	Old Emulator	sched-simple	1.0	100%	Identical
Test 1	New Emulator	sched-simple	1.0	100%	Identical
Test 1	New Emulator	Fluxion FCFS	1.0	100%	Identical
Test 2	Old Emulator	sched-simple	0.4	83.33%	Identical
Test 2	New Emulator	sched-simple	0.4	83.33%	Identical
Test 2	New Emulator	Fluxion FCFS	0.4	83.33%	Identical

Figure 5.1 details the results of our experiment through a box-and-whisker plot. Fluxion (blue) has a median throughput of 274.2 jobs/sec with 1,000 jobs and 272.1 jobs/sec for 10,000 jobs. Fluxion with the emulator (green) has a median throughput of 271 jobs/sec with 1,000 jobs and 273.8 jobs/sec for 10,000 jobs. For the 1,000 job run, Fluxion with the emulator is 1.17% slower on average than Fluxion alone, and in the 10,000 job run, Fluxion with the emulator is 0.62% faster on average. The minor variation in the results between Fluxion and Fluxion with the emulator indicates that the emulator does not introduce noticeable overhead in Fluxion.

5.4 Assessing Scheduling Policy in Fluxion

In this experiment, we use the Flux Emulator and Tuolumne to compare Fluxion with a first-come-first-serve (FCFS) policy against Fluxion with a conservative backfilling policy. We compare these two scheduling policies along three metrics: makespan, system utilization, and job execution order. By comparing these metrics, we can explain the improvements of one policy over the other. This process involved three steps. First, we generated a set of jobs that would produce quantifiable results when introducing backfilling. Then, we used these jobs with the Flux Emulator to predict how they would be executed using the two scheduling policies. Finally, we run an identical set of jobs on Tuolumne using the two scheduling policies and compare the results of the real system to the results of the emulator. The following sub-sections expand upon these steps.

5.4.1 Input Data Generation

To create our input data, we generate synthetic job traces based on a double Gaussian distribution (Figure 5.2) where the first peak reaches 100 nodes and the second peak reaches 60 nodes. We then apply an even distribution of 100 points along our double Gaussian distribution where the Y value represents the resource requirements (number

