ABSTRACT

BEHERA, SUBHENDU SEKHAR. Adaptive Workflow Scheduling and Fault Tolerance to Manage HPC Resources. (Under the direction of Dr. Frank Mueller.)

Large-scale AI and high-performance computing systems are breaking barriers with exascale capabilities that accelerate research and development, driving scientific and technological innovation. However, as scale and heterogeneity increase, failures — and the inefficiencies in recovering from them — cause substantial loss of computation, degrading both workflow performance and overall system efficiency. Moreover, even with massive computing capacity, users often struggle to meet deadlines for large and complex workflows. Consequently, intelligent and adaptive resource management becomes essential for workflow scheduling and fault tolerance to mitigate work loss, enable efficient recovery, and ensure timely execution.

This work identifies and leverages opportunities for prioritization, predictability, and scalability within the HPC ecosystem to alleviate these challenges. We integrate these principles into three adaptive and complementary solutions across the HPC software stack. First, our predictive, failure-aware Checkpoint/Restart model mitigates the limitations of short failure lead times by prioritizing processes within an application, thereby improving efficiency. Second, we introduce a workflow scheduling framework driven by resource availability prediction that overcomes the shortcomings of traditional cloud bursting approaches and enables deterministic workflow execution. Finally, we leverage the existing scalable scheduling techniques for workflow management that support efficient and coordinated recovery from failures while reducing resource inefficiencies associated with uniform, isomorphic failure handling. Together, these contributions offer cohesive and synergistic resource management strategies that advance adaptive workflow scheduling and fault tolerance on modern HPC systems.

Adaptive Workflow Scheduling and Fault Tolerance to Manage HPC Resources

by
Subhendu Sekhar Behera

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2026

APPROVED BY:

Dr. Huiyang Zhou	Dr. Jiajia Li
Dr. Xipeng Shen	Dr. Frank Mueller Chair of Advisory Committee

DEDICATION

This dissertation is dedicated to my family, for their unwavering love and support, and to my native place, whose roots have guided and grounded me.

BIOGRAPHY

The author grew up in the village of Bhatakumarada, Odisha. He completed his early schooling there before moving to Brahmapur, Odisha, for higher studies. He earned a Bachelor of Engineering in Computer Science from the National Institute of Science & Technology, Odisha. Following graduation, he worked in Bengaluru at Broadcom and Juniper Networks before moving to the United States. He pursued a master's degree in Computer Science at North Carolina State University (NCSU) and continued on to a Ph.D. under the guidance of Dr. Frank Mueller. During his time at NCSU, he completed summer internships at ORNL, LLNL, and NVIDIA.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to the following advisors and collaborators.

First, I thank Lipeng Wan and Matthew Wolf for providing me the opportunity to undertake a summer internship at Oak Ridge National Laboratory in 2019. That experience served as a gateway to research and inspired my decision to pursue a PhD. I also extend my gratitude to Jae-Seung Yeom, Daniel Milroy, Dong H. Ahn, and Stephen Herbein at Lawrence Livermore National Laboratory for their continuous support, guidance, and encouragement throughout this journey.

I am grateful to my committee members, Dr. Huiyang Zhou, Dr. Xu Liu, Dr. Jiajia Li, and Dr. Xipeng Shen, for their time, insightful feedback, and valuable suggestions on my research. Finally, I would like to offer my deepest thanks to my MS and PhD advisor, Dr. Frank Mueller, for his unwavering support, mentorship, and guidance over the years.

I also want to thank my family and friends for their love, patience, and encouragement, which have been a constant source of strength throughout this journey.

AUTHORSHIP STATEMENT

Contributions of Subhendu Sekhar Behera and co-authors are listed below for each chapter.

Chapter 1: Introduction

Contributions:

- Subhendu Sekhar Behera: sole author of Chapter 1.

Chapter 2: P-ckpt: Coordinated Prioritized Checkpointing

Citation: S. Behera, L. Wan, F. Mueller, M. Wolf and S. Klasky, "P-ckpt: Coordinated Prioritized Checkpointing," 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 2022, pp. 436-446, doi: 10.1109/IPDPS53621.2022.00049.

Contributions:

- Subhendu Sekhar Behera: preliminary investigation, design, implementation, literature review, development of methodologies, coding, evaluation, and writing (drafting).
- Frank Mueller: design, supervision, feedback, and writing(reviews/edits).
- Lipeng Wan, Matthew Wolf, Scott Klasky: design, feedback, and facilitation of resources for evaluation and analysis.

Chapter 3: Predictive Execution of Workflows in a HPC+Cloud Environment

Citation: Accepted and presented at HiPC 2025. In the process of being published.

Contributions:

- Subhendu Sekhar Behera: preliminary investigation, design, implementation, literature review, development of methodologies, coding, evaluation, and writing (drafting).
- Frank Mueller: design, supervision, feedback, and writing (reviews/edits).
- Jae-Seung Yeom, Daniel Milroy: design, feedback, and writing (reviews/edits).
- Marc Niethammer: provision of workflow, feedback.

Chapter 4: WHESL: A Distributed Exception Management Framework for HPC

Citation: Submitted.

Contributions:

- Subhendu Sekhar Behera: preliminary investigation, design, implementation, literature review, development of methodologies, coding, evaluation, and writing (drafting).
- Frank Mueller: design, supervision, feedback, and writing (reviews/edits).

- Jae-Seung Yeom, Daniel Milroy, Dong H. Ahn, Stephen Herbein: design and feedback.

Chapter 5: Conclusion
Contributions:

- Subhendu Sekhar Behera: sole author of Chapter 5.

Use of generative artificial intelligence: In chapters 1 and 5, a Large Language Model-based tool (ChatGPT) was used to improve clarity and grammatical correctness and bring smooth transitions between sentences.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1 INTRODUCTION

Adaptive and efficient resource management techniques are critical to achieving resilience and deterministic scheduling on modern large-scale High-performance Computing (HPC) systems. In the absence of robust resiliency and timely scheduling mechanisms, HPC systems risk underutilization and inefficient use of compute resources due to failures, as well as an inability to meet deadlines for time-critical workloads. To highlight these concerns, we identify three key challenges faced by today's HPC ecosystem.

Challenge 1: Checkpoint/Restart (C/R) is widely used to enable reliable execution of applications ranging from exascale simulations to large language model training. When coupled with failure prediction, C/R can further improve overall execution efficiency. However, existing failure-aware C/R techniques fall short when handling failures with low lead time. Under high I/O contention, vulnerable nodes cannot deterministically save state to persistent storage, such as a parallel file system. In such scenarios, large-scale applications suffer from computation loss, resulting in resource wastage.

Challenge 2: Beyond resiliency, efficient resource utilization and predictable performance across heterogeneous HPC–Cloud systems remain open challenges. Meeting deadlines for data-intensive workflows is difficult because HPC jobs experience variable queue wait times. Cloud Bursting can provision additional public-cloud resources to meet demand, but it lacks deterministic scheduling because it depends on local HPC resources that may not be available. Additionally, data movement latency between HPC and cloud environments becomes a performance bottleneck, leading to resource idleness and increased cost.

Challenge 3: As workflows grow more complex and distributed, fault tolerance must extend beyond individual applications to entire workflow ecosystems. Existing fault-tolerance approaches, such as C/R and resource over-provisioning, perform well for monolithic parallel jobs but fail to support complex heterogeneous workflows due to limited awareness of the workflow–HPC hierarchical ecosystem. Workflow management systems orchestrate such workflows but typically provide only basic fault tolerance, such as job restarts or simple error handling via programming-language mechanisms. These approaches lack the holistic, system-wide coordination required for efficient recovery.

To address this diverse set of challenges, we propose the following hypothesis.

Hypothesis: *To improve application and system efficiency in the presence of failures and inefficient recovery, and to ensure the timely execution of workflows, modern HPC systems require intelligent and adaptive resource management techniques for fault tolerance and scheduling that leverage prioritization, predictability, and scalability within the HPC ecosystem.*

To evaluate this hypothesis, we developed three primary solutions that leverage techniques such as prioritization, predictability, and scalability within the HPC ecosystem. This thesis is organized into three main chapters. In Chapter 2, we develop *p-ckpt*, a novel checkpoint/restart (C/R) technique driven by a failure prediction model that coordinates distributed processes to prioritize checkpointing on vulnerable nodes, thereby enabling contention-free access to the parallel file system (PFS). We further integrate complementary technologies, such as live migration and burst buffers, into *p-ckpt* to reduce work loss and improve checkpointing efficiency. Our hybrid *p-ckpt* C/R model considers prediction lead time and checkpoint latency to the PFS when deciding on feasible proactive actions, such as proactive checkpointing or live migration. We evaluate our solution through simulation using six real-world applications on the Summit supercomputer, comparing it against state-of-the-art approaches in terms of overhead reduction. We assess our C/R models under multiple failure distributions and analyze the impact of lead-time variability and failure prediction accuracy. Based on this evaluation, we discuss the trade-offs associated with these models and their effects on overall application overhead.

In Chapter 3, we present a predictive workflow scheduling approach that enables deterministic execution and controlled cost–deadline trade-offs across hybrid environments, providing reliable guarantees on both cost and deadlines for large-scale workflows. We propose scheduling data-intensive workflows deterministically over a combined HPC–cloud hybrid environment by scavenging unused HPC resources. We predict the availability of HPC resources and exploit these predictions to dynamically split resource allocation between unused HPC capacity and cloud on-demand resources to complete workflows by a given deadline. Deterministic resource allocation enables preloading of input data for workflow tasks, thereby avoiding execution delays. Furthermore, we develop an adaptive scaling algorithm that effectively backs up the targeted HPC allocation with cloud resources to prevent workflow execution delays in the event of inaccurate resource availability estimation. We evaluate our framework against state-of-the-art solutions based on task completion rate, impact on HPC production jobs, cost savings, and cost estimation error.

In Chapter 4, we present the Workflow Hierarchy-aware Distributed Exception Management System (WHESL), a novel solution that standardizes FT specification and manages exceptions among disparate components in workflows in a coordinated and programmable manner. WHESL is primarily built on three core components: ESL, an Exception Specification Language that can succinctly define a rich set of exceptions and relate them to the hierarchy of workflows; EMIL, the Exception Management and Isolation Layer, which provides exception isolation and coordination among the components of the workflow execution environment; and ERM, the Exception Resource Management library that provides advanced recovery mechanisms. Our solution is built on top of Flux, a next-generation hierarchical resource and job management system, from which we utilize scalable and nested scheduling capabilities. By evaluating different types of exceptions, we show that WHESL can significantly extend the traditional HPC FT support for modern workflows, thereby enabling scalable, usable, and portable fault tolerance on modern HPC systems.

CHAPTER

2 P-CKPT: COORDINATED PRIORITIZED CHECKPOINTING

2.1 Introduction

Failures and I/O contention add significant overhead to application execution and become the key challenge for C/R efficiency [Gei03; Sat12; Sch07; Luc14; Liu12; Isk08]. In past years, significant progress has been made on failure prediction [Das18b; Das18c; Gai12b; Gai12a], live migration (LM) [Wan08; Wan12], and Burst Buffers (BBs) utilization [Liu12; Fan] to address the challenges of fault tolerance and PFS I/O contention. Based on these techniques, many hybrid solutions such as failure-aware safeguard checkpointing [Bou13; Tiw14], and multi-level C/R by orchestrating failure prediction, LM, BBs and periodic checkpointing [Beh20] have been proposed. Multi-level Checkpoint models employ multiple storage layers with different latency to minimize checkpoint latency while improving system efficiency [Moo10; Di17; Di14]. Safeguard Checkpoint [Bou13] tries to minimize computation loss by checkpointing just-in-time before an anticipated failure. However, effectiveness of these failure-aware C/R solutions depends on the accuracy of the failure prediction model and the length of the lead times to predicted failures. Proactive techniques, such as LM, require high lead times for a larger memory footprint while safeguard checkpoints require enough lead time to complete a proactive checkpoint until the data is committed to the PFS amid I/O congestion. Without adequate lead time, an effort to complete the safeguard checkpoint is not guaranteed to succeed (i.e., complete before the failure) on vulnerable nodes given the unpredictable nature of I/O completion on large HPC systems. Such challenges require solutions that can effectively meet the deadline to commit checkpoint data to the PFS on vulnerable nodes. Prioritizing checkpoint data bleed-off on individual nodes is a promising direction. A filesystem-level implementation of prioritizing checkpoint data on vulnerable nodes to PFS requires complex changes in the filesystem, I/O server layer, and interconnect network layer. Further, in a system with a high failure rate, simple prioritization of an unhealthy node would fail given the Weibull distribution of failure arrival times on HPC systems [Tiw14].

To address these challenges, we propose a novel coordinated prioritized checkpoint method, called *p-ckpt*, that coordinates processes within an application in their effort to store checkpoint data to the PFS in giving vulnerable nodes higher priority for such actions. The core idea is as follows: In the event of failure predictions, a *p-ckpt* request gets initiated by the vulnerable node. It triggers the checkpointing process to, in the first phase, checkpoint to the PFS on only the vulnerable nodes. During this time, the healthy nodes await a checkpoint completion notification from vulnerable nodes to move forward to the next phase, and a new node predicted to fail during this phase gets queued in the node-local priority queue based on its lead time to failure. Once all the vulnerable nodes commit their checkpoint data, the remaining nodes commit their checkpoint data to the PFS in a second phase. Such coordination is facilitated by prioritizing vulnerable nodes based on the lead time to the predicted failure. A lower lead time implies a higher priority. Further, we incorporate LM into the C/R model, thereby creating a so-called *hybrid p-ckpt*. LM is the preferred proactive choice over prioritized checkpoints as it is cost-effective in terms of network traffic and its ability to let the application continue execution while LM is in progress [Wan08].

Our multi-level *hybrid p-ckpt* C/R model for modern HPC systems can benefit in performance from failure prediction and an analysis model for effective use and coordination of multiple resilience techniques such as *p-ckpt*, LM, and periodic checkpoints. Our adaptive C/R model is driven by failure lead time prediction to select an appropriate proactive action with the smallest possible overhead in the presence of failures. Desh's [Das18b] log-based failure chain characterization technique is utilized to detect instances of likely failures in real-world HPC system logs (three HPC systems) and their lead time distribution, which provides the means for failure prediction and system failure rate calculation. LM and *p-ckpt* checkpoints are integrated into our C/R model to provide just-in-time mitigation of a predicted failure. The approach is unique in that it selects the best possible actions dynamically based on the lead time to failure, either via *p-ckpt*, to checkpoint to the PFS, or via LM with minimal interruption to application execution. The impact of this trade-off is subject to our evaluations.

In summary, we make the following contributions:

• We assess the impact of short lead times to predicted failures comparing existing safeguard checkpoint and live migration [Bou13; Beh20] solutions.

• We propose a novel checkpoint technique, (*p-ckpt*), that allows coordination at application-level to prioritize the saving of state on nodes with imminent health problems to avoid computational loss due to failures.

• We develop a *hybrid p-ckpt* C/R model that coordinates fault tolerance techniques of LM and coordinated prioritized checkpointing while considering the latency of multiple I/O layers for storing checkpoints, driven by a log-based failure analysis and prediction model to reduce failure overhead.

• We evaluate the *p-ckpt* and *hybrid p-ckpt* C/R models against multiple failure distributions and measure their effectiveness while assessing sensitivity to lead times to failures. We test their robustness against failure prediction accuracy. We discuss these evaluations and make suggestions on their applicability. Further, we evaluate the impact of checkpoint size and LM transfer size on the performance of LM and *p-ckpt* models and provide an analytical model.

2.2 System Model

Our work is modeled on an HPC system that resembles the Summit supercomputer architecture. Each compute node has a BB serving as an intermediate storage device to absorb I/O write bursts locally. Other HPC architectures historically employed a cluster of dedicated BB nodes, e.g., NERSC's Cori [Bhi16]. On Summit, each BB device has 1.6 TB capacity, compared to a 512 GB DRAM size, with up to 2.1 GB/sec write and 5.5 GB/sec read I/O bandwidth [Vaz18]. BBs assist in reducing PFS I/O contention in two situations in our C/R model: First, periodic checkpoints are cached on the BBs and later asynchronously bled off to the PFS. The asynchronous bleed off is optimized by limiting the number of nodes that transfer data to the PFS at any time. Second, during recovery from unmitigated failures, only one node, the replacement node, needs to recover checkpoint data from the PFS. The rest of the nodes recover data from their local BB.

Further, each node has an instance of a fault predictor using the Aarohi [Das20b] model running on a separate core than the application. Aarohi suggests placing predictors on Hardware Supervisory System (HSS) that manages a chassis on Cray systems. Their observations are based on the grounds of application interference from the predictor. However, notifying the prediction handler subsystem/thread on a compute node from a chassis controller can be challenging in terms of latency, particularly when prediction lead times are in the range of a few seconds. Placing the predictor on the node itself eliminates this problem, preferably on a spare core or otherwise by core sharing with applications. Aarohi itself is a lightweight monitor that predicts a failure by analyzing 18 different logs within 0.31 msecs on average. Checkpoint data is transferred from BBs to the PFS asynchronously, e.g., via the Spectral library on Summit [ORN20]. We assume that the integrity of the checkpoint data stored on local BBs is maintained, and the checkpoint size per node never exceeds the DRAM or BB size.

Checkpoint Model: Our proactive checkpoint technique (applicable to both *p-ckpt* and safeguard checkpoint) mandates that all the nodes commit their checkpoints to the PFS in the event of failure prediction, thereby bypassing the BBs. In contrast, periodic checkpoints are staged on to the BBs first and later drained to the PFS asynchronously. Other strategies have been pursued in multi-level checkpointing models, such as neighbor checkpointing and local disk/BB checkpointing [Moo10; BG11]. Evaluating these methods are beyond the scope of this work, but as they are orthogonal, can themselves benefit from prioritization. Further, we mandate all the nodes in an application to save their state to avoid application restart and synchronization issues. Thus, if recovery happens from a non-handled failure, then all the healthy nodes recover checkpoint data, which was stored in a periodic checkpoint on their local BBs, while the replacement node recovers from the PFS. If a failure is mitigated with proactive checkpointing, then all the nodes recover from the PFS.

Failure Model: We make the following assumptions in our failure model:

- Failures can happen at any point in time.
- The impact of failure is limited to a single node.
- Another failure, predicted or not, can occur on a node that already had a previous failure predicted, but still with some lead time left before the failure is predicted to occur.

The Optimal Checkpoint Interval (OCI) is the near-optimal time gap between two consecutive checkpoints that aims to lower the checkpoint frequency while minimizing the computation loss due to failures. Young's formula [You74] for OCI applies to single-level C/R models. Previous work by Di et al. [Di14; Di17] and Benoit et al. [Ben17] focused on the optimal checkpoint interval for multiple types of checkpoints, each of them stored on a separate storage medium. However, our HPC system model employs intermediate storage devices (BBs in this case) that stage the checkpoints before being bled off to the PFS asynchronously. Failures during the asynchronous checkpointing can cause loss of computation performed during the current and previous iterations as shown in Figure 2.1(B). However, during our evaluation, we found that this asynchronous checkpoint window is negligible compared to the OCI because of the high performance of the PFS on Summit (see Section 2.4). So we use Young's formula in Eq. 2.1 for OCI calculation, where t_{cmpt}^{opt} is the OCI, λ the failure rate, c the number of compute nodes a job is running on, and t_{ckpt}^{bb} the time required to

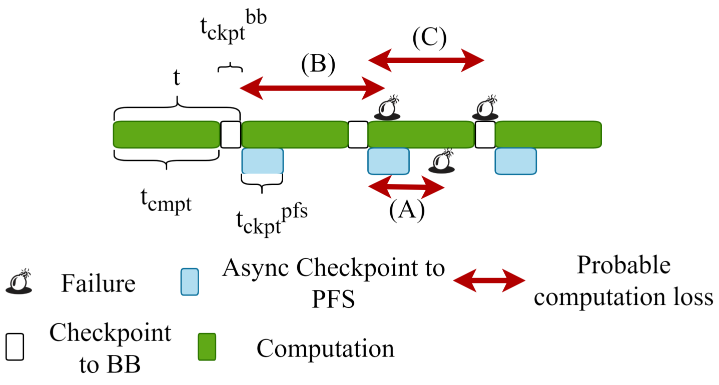


Figure 2.1 Computation loss upon failure during (A) computation post checkpointing to PFS, (B) asynchronous checkpointing to PFS, and (C) synchronous checkpointing to BB

write one checkpoint to the BBs by a job.

$$t_{cmpt}^{opt} = \sqrt{\frac{2t_{ckpt}^{bb}}{\lambda c}} \tag{2.1}$$

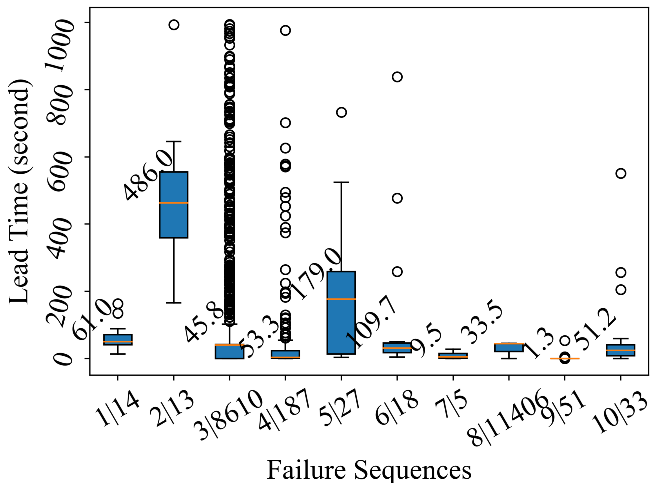


Figure 2.2 Failure prediction lead time distribution

The OCI in our checkpoint model further includes a rigorous analysis of failures logs. The study analyzes the system logs collected from three real-world HPC systems from Desh [Das18b] over a period of six months. Using the Desh approach, the most common sequences of phrases in logs that may lead to failure are considered. Our assumption in this work is that any sequence of phrases, so-called failure chains, results in an actual failure. The time difference between the first phrase and the last phrase in a chain is calculated as the lead time. Figure 2.2 shows the distribution of lead times as box plots for different failure instances (sequence 1-10), each of which occurs repeatedly in these logs. Failure sequence ID and the number of occurrences in the logs are on the x-axis. The y-axis represents the lead time in seconds. Mean lead time is on the left side of each boxplot. We observe that most failures are bounded by the whiskers, only a few outliers exist, with an exception of failure sequences 3 and 4. In the following experiments, we consider the actual lead time of any failure during simulation.