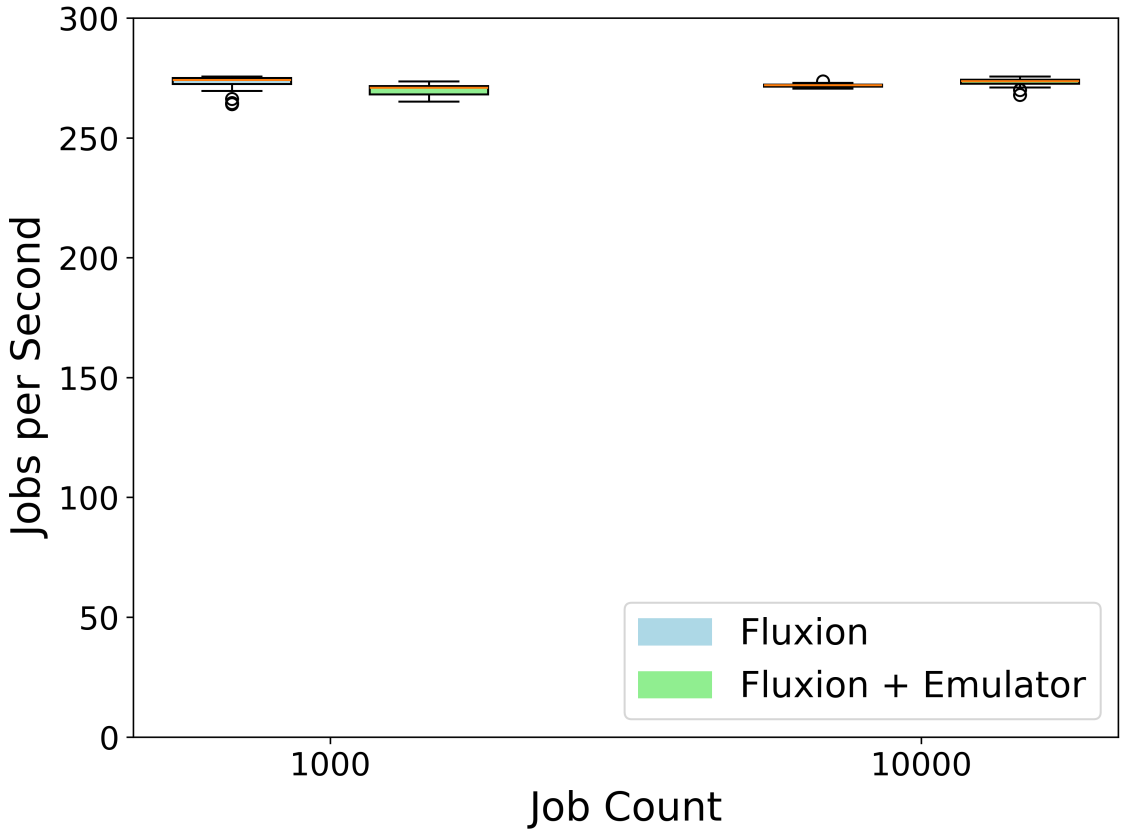


Figure 5.1: Results of Flux throughput with Fluxion alone and Fluxion with the emulator

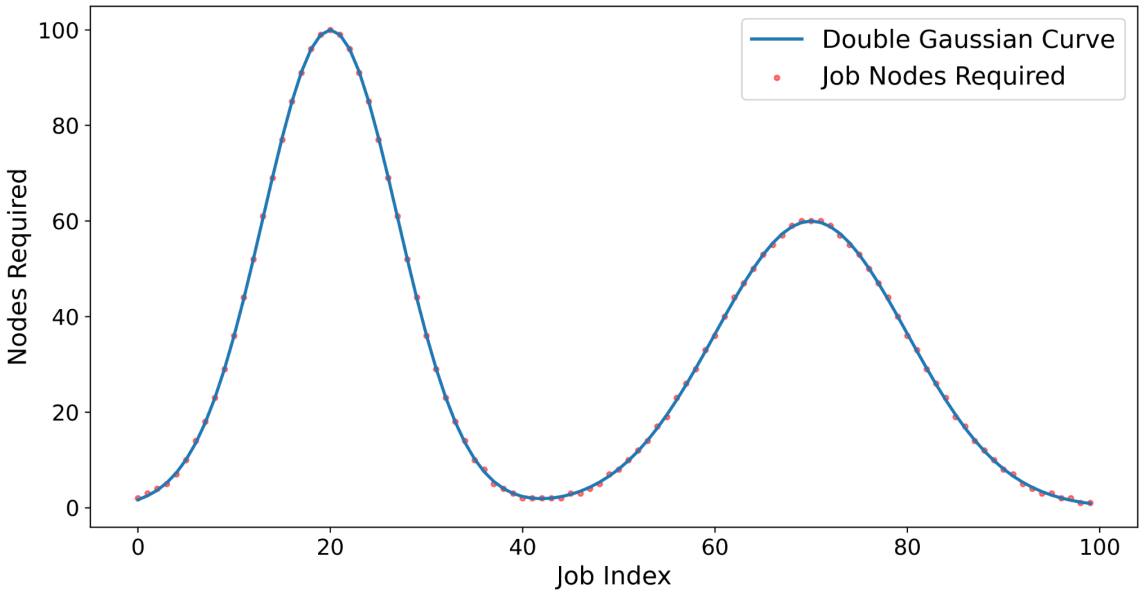


Figure 5.2: Double Gaussian distribution used to generate synthetic job traces.

of nodes) for each job. Each job is given 6-minute runtimes and 10-minute timeouts. By varying resource demands for jobs while keeping runtimes consistent, we create a dataset that highlights the impact of backfilling on the total makespan.