We introduce a parameter σ that represents the percentage of failures that can be predicted with enough lead time in excess of the time required to migrate a process from a faulty node to a new and healthy node. By considering a σ percent decrease in the rate of failures, we further improve the OCI. That means σ percent of failures can be predicted with a lead time in excess of θ seconds and thus can be avoided with proactive live migration. We calculate the value θ by assuming that the total amount of data transferred during live migration is equal to three times the processes' checkpoint data and is bounded by RAM size (512 GB). We account for a 3x higher footprint for LM as it migrates an entire process rather than just a subset of application data. Consider a stencil with a temporal domain of t-1, t, t+1, i.e., any particle point has 3 values in time (needed by LM whereas *p-ckpt* only needs one as others can be recalculated). This approximates the overhead assuming that these data structures dominate the memory consumption of an application. Note that Eq. 2.1 is used for the *p-ckpt* model while Eq. 2.2 is applicable to the *hybrid p-ckpt* model. We do not incorporate the percentage of failures handled by *p-ckpt* in the OCI as they cause the application to recover after failure. In contrast, with live migration, failures are avoided, i.e., no recovery process is required.

$$t_{cmpt}^{opt} = \sqrt{\frac{2t_{ckpt}^{bb}}{\lambda c\,(1-\sigma)}} \tag{2.2}$$

2.3 Simulation Framework

For evaluating the C/R models developed in this chapter, we rely on simulation. SimPy [Tea20], a process-based discrete-event simulation framework in Python, is used for developing our simulation framework. With SimPy, we simulate the time spent during computation, checkpointing to BBs

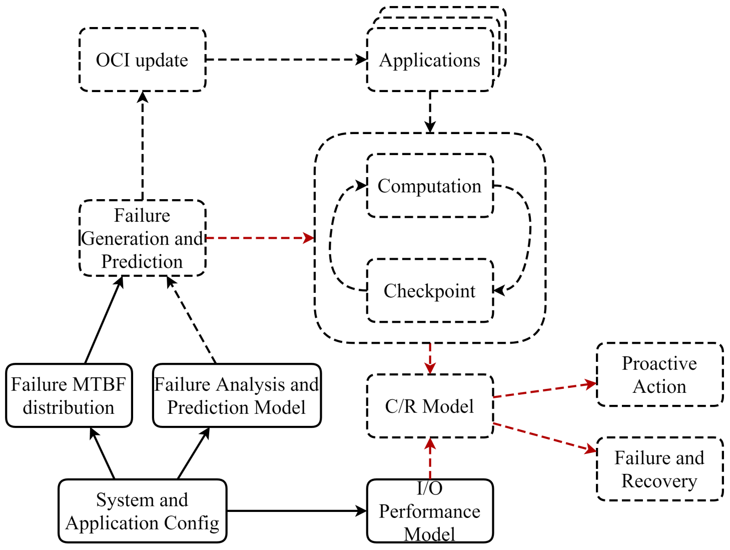

OCI update
Applications
Computation
Checkpoint
Failure Generation and Prediction
Proactive Action
C/R Model
Failure MTBF distribution
Failure Analysis and Prediction Model
Failure and Recovery
System and Application Config
I/O Performance Model

Figure 2.3 Simulation framework

and PFS, proactive operations, and inject failure events. Our simulation framework comprises multiple components (see Figure 2.3). Components boxed with dotted lines run as a SimPy process during simulation. The boxes with solid lines represent the input to these components. Arrows of dotted edges indicate either actions or input at runtime, arrows with solid lines are inputs during initialization time. Each simulated application runs as a SimPy process performing computation and periodic checkpointing iteratively. The OCI of each application SimPy process is updated periodically using Eq. 2.1 and Eq. 2.2 to better account for a dynamically changing system failure rate. The checkpoint period is constant as the applications store checkpoints to BBs while asynchronously draining them to PFS.

The system and application configuration file contains input describing application characteristics, PFS I/O performance statistics, failure distribution parameters (Table 2.3), and failure analysis (lead times) in detail. These data are fed into the simulation framework creating the static and dynamic components required for the simulation. The failure generation and prediction component uses the failure distribution parameters to generate one of the failures along with its prediction lead time using failure analysis described in Section 2.2 that is then injected into an application. For each failure generation, a node is randomly selected from a uniform probability distribution. The application on that node gets a failure prediction notification before the actual failure is triggered. Upon prediction, the application selects one of two proactive actions; which one depends on the C/R model algorithm being simulated.

When a failure is injected, the interrupted application uses the SimPy framework's time measurement APIs and the PFS I/O performance model to determine its state at the time of failure and calculates the amount of computation loss. Hence, the I/O performance model is an integral part of our simulation framework, which is described in the following section. Further, we make the following assumptions in our simulation:

• The rate of failures is lower than the rate of recovery for failed nodes so that reserved nodes are always available to the resource manager, such as Slurm [Yoo03] or Flux [al.18a].

• No distinction is made between soft failures and hard/node failures, i.e., both are handled uniformly, except during the recovery process. A failed node is always replaced by a new and healthy node.

• Checkpointing resembles application-level checkpointing.

2.4 I/O Performance Model

I/O performance is known to be variable due to I/O contention between different jobs. Even on modern HPC systems, the I/O bandwidth of an application is severely impacted by concurrent I/O operations performed by other applications. This causes significant variability in I/O performance. On Summit, IBM's SpectrumScale $GPFS^{TM}$ PFS handles application I/O using IBM's $GL4^{TM}$ Elastic Storage Servers as I/O nodes. The I/O subsystem evaluation in [Vaz18] shows an aggregate bandwidth of 2.5 TB/sec can be realized. However, the evaluation measures the performance of the I/O node server. It does not measure the I/O performance realized within an application. The objective here is to characterize the *actual* I/O performance seen by an application.

To characterize the I/O performance of the $GPFS^{TM}$, two experiments are conducted. The first experiment determines the optimal number of MPI processes that can achieve maximum aggregate I/O bandwidth on a single compute node. A compute node on Summit has 42 physical cores, which are evenly distributed over two sockets along with DRAM. This experiment measures the average I/O bandwidth for different aggregate data transfer sizes over multiple MPI tasks from 1 to 42 in 10 different runs. These MPI tasks are evenly distributed over the two sockets on the compute node and use POSIX write to transfer data. I/O buffers are flushed via the fsync() call to ensure that the data is not cached but rather committed to the devices. Figure 2.4 depicts the aggregate I/O bandwidth (y-axis) for different transfer sizes (x-axis) on curves ranging from 1 to 42 processes. These results indicate that 8 MPI tasks on a single compute node result in the maximum I/O bandwidth. Hence, 8 MPI tasks are used to store checkpoints in the C/R model.

The second experiment assesses the effect of weak scaling on aggregate I/O bandwidth for different sizes of aggregate data transfer per node. 8 MPI tasks are used to perform I/O on a node and

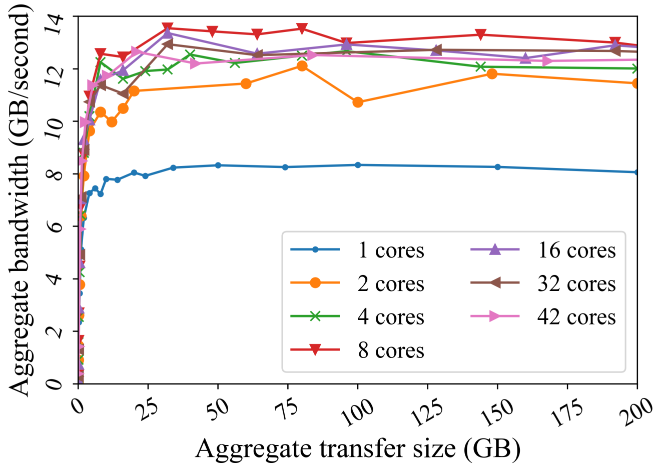


Figure 2.4 I/O performance on single compute node

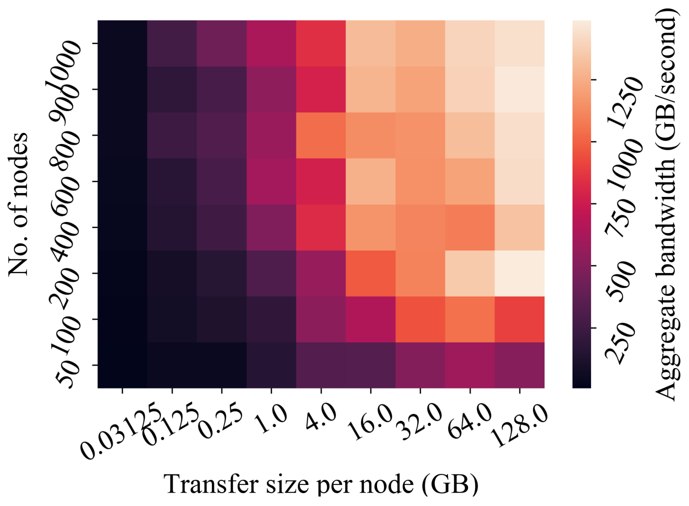


Figure 2.5 Impact of scaling on I/O bandwidth

its aggregate bandwidth is averaged over 10 runs. Effectively, the I/O performance of the $GPFS^{TM}$ parallel file system is modeled. Figure 2.5 shows the effect of scaling (nodes on the y-axis, transfer size on the x-axis) on aggregate I/O bandwidth (indicated by the heat map). For weak scaling, a fixed data size (each column) is exposed to an increasing number of nodes. We increase the data size along the x-axis and construct the I/O performance matrix. In our simulation, this performance matrix is used to calculate the time required to store checkpoint data in the PFS. Our simulation is based on the assumption that the aggregate bandwidth of a job is not affected by the I/O traffic generated by other running applications for now. I/O congestion will add more overhead for the non-frequent and failure prediction driven proactive checkpoints (safeguard and *p-ckpt*) as they checkpoint to the PFS directly, but not for the asynchronous periodic checkpoints from BBs to PFS resulting in minimal impact on performance overhead across all the C/R models. Adding the effect of background traffic impacts the checkpoint overhead across all models. For evaluation purposes, we assume the same performance matrix for the I/O read operations. PFS write achieves better throughput than read because data is cached. But checkpoints must be committed to the PFS before recovery. Hence, our I/O experiments use fsync() to purge caches. Further, as stated in Section 2.2, all nodes recover checkpoint data from BBs, except for the new replacement node in case of non-handled failures. This reduces PFS reads to a single node and thus no longer results in PFS contention, i.e., I/O performance is well below the thresholds of the aggregate scenario as discussed before. So recovery mainly depends on BBs speed, PFS is not the bottleneck.

2.5 Impact of Lead Time Variability

We simulated the execution of six real-world scientific applications listed in Table 2.1 to assess the impact of prediction lead time variability. Previous works [Wan17; Tiw14] use these application characteristics with the OLCF's Titan supercomputer as their platform. Since our experiments are based on Summit, we scale up the checkpoint size for each application proportionately to the change in DRAM size using Eq. 2.3.

$$Size_{new} = \frac{Size_{old} * \#Nodes_{new} * DRAMSIZE_{new}}{\#Nodes_{old} * DRAMSIZE_{old}} \tag{2.3}$$

To generate failures, we use the Weibull distribution parameters of Table 2.3 for OLCF's Titan in place of Summit's because of unavailability of the latter. A total of 1000 simulation runs were performed and then averaged.