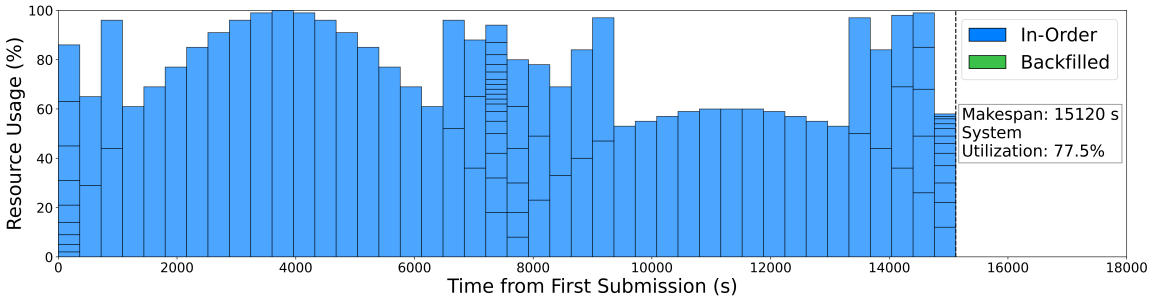
5.4.2 Results with the Flux Emulator

After generating the dataset, we input it to the emulator, which is configured to emulate a 100 node cluster. We perform one run using FCFS and one run with conservative backfilling. The results in Figure 5.3 show that conservative backfilling in Fluxion reduces the makespan by 16.7% and increases system utilization from 77.5% to 92.9% over FCFS. We obtain these results in seconds because the test uses the emulator, whereas the actual combined makespans on a real system theoretically exceed 7.7 hours.

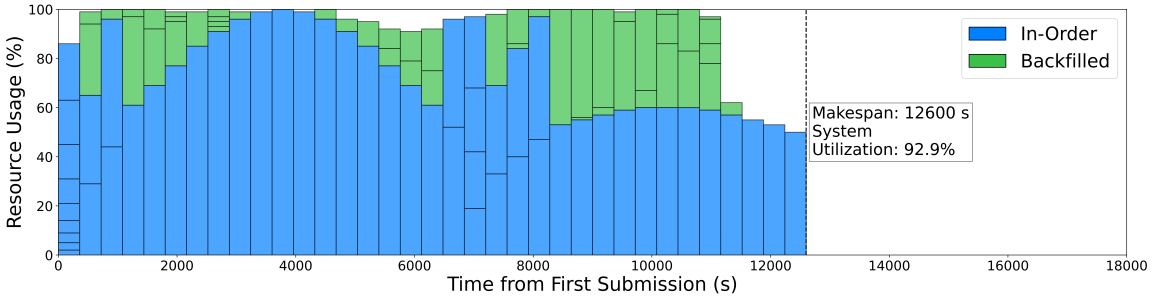
5.4.3 Comparison With a Real System

Next, we run an identical set of jobs on Tuolumne to determine whether the emulator provides an accurate representation of Flux on a real system. To replicate the conditions of our emulated scheduling policy assessment using Tuolumne, we submit a batch job for 100 nodes and, within that batch job, we submit 100 jobs with the same resource requirements and time limits that were used in our emulator test. We use a batch job because it allows our test to run in its own isolated Flux instance without interference from other users. To replicate the duration of individual jobs, we use 6-minute sleep commands. We then collect information about the completed jobs' runtimes.

The results in Figure 5.4 are very similar to those we generate using the Flux Emulator. We observe subtle differences in the sequence of backfilled jobs in the conservative backfilling test and a slightly longer overall makespan for both scheduling policies. These variations are likely due to the emulator abstracting minor fluctuations in the start and finish times for jobs and not accounting for prolog and epilog events. However, minor variations such as these are expected in an emulated system, and the emulator accurately predicts both the decrease in makespan and the increase in system utilization when comparing FCFS (5.4a) to conservative backfilling (5.4b) on Tuolumne. Overall, these findings indicate that the Flux Emulator is a reliable tool for predicting the impact of scheduling policy changes on a real system.

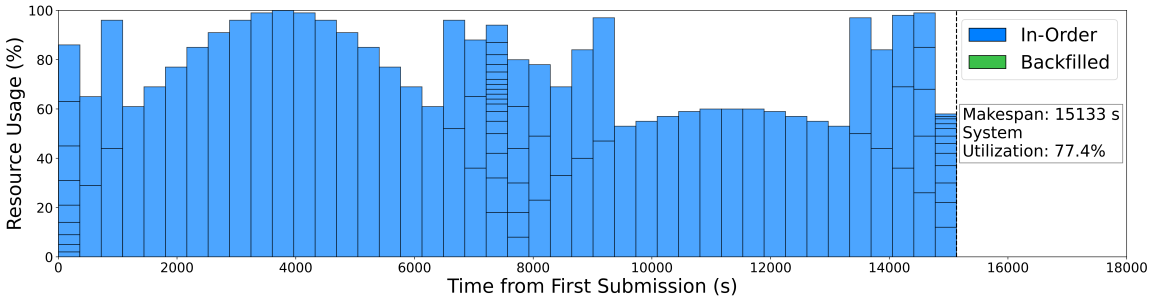


(a) First Come First Serve (FCFS)

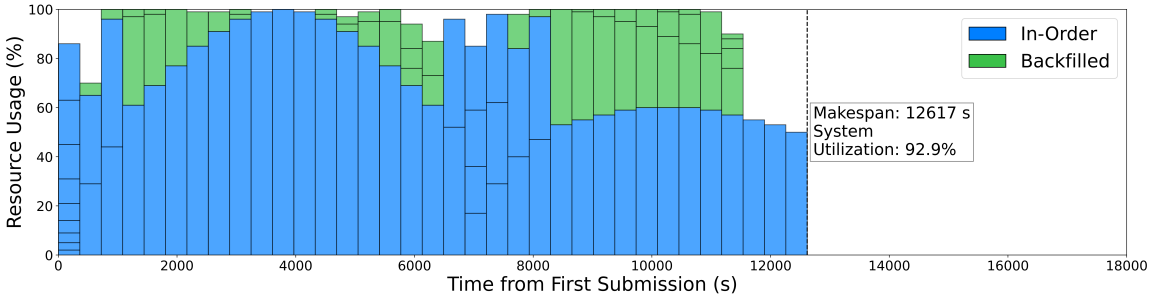


(b) Conservative Backfilling

Figure 5.3: Results of job submissions using FCFS (top) and Conservative Backfilling (bottom) scheduling policies with the Flux Emulator.



(a) Tuolumne: First Come First Serve (FCFS)



(b) Tuolumne: Conservative Backfilling

Figure 5.4: Results of job submissions using FCFS (top) and Conservative Backfilling (bottom) scheduling policies on Tuolumne.

5.5 Lessons Learned and Next Steps

Through a series of experiments, we validate the correctness, efficiency, and practicality of the Flux Emulator. Our correctness tests confirm that the new emulator produces results identical to the old emulator when using both sched-simple and Fluxion with a first-come-first-serve (FCFS) scheduling policy. This ensures that the emulator maintains accuracy while integrating new scheduling mechanisms. We assess the emulator's overhead by measuring job throughput in Fluxion with and without the emulator. Our results indicate that the emulator introduces negligible overhead, with variations of less than 1.17% in job throughput. This suggests that the emulator can be used to model scheduling scenarios without significantly affecting the performance of the scheduler. Finally, we use the emulator to analyze the impact of scheduling policies, comparing FCFS against conservative backfilling. The emulator predicts a 16.7% reduction in makespan and an increase in system utilization from 77.5% to 92.9% when using conservative backfilling. When we replicate these experiments on Tuolumne, we observe similar results, with minor variations attributed to real-world execution factors such as prolog and epilog events. These findings demonstrate that the emulator accurately models scheduling behavior, making it a valuable tool for evaluating policy changes before deploying them in production. Overall, our results demonstrate that the new Flux Emulator replicates the behavior of the old Flux Emulator while allowing users to study scheduling policy in Fluxion without introducing overhead.