We performed our analysis using three existing C/R models:

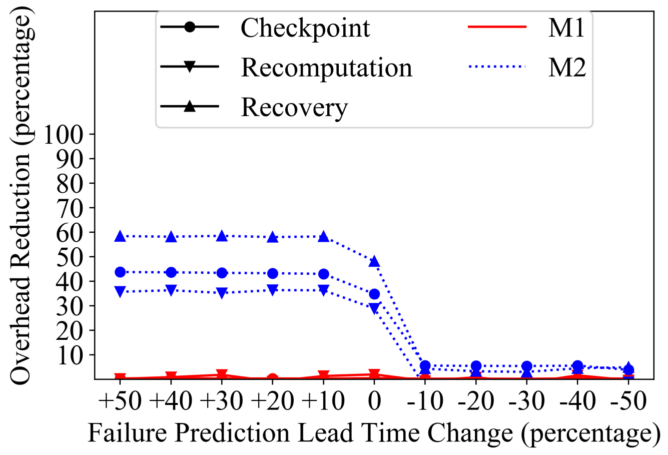


Figure 2.6 Impact of lead time variability on safeguard checkpointing and LM for application CHIMERA

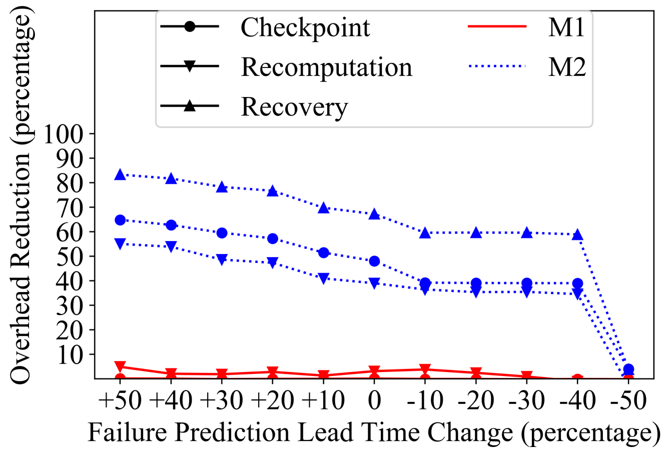


Figure 2.7 Impact of lead time variability on safeguard checkpointing and LM for application XGC

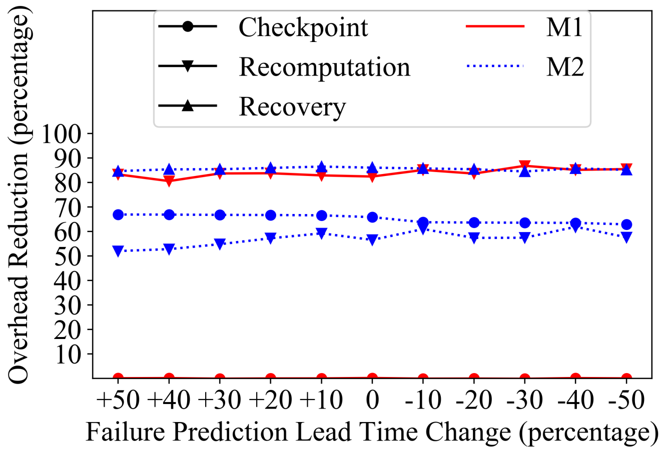


Figure 2.8 Impact of lead time variability on safeguard checkpointing and LM for application POP

Table 2.1 HPC workload characteristics

Application	Number of Nodes	Checkpoint Size (GB) on Summit	Computation Time (hour)
CHIMERA	2,272	646,382	360
XGC	1,515	149,625	240
S3D	505	20,199	240
GYRO	126	197.2	120
POP	126	102.5	480
VULCAN	64	3.27	720

- *Model B:* Periodic checkpointing + No prediction (base model);
- *Model M1:* Periodic checkpointing + Failure prediction & analysis model + Safeguard checkpointing; and
- *Model M2:* Periodic checkpointing + Failure prediction & analysis model + Live migration.

Both models M1 and M2 are driven by failure predictions to perform proactive actions to avoid losses due to failures. Model M2 represents the LM-C/R model [Beh20] and starts the LM process with adequate time before failure. Model M1 [Bou13] performs just-in-time checkpoints or safeguard checkpoints before a failure.

Figures 2.6, 2.7 and 2.8 illustrate the impact of prediction lead time variability on the applications (results for S3D, VULCAN, and GYRO omitted as they behave similarly to POP). The curves represent the percent change of overhead for each phase of M1 (red) and M2 (blue) relative to the base model B (y-axis) over percent lead time variation (x-axis). When lead times are varied, failure prediction timing is impacted. For example, with a 50% increase in lead time, failures are predicted 1.5x earlier than the original lead times. At 0% (y-axis), the overhead remains unchanged, at 100% the overhead is completely removed, i.e., higher is better. The phase of each model is indicated by the legend and defined as follows:

•*Checkpoint Overhead:* Duration for which application execution is blocked for checkpointing.

•*Recomputation Overhead:* Duration to recompute the portion of execution that was lost due to a failure.

•*Recovery Overhead:* Duration to recover from all failures.

Observation 1: Model M2 shows moderate improvement in reducing resilience overhead when lead times increase for large applications, but its performance diminishes once lead times are shorter than their reference values. In contrast, M1 reduces recomputation overhead by a larger amount than M2, but only for the smallest of applications; other overheads, and recomputation for larger applications remain unchanged.

For the large applications, CHIMERA and XGC, safeguard checkpoints (M1) do not add any benefit while they eliminate 85% of recomputation cost for smaller applications (even for a 50% decrease in lead time) and tolerate the impact of lead time variability. For S3D, in particular, M1's recomputation cost reductions gradually decrease from 77% (for 50% increased lead time) to 50% reductions (for 40% decreased lead times) and evaporates with further decrements in lead times. Safeguard checkpoints (M1) have no impact on checkpoint and recovery overhead regardless of application size.

We observe a more differentiated pattern under model M2. The support of LM in M2 reduces all types of overheads and changes with lead time variability at different rates depending on application size. For the largest application, CHIMERA, M2 sees all types of reductions rise by 8-10% (for a 10% increase in lead times relative to the reference) resulting in 35%-60% savings over the base model, and then remaining stagnant for longer leads. However, a mere 10% decrease in lead times diminishes all types of benefits provided by LM in M2. Similarly, for XGC, the second largest application, benefits for all types of reductions gradually increase with longer lead times, and these benefits rise at a faster rate than for CHIMERA. With a decrease in lead time, these benefits diminish only after lead times decrease by 50% or more. For smaller applications, M2 provides consistent reductions in all types of overheads that are not affected by lead time variability.

To understand the impact of lead time variability on M1 and M2, we define two terms: FT latency and FT ratio. FT latency is the time required by M1 or M2 to complete its proactive action to mitigate failures. FT ratio is the ratio of successfully mitigated failures to the total number of failures for an application. Application size and FT latency are two key factors that impact the performance benefits in M1 and M2 when lead time is varied. Table 2.2 represents the FT ratio in M1 and M2 for CHIMERA, XGC, and POP under varied lead times. As application size increases, both M1 and M2 require larger lead times to mitigate failures. So there is a drop in FT ratio for a lead time reference resulting in decreasing overhead reductions. This drop is also seen with increased reference lead times. This suggests that both M1 and M2's FT latencies are too high for large applications. Also, a decrease in lead time brings the FT ratio for M2 to near zero for large applications (CHIMERA and XGC) resulting in near-zero overhead reductions. However, since M2's FT latency is lower than M1's, it results in a higher FT ratio in M2 for large applications. In contrast, for smaller applications, the FT ratios remain similar and unchanged for both M1 and M2.

This experiment illustrates that lead time variability can have a severe impact on failure prediction-assisted fault tolerance solutions. First, Safeguard Checkpointing (M1) fails at providing any benefits for large applications and only provides reductions in recomputation overhead for smaller applications. Second, a small decrease in lead time can reduce the performance benefits of LM (under M2) for large applications. Given these results, our proposed p-ckpt solution aims to tackle these challenges of short lead times as described in the following section, followed by an evaluation with the same experimental methodology as discussed so far.

2.6 Priority-based Coordinated Checkpointing

In this section, we describe the overall design of our priority-based coordinated checkpointing method, *p-ckpt*, and the *hybrid p-ckpt* model. The core idea behind *p-ckpt* is that it applies coor-

Table 2.2 FT Ratio for applications under M1 and M2

Lead Time Change	**FT Ratio**					
	CHIMERA		***XGC***		***POP***	
	M1	M2	M1	M2	M1	M2
+50%	0.007	0.57	0.04	0.83	0.84	0.85
+10%	0.006	0.57	0.04	0.69	0.82	0.85
0%	0.006	0.47	0.04	0.66	0.84	0.85
-10%	0.004	0.04	0.04	0.58	0.83	0.86
-50%	0	0.04	0.009	0.04	0.83	0.85

dination among the nodes within an application before checkpointing to the PFS. It supports the prioritization of vulnerable nodes during the checkpointing to guarantee them contention-free access to the PFS. The *hybrid p-ckpt* model orchestrates *p-ckpt* with another proactive choice LM. However, LM is the preferred choice in our C/R model over *p-ckpt* as it allows the application with a vulnerable node to continue its execution while its pages are being copied to a replacement node [Wan08]. Further, checkpointing/restarting (to/from PFS) is more costly than LM in terms of network traffic for medium to large applications.

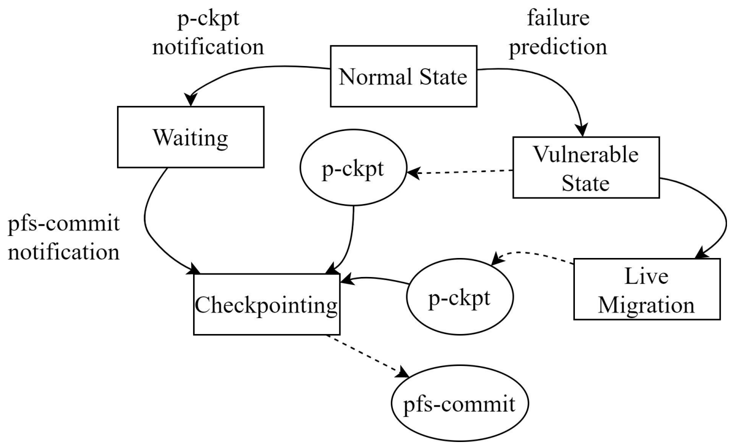


Figure 2.9 State diagram of a node in *hybrid C/R model*

Figure 2.9 depicts the state transitions of a node in the *hybrid p-ckpt* model. The square boxes with solid lines represent different states of a node. The ellipses with dotted transitions represent notifications as required. The solid arrows represent state transitions. When a failure is predicted, a node transitions from the normal state of periodic computation and checkpointing to the vulnerable state. In this state, based on the predicted lead time, a decision is made on the proactive action. If there is enough time to migrate the process from the vulnerable node to a new and healthy node, then the live migration process starts. Otherwise, the vulnerable node sends a *p-ckpt* notification to all other nodes and the *p-ckpt* process begins. When a *p-ckpt* notification is received, healthy nodes transition to the waiting state and wait for the vulnerable nodes to finish checkpointing to the PFS. Once the vulnerable nodes finish storing their state to the PFS, they broadcast the *pfs-commit* message to all other nodes within the application. When the healthy nodes receive this notification, they proceed with checkpointing to the PFS. The *p-ckpt* process is implemented with node-local priority queues, where vulnerable nodes with lower lead time to failures have higher priority while all healthy nodes have equal lower priorities. When live migration is in progress and another failure prediction occurs with lower lead time, live migration is aborted and the *p-ckpt* process begins (see state diagram).

P-ckpt performs a few global synchronizations and broadcast operations, which adds performance overhead. However, these operations are in the order of microseconds on Summit [Vaz18]. A global barrier with 2048 nodes takes only $\approx 8\mu$secs. We do not account for these small overheads during simulation. Further, the *p-ckpt* threads run only when a *p-ckpt* is taken but otherwise do not impact applications during execution. LM's execution interleaves with application execution. However, the overhead is quite low adding just 0.08-2.98% in runtime during live migration [Wan08].

2.7 Evaluation

The simulator used in Section 2.5 is also used for the evaluation of two new models as below relative to the same base model B as before:

- *Model P1:* Periodic checkpointing + Failure prediction & analysis model + *p-ckpt.*
- *Model P2:* Hybrid of periodic checkpointing + Failure prediction & analysis model + *p-ckpt* + LM.

Model P2 combines two different proactive fault tolerance techniques, LM and *p-ckpt.* The objective is to showcase our contributed models' benefits over fault tolerance models in prior work.

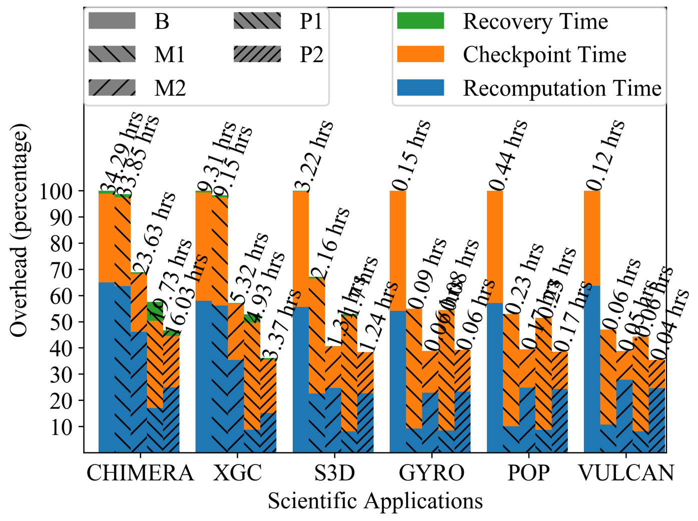


Figure 2.10 Reduction in overhead for Summit under Titan's Failure Distribution

No prior work combined M1+M2, and benefits may be limited for large applications (CHIMERA and XGC) as M1 is ineffective for large applications (see Section 2.5).

Figure 2.10 depicts for each application (x-axis) the overhead of fault tolerance in percent (y-axis) normalized to the base model (B) with periodic checkpoints (first bar) compared to failure prediction models M1, M2, P1 (*p-ckpt*) and P2 (*hybrid p-ckpt*). All models are annotated with rounded total overhead (in hours) on top of each bar. To test the robustness of our C/R model, we applied three different failure distributions from systems referenced in Table 2.3 [Wan17; Tiw14]. Here, we make the assumption that OLCF's Titan's failure distribution applies to Summit, i.e., Figure 2.10 depicts the overhead distribution for Summit under Titan's failure distribution.

Observation 2: *p-ckpt* (P1) and *hybrid p-ckpt* (P2) help reduce application overhead over the base model by ≈42%-55% and ≈53%-65% on Summit, respectively.

Table 2.3 Weibull distributions for failure generation

HPC System	Shape	Scale
LANL System 8 (164 nodes)	0.7111	67.375
LANL System 18 (1024 nodes)	0.8170	6.6293
OLCF Titan (18868 nodes)	0.6885	5.4527

In related work [Beh20], the LM-C/R model (M2) was guided by failure prediction and reduced the application overhead by ≈31%-61%. This reduction was due to the assistance of LM. The safeguard checkpoint model (M1) by Bouguerra et al. [Bou13] when driven by lead time-based failure prediction, reduced overall application overhead by ≈0-52% without providing any benefits for large applications. With *hybrid p-ckpt*, we observe a significantly higher reduction in cost, by ≈53%-65% (see Figure 2.10), than in [Beh20; Bou13]. The savings can be attributed to a combination of prioritized coordinated checkpointing (*p-ckpt*) against failures with short lead times (model P1) and lower failure rates due to prediction and successful mitigation via LM (model P2). The assistance of *p-ckpt* alone brings a ≈42%-55% reduction in application overhead (see Figure 2.10), which is higher than model M2 for large applications. Table 2.4 represents the FT ratio in P1 and P2 for CHIMERA, XGC, and POP under varied lead times. As can be seen, the lower FT latency of *p-ckpt* allows both P1 and P2 to obtain a higher FT ratio compared to models M1 and M2 (see Table 2.2). Model M1 cannot handle failures with short lead times for large applications with safeguard checkpoints, and its FT ratio remains near zero. However, *p-ckpt* successfully handles such failures as it commits the checkpoint on the vulnerable nodes without any congestion in a prioritized manner. While M2's LM yielded an FT ratio of 0.5 and above for the base lead times and above for large applications, *p-ckpt* pushed the FT ratio in P1 and P2 even higher, resulting in better overhead reductions. Notice that the FT ratios for P1 and P2 are almost equal for all the applications. That means both P1 and P2 can handle an equal amount of faults, but the overhead reduction difference between them is significant, as discussed later.

Table 2.4 FT Ratio for applications under P1 and P2

Lead Time Change	**FT Ratio**					
	CHIMERA		***XGC***		***POP***	
	P1	P2	P1	P2	P1	P2
+50%	0.84	0.83	0.85	0.84	0.88	0.86
+10%	0.76	0.76	0.84	0.84	0.87	0.85
0%	0.70	0.69	0.84	0.83	0.86	0.85
-10%	0.67	0.67	0.84	0.84	0.84	0.87
-50%	0.36	0.37	0.84	0.84	0.86	0.86

The stacked bars break down overhead that can be attributed to checkpointing and, after a failure, recovery to reload a checkpoint plus recomputation time to catch up with the execution to the point of failure. Notice that recovery overhead is negligible for all the models except for P1. This is due to our proactive checkpointing model, where all nodes commit their checkpoint to the PFS bypassing the BBs unlike regular checkpointing. A mitigated failure by a proactive checkpoint takes longer to recover, whereas failures unhandled are recovered faster with the assistance of BBs. We observe that recovery contributes ≈2.5%-6% of total overhead for P1 compared to less than 1% for other models.

Observation 3: Both *p-ckpt* and *hybrid p-ckpt* can tolerate the impact of prediction lead time variability better than prior models for large applications.

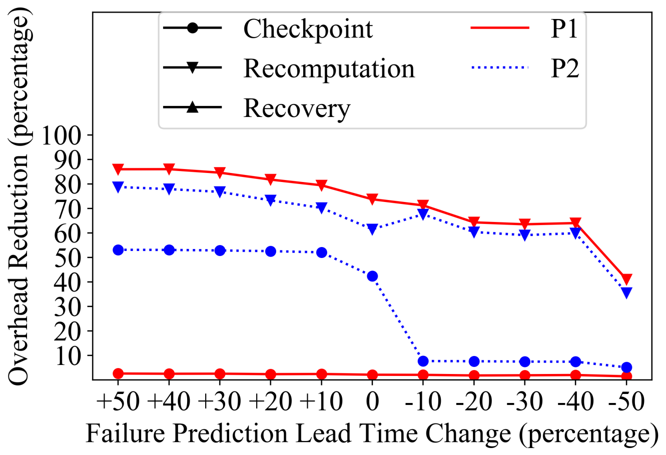


Figure 2.11 Impact of lead time variability on *p-ckpt* and *hybrid p-ckpt* for applications CHIMERA

Figures 2.11, 2.12 and 2.13 assess the impact of varied prediction lead time on models P1 and P2 for all the applications (results for S3D, VULCAN, and GYRO omitted as they behave similarly to POP) with the same x- and y-axes as in Figure 2.6 before. *p-ckpt* (P1) does not provide any additional benefits for recovery and checkpoint overheads like the M1 model (Section 2.5). However, for the largest application CHIMERA, it produces more recomputation overhead reductions than M2 and P2 and can tolerate up to a negative 50% change (i.e., reduction) in lead times while still providing some savings in recomputation relative to the base model due to the prioritization of vulnerable nodes. In contrast, Model M1 (safeguard checkpoints) does not provide performance benefits for

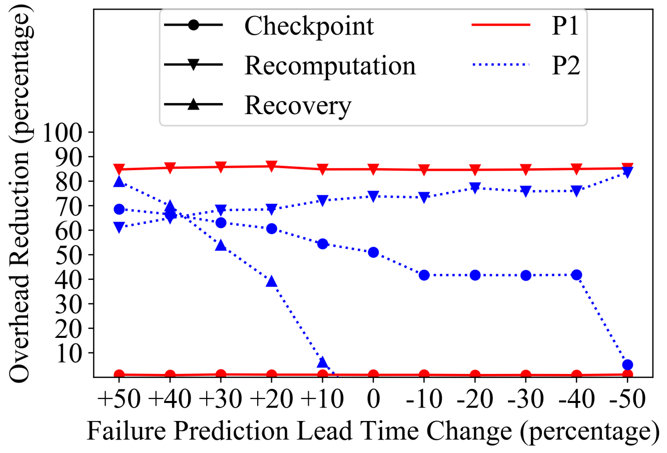


Figure 2.12 Impact of lead time variability on *p-ckpt* and *hybrid p-ckpt* for applications XGC

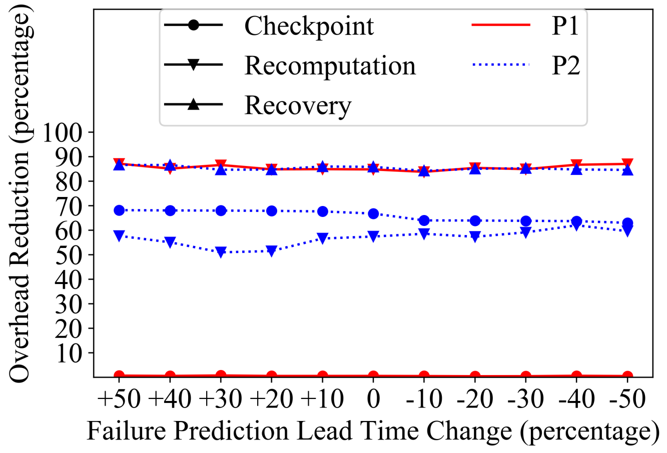


Figure 2.13 Impact of lead time variability on *p-ckpt* and *hybrid p-ckpt* for applications POP

CHIMERA, and M2's benefits diminish when lead time decreases by 10% relative to the reference. For XGC, P1 nearly eliminates the entire recomputation overhead regardless of lead time variations. In contrast, M2's performance benefits diminish with a 50% reduction in lead time while M1 is not effective at all.

The overhead reduction pattern for checkpointing in model P2 (*hybrid p-ckpt)* with respect to lead time changes follows model M2 for both CHIMERA and XGC. The pattern for recomputation overhead follows M2 largely when lead time increases, but follows P1 when the lead time shrinks. With the support of coordinated prioritized checkpointing (*p-ckpt*), P2 achieves a similar recomputation overhead reduction pattern as model P1 gaining a significant advantage over model M2. For large applications, both models, P1 & P2, not only achieve better recomputation overhead reductions, but increase their tolerance against prediction lead time variability. Further, because of our checkpointing model (see Sec. 2.2) in *p-ckpt* and the recovery process after proactive checkpoints, reductions in recovery overhead for P2 are not seen for XGC when lead time is less or equal to the reference. These reductions completely diminish for CHIMERA. Patterns for P1 and P2 for smaller applications follow M1 and M2, respectively.

Finding: *Variability in prediction lead time has a significant impact on the performance benefit of prediction-based C/R models, and our hybrid p-ckpt model outperforms prior related models under such circumstances.*

Observation 4: *p-ckpt* is more effective for large applications compared to LM. Higher lead times favor LM; conversely, when lead times are low, *p-ckpt* takes over.