In future work, we plan to explore the emulator's scalability limits by increasing the number of simultaneous job inputs. Additionally, we will remove the abstraction of timing for the start and end of jobs in the emulator. This will allow us to achieve greater consistency in results when compared to real systems, and it will allow us to study the performance of Fluxion, which is important for testing advanced scheduling techniques that include intensive operations. Finally, we plan to use the emulator to facilitate active research into scheduling within Flux, such as multi-cluster scheduling.

Chapter 6

Summary and Future Work

6.1 Summary

When addressing the challenge of monitoring resource differences in HPC systems when using SLURM, our approach significantly affects system administrators and end users. Our solution addresses a pivotal shortfall in tools such as XdMoD by facilitating prompt GPU usage monitoring and analysis on a per-job basis, specifically on ACCESS systems. Among our key contributions, we propose an adaptable data ingestion technique that circumvents the need to modify SLURM configurations and makes use of the existing Performance Co-Pilot (PCP) framework. This system not only identifies, but also resolves inefficiencies in GPU utilization while offering a user-oriented interface equipped with notification capabilities and a web-based dashboard to enhance resource management practices. Our findings are encouraging, indicating the potential for a reduction in GPU job queue times, enhanced system utilization, and more efficient optimization of current resources to accommodate future expansion.

On the other hand, the findings obtained from our research using the updated version of the Flux Emulator indicate that it serves as an accurate tool for emulating the Flux Framework without imposing any additional computational burden. Our investigation, which incorporates a backfilling study, demonstrates the potential

of the emulator by achieving a significant reduction in the makespan by 16.7% and an increase in overall system utilization from 77.5% to 92.9% when employing conservative backfilling techniques. When comparing these results to those of an identical experiment performed on a real system, we observe very similar behavior to the emulator with minor variations attributed to real-world execution factors such as prolog and epilog events. Additionally, the emulator produces these results in a matter of seconds, whereas the real system takes more than seven hours. Such results emphasize the Flux Emulator's utility in expediting development and assessment processes related to novel scheduling policies within high-performance computing (HPC) systems that incorporate the Flux framework.

6.2 Future Work

Future research directions include a pilot study to evaluate the effectiveness of our data ingestion method and user-facing platform in increasing GPU utilization. These next steps involve the integration of our proposed solution into the Open XdMoD framework, and we aim to expand its applicability to encompass other systems within the ACCESS infrastructure.

In future research using the Flux Emulator, our intention is to investigate the scalability limits of the emulator by progressively increasing the volume of concurrent job inputs that it can process. Also, our aim is to make improvements to the emulator's code that facilitate maintenance and remove abstractions affecting accuracy.

Bibliography

[1] Dong H. Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, and Martin Schulz. Flux: A Next-Generation Resource Management Framework for Large HPC Centers. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 9–17, 2014. doi: 10.1109/ICPPW.2014.15. 2

[2] S. T. Brown et al. Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research. In *Practice and Experience in Advanced Research Computing*, pages 1–4, 2021. doi: 10.1145/3437359.3465593. 12

[3] Paola A. Buitrago and Nicholas A. Nystrom. Neocortex and Bridges-2: A High Performance AI+HPC Ecosystem for Science, Discovery, and Societal Good. In *High Performance Computing*, pages 205–219. Springer International Publishing, 2021. doi: 10.1007/978-3-030-68035-0_15. 12

[4] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. Batsim: A Realistic Language-Independent Resources and Jobs Management Systems Simulator. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 178–197. Springer, 2015. 3

[5] Jim Garlick. 21/Job States and Events Version 1, . URL https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_21.html. Flux 0.13.0 documentation. 7

[6] Jim Garlick. 20/Resource Set Specification Version 1, . URL https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_20.html. Flux 0.13.0 documentation. 7

[7] Morris Jette. Tuning Slurm Scheduling for Optimal Responsiveness and Utilization, 2014. URL https://slurm.schedmd.com/SUG14/sched_tutorial.pdf. SLUG14. 5

[8] Morris Jette and Danny Auble. SLURM Version 1.3, May 2008. URL https://slurm.schedmd.com/pdfs/slurm_v1.3.pdf. 5

[9] Jie Li, George Michelogiannakis, Brandon Cook, Dulanya Cooray, and Yong Chen. Analyzing Resource Utilization in an HPC System: A Case Study of NERSC Perlmutter, 2023. URL https://arxiv.org/abs/2301.05145. 13

[10] LLNL. Tuolumne, 2024. URL https://hpc.llnl.gov/hardware/compute-platforms/tuolumne. Tuolumne — HPC @ LLNL. 26

[11] Alejandro Lucero. Slurm Simulator, 2011. URL https://slurm.schedmd.com/slurm_ug_2011/slurm_simulator_phoenix.pdf. SLUG11. 3

[12] Maxime Martinasso, Miguel Gila, Mauro Bianco, Sadaf R. Alam, Colin McMurtrie, and Thomas C. Schulthess. RM-Replay: A High-Fidelity Tuning, Optimization and Exploration Tool for Resource Management. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 320–332, 2018. doi: 10.1109/SC.2018.00028. 3

[13] Jeffrey T. Palmer et al. Open XDMoD: A Tool for the Comprehensive Management of High-Performance Computing Resources. *Computing in Science & Engineering*, 17(4):52–62, 2015. doi: 10.1109/MCSE.2015.68. 2

[14] Tapasya Patki, Dong Ahn, Daniel Milroy, Jae-Seung Yeom, Jim Garlick, Mark Grondona, Stephen Herbein, and Thomas 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, SC-W '23, pages 2077–2088, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707858. doi: 10.1145/3624062.3624286. URL https://doi.org/10.1145/3624062.3624286. 4, 8

[15] Josko Plazonic, Jonathan Halverson, and Troy Comi. Jobstats: A Slurm-Compatible Job Monitoring Platform for CPU and GPU Clusters. In *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, PEARC '23, pages 102–108, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399852. doi: 10.1145/3569951.3604396. URL https://doi.org/10.1145/3569951.3604396. 13

[16] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Robert L. DeLeon, Joseph P. White, Steven M. Gallo, Abani K. Patra, and Thomas R. Furlani. A Slurm Simulator: Implementation and Parametric Analysis. In Stephen Jarvis, Steven Wright, and Simon Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 197–217, Cham, 2018. Springer International Publishing. ISBN 978-3-319-72971-8. 3

[17] TOP500. El Capitan achieves top spot, Frontier and Aurora follow behind, 2024. URL https://top500.org/news/el-capitan-achieves-top-spot-frontier-and-aurora-follow-behind/. TOP500. 2

[18] Tiffany Trader. Livermore's El Capitan Supercomputer to Debut HPE "Rabbit" Near Node Local Storage, February 2021. URL https://www.hpcwire.com/2021/02/18/livermores-el-capitan-supercomputer-hpe-rabbit-storage-nodes/. HPCwire. 2, 8

[19] Michael R WyattII, Stephen Herbein, Todd Gamblin, and Michela Taufer. AI4IO: A Suite of AI-Based Tools for IO-Aware Scheduling. *The International Journal of High Performance Computing Applications*, 36(3):370–387, 2022. doi: 10.1177/10943420221079765. URL https://doi.org/10.1177/10943420221079765. 21

[20] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39727-4. 1

Vita

Walter Jay Ashworth, an alumnus of The University of Tennessee, completed his Bachelor's degree in Computer Science in May 2023, and he chose to delve deeper into academia at his alma mater by pursuing a Master's degree. He is currently contributing his talents as a Graduate Research Assistant at the esteemed Global Computing Lab, benefiting from the mentorship of the renowned Dr. Michela Taufer. Here, he primarily develops software tools for performance data collection and analytics such as the Flux Framework Emulator in collaboration with Lawrence Livermore National Laboratory. Jay's research interests include HPC schedulers, performance analysis, and visualization.