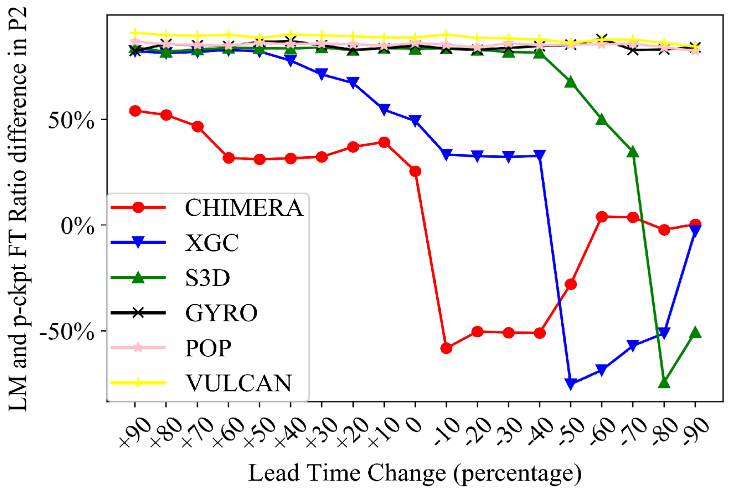


Figure 2.14 Difference in LM and p-ckpt FT ratio in P2 model

Figure 2.14 demonstrates the difference in FT ratio by LM and *p-ckpt* in percent (y-axis) in model P2 over lead time variation (x-axis) for all the applications. In this experiment, the lead time variation range was within (-90%, +90%) expanding the earlier range (-50%, +50%). If the percent difference is positive, then LM is the dominant proactive choice; otherwise, *p-ckpt* is more dominant. As can be seen, for smaller applications, the FT ratio difference between LM and *p-ckpt* remains consistently high (above 75%) across the lead time variation range. Since LM is the preferred choice ahead of *p-ckpt* and its FT latency is small enough, it can tolerate the lead time changes for smaller applications. When application size increases, the FT ratio difference between *p-ckpt* and LM decreases for the base lead time (0% change in lead time). That means *p-ckpt* is more effective for large applications compared to LM because of its lower FT latency. For larger applications, as lead times decrease, the dominance of *p-ckpt* as the proactive choice increases. *p-ckpt*'s dominance over LM is seen earlier for the largest application like CHIMERA followed by XGC and S3D. As lead time changes become negative, *p-ckpt* completely takes over LM before the FT ratio difference reaches zero as lead times completely diminish.

Observation 5: Under *hybrid p-ckpt* (P2), checkpoint overhead is reduced by ≈42%-70% across applications. In contrast, LM (M2) results in reductions of 34% compared to P2's 42% for the largest application.

Even though both P1 and P2 yield equal FT ratios (Table 2.4), P2 performs better than P1 in reducing overhead as LM helps model P2 to reduce checkpoint overhead. There is a negligible change in time spent in storing checkpoints under model P1 because of the adaptive nature of our checkpoint model. That means the schedule of checkpoints is variable in our model depending on the factors such as the time and location of failure prediction, and the proactive action chosen. For example, the scheduled checkpoint changes due to a *p-ckpt* triggered by and completed before a predicted failure. What is more significant is the reduced failure rate resulting from the failure analysis model, which yields a ≈42%-70% decrease in checkpoint overhead in the *hybrid p-ckpt* (P2) model. Further, for the largest application, CHIMERA, P2 reduces checkpoint overhead by 42% compared to just a 34% reduction by M2. Even though LM in both M2 and P2 have the same configuration, the assistance of *p-ckpt* helps P2 in completing the execution earlier (2% faster than M2) and thus reducing checkpoint overhead.

Observation 6: In the presence of frequent faults, applications can suffer higher recomputation overhead with *hybrid p-ckpt* compared to *p-ckpt*.

For all the applications, the recomputation overhead increases under (compare blue bars between P1 and P2 in Figure 2.10 and Figure 2.15) due to the inclusion of LM and elongated checkpoint intervals derived from our extended failure analysis model as per Eq. 2.2. The reduced failure rate increases the optimal checkpoint interval by ≈54%-340%, which indirectly impacts the computation losses due to failures that could not be predicted or avoided even if predicted in advance. The elongated checkpoint interval increases the hours of computation loss when failures are not proactively avoided. P2 experiences a ≈11%-27% increase in recomputation overhead relative to the base model when compared to P1. However, P2's loss in performance benefits is compensated by the reduced checkpoint overhead. This gives rise to the requirement of a careful selection of the C/R model for fault tolerance.

Recommendation: *Based on the analysis in observations 4 and 6, we suggest that HPC systems with a high fault rate and low lead times should utilize p-ckpt (P1) for large applications with short runtimes because of its ability to handle failures with short lead times and reduced computation loss derived from more frequent checkpointing. In contrast, applications with long runtimes should use the hybrid p-ckpt (P2), irrespective of size and failure rate, as checkpoint overhead can eclipse the recomputation overhead.*

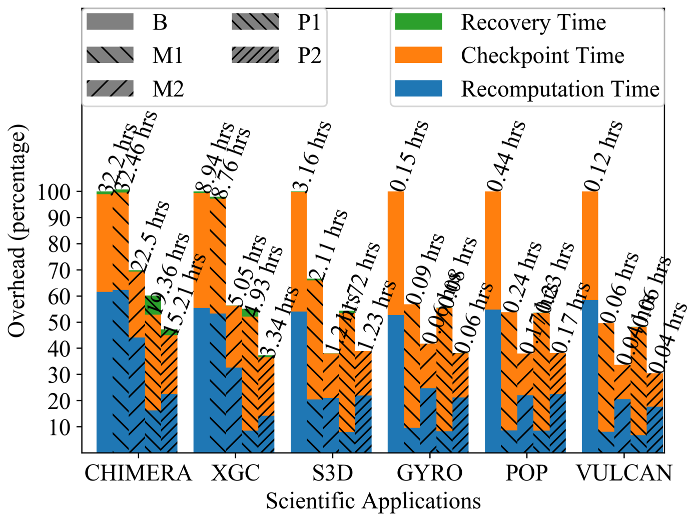


Figure 2.15 Reduction in overhead for Summit under LANL System 18's Failure Distribution

Observation 7: Reductions in overheads for model P2 are robust across different Weibull failure distributions.

Figures 2.15 and 2.16 depict the reduction in overhead on the same x- and y-axes as before (Figure 2.10), yet for Systems 18 with its failure distribution. The reduction in overhead follows a similar pattern for all three failure distributions. For LANL System 8, the decrease in overhead is ≈44%-73% while System 18 results in ≈52%-69% reduced overhead. Furthermore, the same pattern

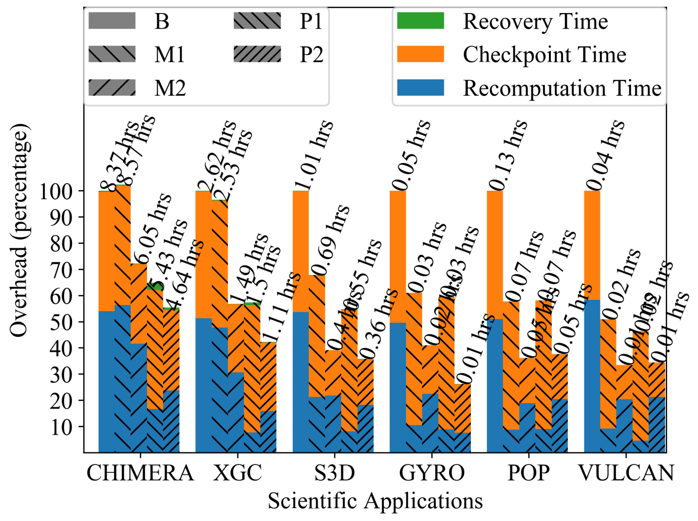


Figure 2.16 Reduction in overhead for Summit under LANL System 8's Failure Distribution

of increasing gains with decreasing checkpoint sizes is observed. This result is significant as it demonstrates that our model is robust and generalizes to other failure distributions. *In principle, our C/R model can be deployed on any HPC system that supports BBs, LM, and failure analysis plus prediction. It also shows that orchestrating failure prediction within a C/R model to drive decisions about when and how to checkpoint and when to live migrate reduces the impact of failures and shortens application execution over simpler failure models.*

Observation 8: The larger an application's checkpoint size is, the larger the advantage of *p-ckpt* over LM will be.

As mentioned in Sec. 2.2, we assume that the amount of data transferred for successful LM is three times that of the checkpoint data size per process. To understand how this factor impacts the performance comparison of LM (M2) and *p-ckpt*, we varied the amount of data transfer for LM and created multiple models designated with M2-*, where * indicates the factor of checkpoint data for transfer. Figure 2.17 shows the impact of varying transfer size for LM. The horizontal bars represent overhead reductions (similar to Figure 2.10) for all the models (B, P1, and M2-*) along the x-axis for three applications on the y-axis. We observe that for large applications (CHIMERA and XGC), *p-ckpt* (P1) performs better than LM (M2) overall until the LM transfer size becomes 1x and 2.5x times the checkpoint size, respectively. For smaller applications, LM always performs better than *p-ckpt*. Furthermore, reductions in recomputation overhead for *p-ckpt* (P1) are significantly larger than for LM (M2).

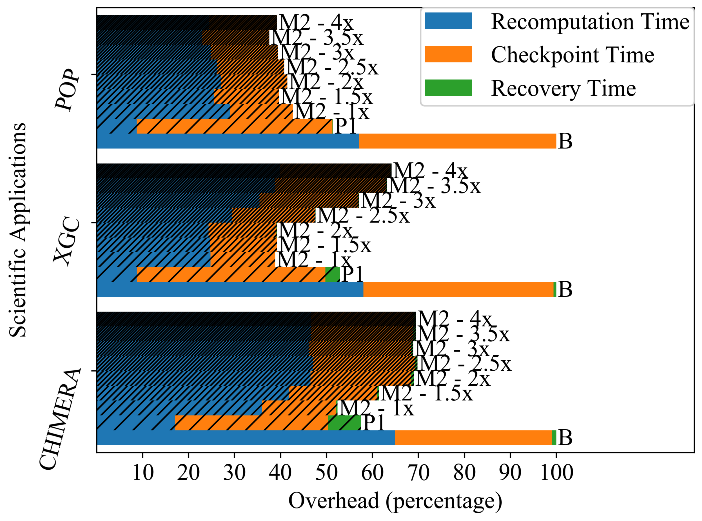


Figure 2.17 Reduction in overhead for LM vs *p-ckpt*

Based on this analysis, we provide an analytical model to compare LM and *p-ckpt*. We observe that LM (M2) reduces checkpoint overhead significantly (Observation 5), whereas *p-ckpt* (P1) yields better recomputation reductions than LM (Observation 4). For *p-ckpt* to perform better than LM, the difference in recomputation overhead reductions between *p-ckpt* and LM must be greater than the checkpoint overhead reduction by LM. This is captured by Eq. 2.4. Notice that we consider the recovery overhead in model P1 as negligible.

$$ckpt^{LM}_{reduction} < \left(recomp^{P-CKPT}_{reduction} - recomp^{LM}_{reduction}\right) \tag{2.4}$$

The first term can be expressed as Eq. 2.5, where the first term represents the total checkpoint overhead in the base model (B) and the third term represents a fractional reduction in checkpoint frequency due to LM (see Eq. 2.2).

$$ckpt^{LM}_{reduction} = ckpt^{B}_{overhead} * \left(1 - \sqrt{(1-\sigma)}\right) \tag{2.5}$$

Similarly, $recomp^{P-CKPT}_{reduction}$ and $recomp^{LM}_{reduction}$ can be represented as $\left(recomp^{B}_{overhead} * \beta\right)$ and $\left(recomp^{B}_{overhead} * \sigma\right)$, respectively. σ and β represent the fraction of failures that can be handled by LM and *p-ckpt*, respectively, and $recomp^{B}_{overhead}$ represents the total recomputation overhead of model B.

The right side of Eq. 2.4 can be simplified as $recomp^{B}_{overhead} * (\beta - \sigma)$. If we consider a uniform distribution of lead times of failures, an equal inter-node network bandwidth and single node PFS write bandwidth (which is the case for Summit with 12.5 GB/sec and 13-13.5 GB/sec, respectively), then β can be expressed using Eq. 2.6. α is the ratio of LM's transfer size to checkpoint data size.

$$\beta = \frac{\alpha - 1 + \sigma}{\alpha} \tag{2.6}$$

Eq. 2.4 can be re-written as

$$\frac{\left(1 - \sqrt{(1-\sigma)}\right)}{\frac{\alpha-1+\sigma}{\alpha} - \sigma} < \frac{recomp^{B}_{overhead}}{ckpt^{B}_{overhead}} \tag{2.7}$$

Assuming application overhead is split in half between recomputation and checkpointing, Eq. 2.7 is further simplified to

$$\frac{\sigma + 1}{\sigma + \sqrt{(1-\sigma)}} < \alpha \tag{2.8}$$

Based on the constraint that the sum of $recomp^{LM}_{reduction}$ and $ckpt^{LM}_{reduction}$ must be less than $recomp^{B}_{overhead}$, $\sigma < 0.61$. Eq. 2.8 suggests that α must increase non-linearly as σ grows. Under the constraints of $0 <= \sigma < 0.61$, the LM transfer size to checkpoint size ratio implies $1.04 <= \alpha < 1.30$ for *p-ckpt* to perform better than LM.

Observation 9: All models (M1/M2/P1/P2) experience a steady decline in total overhead reduction as the false negative rate increases. However, LM-supported models (M2/P2) experience larger declines in recomputation overhead reductions than the safeguard checkpoint and *p-ckpt* models (M1/P1).

To observe the impact of false negatives, we kept the false positive rate constant at 18% (see [Das17]) and varied the false negative rate up to 40%. Models M2 and P2 observe a ≈91%-180% and ≈71%-174% decline in recomputation overhead reduction, respectively, when the false negative rate reaches up to 40%. However, M1 and P1 observe smaller reductions of ≈48%-54% and ≈35%-40%, respectively. This means they actually can handle a fewer number of failures with an increasing false negative rate. LM-assisted models, M2 and P2, overestimate the number of failures they can handle and keep the checkpoint interval larger than models M1 and P1 (see Eq. 2.2). It confirms our recommendation in Observation 6 that on failure-prone systems, P1 holds an advantage over P2. To improve P2, the failure prediction accuracy factor needs to be included in Eq. 2.2, which is part of our future work.

Drawbacks: We rely on simulation for evaluating the models as special privileges are required to reserve nodes on Petascale systems (e.g., Summit) for large-scale experiments. To mitigate this, our evaluated applications are compute-intensive and use I/O during checkpointing based on the real machine data from Summit with a validated I/O write performance model [Beh20]. Simulated failures are based on real-world HPC failure logs [Das18b; Das18c]. Some operations such as synchronization and broadcast introduce overhead, but these are negligible. A *p-ckpt* barrier with 2048 Summit nodes take 8 microseconds. We also ignore the overhead of failure prediction as Aarohi [Das20a] predicts failures within 0.31 msec.

Feasibility: Utilization of two different proactive options under a single FT model (P2) requires coordination among multiple software systems like LM, *p-ckpt*, and periodic checkpointing. Further, to use a different proactive choice for individual applications, LM requires a global system view to avoid migrations that can create conflicts. *p-ckpt* applies to individual applications only by coordinating its processes. *p-ckpt* with a global system view is beyond the scope of this work, as is a complete implementation of the whole system.

2.8 Related Work

Several C/R solutions leverage failure awareness [Bou13; Tiw14; Geo15; Gar18; Wan08; Liu08]. Wang et al. [Wan08] monitor healthy nodes and migrate processes if the node's health deteriorates. However, the evaluation of their model excludes failures and only evaluates the efficiency of the live migration technique. Bouguerra et al. [Bou13] use proactive checkpoints upon failure prediction along with preventive checkpoints to reduce computation waste. They use FTI's [BG11] level 0 checkpointing strategy for proactive checkpoints and regular periodic checkpointing to PFS for periodic/preventive checkpoints. At level 0, checkpoint data of the failing node is stored on a neighbor node. Our model, for both *p-ckpt* and safeguard checkpoint, mandates the checkpoint data of all the nodes to be committed to PFS. Bouguerra et al. [Bou13]'s proactive checkpointing adds 2%-6% overhead to checkpoint time, whereas the adaptive nature of our model limits *p-ckpt*'s overhead to less than 1%. Further, *hybrid p-ckpt* reduces the checkpointing overhead by ≈42%-70% due to reduced failure rate and faster execution completion. Bouguerra et al. [Bou13]'s model relies on the failed node to restart for recovery purposes, whereas our model utilizes reserved nodes. Bouguerra et al. [Bou13]'s model achieves a 22% reduction in total overhead due to proactive checkpointing compared to a ≈53%-65% reduction with our *hybrid p-ckpt* model. Recomputation overhead reduction in Bouguerra et al. [Bou13]'s model is 17%, whereas our model's impact on recomputation

Table 2.5 C/R model comparison

C/R Model	**Failure Awareness**	**Coordinated prioritized checkpoint**	**Safeguard Checkpoint**	**Periodic Checkpoint**	**Live Migration**	**PFS I/O Model**	**Failure Prediction**	**Async Ckpt. Interval**
Hybrid p-ckpt	Failure Lead Time Prediction	✓	✗	✓	✓	✓	✓	✓
Wang et al.	Health Monitoring	✗	✗	✗	✓	✗	✗	✗
Bouguerra et al.	Failure Lead Time Prediction	✗	✓	✓	✗	✗	✓	✗
Tiwari et al.	Failure Locality	✗	✗	✓	✗	✗	✗	✗
Behera et al.	Failure Lead Time Prediction	✗	✗	✓	✓	✓	✓	✗

overhead is a ≈56%-73% reduction. Tiwari et al. [Tiw14] increase the checkpoint interval until failure and skip selected checkpoints post-failure using a temporal distribution of failures. Our C/R model's uniqueness comes from the use of failure prediction that selects the best mitigation action based on lead time. Further, we developed *p-ckpt* that replaces proactive checkpoints and improves fault tolerance for predictions with short lead time, even for large applications. Tiwari et al. [Tiw14]'s evaluation platform is based on OLCF's Titan while ours is on Summit. Their scheme reduces checkpoint time up to 70%, ours by ≈42%-70%. However, our checkpoint overhead accounts for the commits to the BBs while theirs commits to the PFS. Further, their recomputation overhead offsets most of the gains made with checkpoint overhead reductions. In contrast, our model provides more overhead reductions with less recomputation. George et al. [Geo15] deploy partial replication of process sets and predict failures to change the replicated process group. Garg et al. [Gar18] exploit failure locality to schedule applications with higher checkpoint overhead during lower failure rates and applications with lower checkpoint overhead during higher failure rates. In contrast, our model relies on dynamically predicted failures. Behera et al. [Beh20] developed a C/R model with live migration and assessed its benefit under failure prediction. However, their work could not handle faults with low prediction time. Table 2.5 compares our C/R model with other C/R models and illustrates the uniqueness and comprehensiveness of our approach in contrast to prior work.

2.9 Summary

We developed a multi-level C/R model that provides fault tolerance by orchestrating failure prediction with proactive actions of coordinated prioritized checkpoints (*p-ckpt*) and live migration via prioritization along critical failure paths, which results in reduced application overhead by ≈53%-65% compared to a ≈31%-61% reduction by conventional LM-C/R.

Overall, proactive actions should resort to *p-ckpt* in an HPC system with short lead times and high failure rates for short-running, large applications. In contrast, *hybrid p-ckpt* should be used for long-running applications, irrespective of application size and system failure rate. Our *hybrid p-ckpt*'s coordination of multiple fault tolerance techniques through failure prediction is unprecedented while providing better tolerance against failures with short lead times.

CHAPTER

3 PREDICTIVE EXECUTION OF WORKFLOWS IN A HPC+CLOUD ENVIRONMENT

3.1 Introduction

Workflows represent a wide array of applications in data analysis, knowledge discovery, and complex simulations in both Cloud and HPC environments [al.23; al.21b; Roy22a; Roy22b]. Cloud workflows often handle big data in science and commerce (including many medical applications) while HPC workflows tend to involve multi-science numerical problems spanning different abstraction levels or are part of large device experiments, such as high-intensity lasers [She19; al.22a]. Many modern-day scientific and big data workflows are also growing in scale, resource requirement diversity, and complexity [al.18b].

Many studies have focused on scheduling workflows on the Cloud with deadline and cost constraints. However, data-intensive workflow execution on the Cloud incurs significant monetary cost. To avoid such a cost, users schedule workflows on HPC systems in multiple ways. Typically, Workflow Management Systems (WMS) split workflows into tasks and schedule them as individual jobs or through executors that already run as jobs while satisfying dependency constraints [al.19; al.17]. However, HPC jobs are delayed when resources are not allocated immediately. Entire workflows can be scheduled as a pilot job to reduce the wait times for individual tasks, but the larger resource requirement delays execution startup. We can avoid the startup delay for pilot jobs by scheduling the pilot job on a reservation with a special privilege, which needs to be approved by the system administrators and cannot meet the deadline constraint. Further, pilot jobs may cause resource wastage with over-allocation. Recent works [al.22c; al.22b] in converged computing indicate a growing interest in utilizing on-demand Cloud resources to address the ever-growing computation demand of HPC workflows.

Cloud Bursting (CB) is a technique primarily used by private enterprises to scale up resources in the Cloud on demand, in addition to their own permanent on-premises private Cloud [Guo14]. However, applying CB to workflow scheduling is different from the traditional CB for Cloud workloads. Traditional CB usually relies on future workload prediction to allocate or scale additional resources (bursting) on the Cloud for task offloading. However, this methodology has two drawbacks. First, HPC system schedulers are complex and employ advanced strategies such as backfilling for better resource utilization. Predicting workloads only characterizes users' job submission behavior, not a system's scheduling behavior. So task offloading to the Cloud — based on workload behavior — may not utilize HPC resources optimally as it cannot estimate the resource availability (RA) on an HPC system from batches of jobs submitted by users with inaccurately estimated job durations. CB is dependent on workload behavior, such as spikes or drops in user requests, to perform bursting, assuming dedicated and on-demand local resources, which are not guaranteed on HPC systems. Second, because of the dynamic generation of input data for workflow tasks and on-the-fly offloading tasks to the Cloud in CB, data needs to be migrated just before execution begins. The lack of guidance (which and when) for offloading tasks to the Cloud prohibits data from being preloaded before execution begins.

We contribute a novel combination of HPC and Cloud in a hybrid, orchestrated manner by splitting computations across domains for a workflow to reduce delays resulting from on-the-fly data movement. Specifically, we build an execution schedule for a workflow in a HPC+Cloud environment by predicting the unused HPC resources complemented by the guaranteed on-demand Cloud resources to meet the deadline. Concerted deterministic resource allocation and task assignment allow better utilization of unused HPC resources and preload the dynamically generated intermediate input data on HPC and Cloud to avoid execution delays. Further, we use deterministic resource allocation to compute Cloud cost by considering the required Cloud resources, storage, and inter-domain data transfer.

We developed an adaptive resource allocation/scaling method to reserve resources both on

Cloud and HPC before executing a workflow per a schedule. This is accomplished by requesting the determined number of resources on HPC and Cloud on an hourly basis, with a runtime limit of one hour, thereby allowing the HPC system scheduler to allocate limited resources instantly, typically for as many single-node requests as can be accommodated by the back-filling algorithm. These underutilized resources may otherwise remain idle and are hence considered free in our model. Without a bounded wait time for HPC resource allocation, we back up the requested HPC resources with an equivalent redundant allocation on the Cloud to avoid execution delays. As the requested HPC resources are allocated, the redundant backup Cloud instances are reduced (de-allocated). This migration of computation from the Cloud back to HPC allows us to limit cost for Cloud resources to a predictable level while satisfying deadline constraints.

In summary, we make the following contributions:

• We identify contemporary challenges of scheduling large-scale workflows exclusively on HPC systems and analyze their root causes.

• We propose and develop a novel solution for scheduling large-scale workflows deterministically in that both cost and deadline can have highly reliable guarantees that can be traded off. This is accomplished by predicting RA of an HPC system, which facilitates the determination of resource requirements split between HPC and Cloud for different deadlines and avoids execution delays due to on-the-fly data movement.

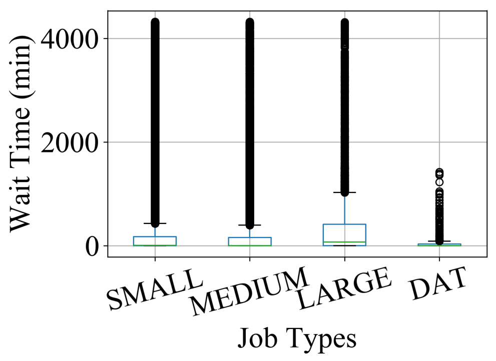
• We developed an elastic resource allocation/scaling algorithm to scale up/down heterogeneous resources on HPC and Cloud such that HPC resources are backed up by the Cloud to guarantee deadlines while avoiding RA mispredictions.

• We evaluated our method by developing a simulator for HPC+Cloud hybrid execution. We first validate our simulator by running a limited number of experiments in an HPC+Cloud setting to assess its performance. We subsequently use the validated simulation as the cost of Cloud execution in our runs would be prohibitive for the experiments we conduct. Our evaluation is based on factors such as meeting deadlines, correct estimation of cost, cost savings, and impact on other production jobs.

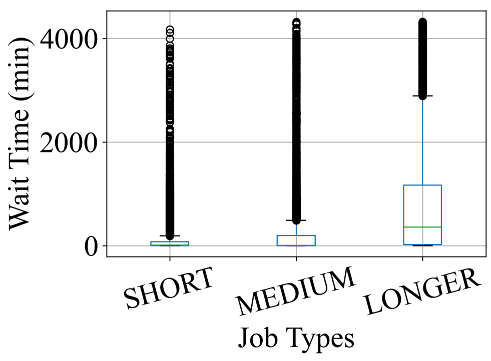
3.2 Background

To investigate the challenges and opportunities in scheduling large-scale workflows, we analyzed production jobs submitted to the Lassen supercomputer [LLN24a].

Data Collection: Lassen, a Top500 [Top] supercomputer installed at Lawrence Livermore National Laboratory (LLNL), comprises 795 compute nodes. Each node is equipped with an IBM Power9 CPU and four NVIDIA V100 GPUs. The job records from the IBM CSM database [al.20a] provide



(a) Wait time vs Job Size on Lassen



(b) Wait time vs Job Run Time

Figure 3.1 Challenges with Large Job Scheduling

detailed information about the job lifecycle, resource utilization, performance statistics, and other metrics. We used data from two years and two months [LLN24b], which includes production jobs submitted by real users and subject to the primary allocation algorithm. The jobs are categorized based on the requested number of nodes: small (1-7), medium (8-63), large (64-256), and DAT (256+). Notably, Dedicated Access Time (DAT) jobs are exempt from the primary allocation. Instead, DAT jobs are submitted with special privileges and are scheduled at a predetermined time.

To gain insights into the scheduling challenges, we examined the wait times for all job categories. Figure 3.1a illustrates the distribution of wait times in minutes for jobs waiting to run. We focused on jobs with wait times within three days and excluded those outside this range (only 2.2%) to improve visibility of the plots. The data reveals that DAT jobs have the shortest wait times (quartiles Q1 and Q3: 0-34) due to their privileged status. Small and medium jobs have wait time quartiles (Q1 and Q3): 0-171 and 0-158, respectively. In contrast, large jobs exhibit the longest wait times with quartiles (Q1 and Q3): 0-411, primarily due to their increased node requirements and scheduling under the primary allocation algorithm. Next, we categorized the jobs as SHORT (<2 hours), MEDIUM (2-6 hours), LONG (> 6 hours) according to their requested run-time limits and analyzed their wait times as shown in Figure 3.1b. We observe that as the requested time limit increases, the wait times also increase. SHORT and MEDIUM jobs have lower wait times quartiles (Q1 and Q3), i.e., 0-75 and 1-194 minutes, respectively, followed by LONG jobs with quartiles (Q1 and Q3) of 22-359 minutes.

These findings suggest that large jobs submitted without special privileges (non-DAT jobs) face significant wait times. Large jobs, in terms of both run time limit and requested node size, experience significantly higher wait times than other job categories. Our scheduling technique considers each single node with a one-hour time limit so that jobs can maximize resource allocation on HPC while offloading required execution to the Cloud. Further, the communication overhead between HPC and Cloud systems can cause significant delays in application execution [Ben15]. We address these challenges with our predictive and adaptive execution framework as described next.

3.3 System Design

3.3.1 Execution Model

In our execution model, the workflow tasks are processed on the executors launched by a WMS or workflow scheduler. Each executor runs on an allocated HPC or Cloud resource and executes tasks sequentially from an assigned stage in the workflow. Executors are limited to a one-hour time limit, as jobs with a small runtime limit are more likely to be allocated faster, as shown in Section 3.2. Before running out of their allocated hour, HPC executors checkpoint their task, where the checkpoint size is assumed to be equal to the input data size. In contrast, Cloud executors exit after completion of their current task, even after the one-hour time limit expires. We maintain two mutually exclusive queues per stage in the workflow: one for HPC and one for the Cloud. The Cloud and HPC executors pull a task for execution from its assigned stage's respective queue along with the required input data location. If the input does not exist in the same domain (HPC or Cloud), it copies the data before executing. An input data is kept alive on any domain until all the tasks dependent on the data complete execution. Shared filesystems on both HPC and the Cloud are used to store the intermediate output files, whereas the final output is stored on the HPC filesystem.

The WMS runs on a reserved node of the HPC system, where we store runtime information such as task queues, resource allocation details, and the storage location of the task's outputs. This information facilitates resource allocation/de-allocation and input data transfer between HPC and the Cloud.

3.3.2 Design

Figure 3.2 depicts an overview of the design and identifies the different modules of our scheduling framework and their relationship. Our framework consists of two phases, namely workflow schedule construction (green) and workflow execution (blue). To build a workflow schedule, our system needs the workflow's execution profile and future RA prediction. We assume that the execution profile is unknown to the user before scheduling. Hence, our framework runs a small-scale dynamic performance benchmark (DPB) for all the stages in the workflow on the platforms resident on HPC and Cloud. Next, we predict RA on the HPC system for a deadline using our developed RA

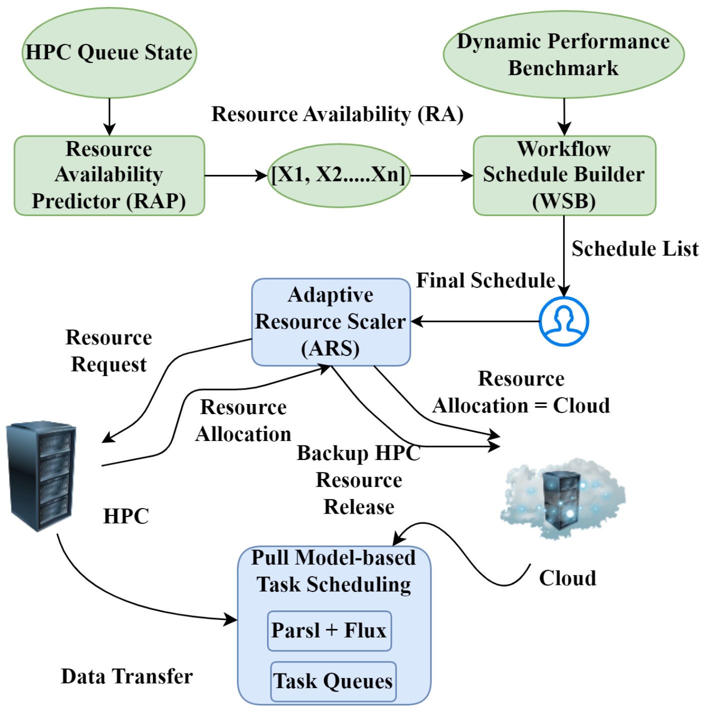

HPC Queue State
Dynamic Performance Benchmark
Resource Availability (RA)
Resource Availability Predictor (RAP)
[X1, X2.....Xn]
Workflow Schedule Builder (WSB)
Schedule List
Final Schedule
Adaptive Resource Scaler (ARS)
Resource Request
Resource Allocation
Resource Allocation = Cloud
Backup HPC Resource Release
HPC
Cloud
Pull Model-based Task Scheduling
Parsl + Flux
Task Queues
Data Transfer

Figure 3.2 Overall System Design

predictor (RAP) and feed it to the schedule builder (WSB) along with the DPB. The WSB calculates the additional Cloud resource requirements for each given RA to complete the execution before a given deadline. We also calculate the associated Cloud cost with each deadline replayed to the user as information to assist in the tradeoff between deadline and cost. Once the user selects an execution schedule, execution begins. Our Adaptive Resource Scale (ARS) allocates resources per the split between HPC and Cloud in the schedule on an hourly basis.

Our framework is built using Parsl [al.19] and Flux [al.18a] for resource allocation and communication. The Parsl library supports parallelization of Python and Bash functions with multiple execution models across various platforms, such as an HPC system, Cloud, and a local computer. We use Parsl to launch task executors on both HPC and Cloud. Flux is a workload and job management system that can be nested within another system scheduler or itself at multiple levels while providing various communication patterns among the instances of Flux. Our scheduler and the task executors use Flux capabilities to exchange scheduling information and relevant notifications. Next, we describe various components of our framework and their methodology in detail.

Resource Availability Predictor (RAP): The RAP is an integral part of our scheduling technique. The RAP takes the HPC system's job history as input and applies Machine Learning (ML) techniques to predict future RA. The output of the RAP is a vector, X, where each element x_i represents the minimum number of unused/free nodes during the i^{th} hour. End-to-end training and the prediction procedure are described in Section 3.4.

Dynamic Performance Benchmark (DPB): To estimate the resource allocation split between HPC and Cloud for a workflow, the WSB needs to know the comparative performance of each stage on multiple platforms of HPC and Cloud. The DPB provides the required performance model along with a stage's input/output size. It is constructed by dynamically running and recording the performance benchmark for all stages of the workflow on all platforms (including HPC and Cloud) before completion of either one hour of runtime or 5% of total tasks. Since HPC nodes may not be immediately available, we continue building the pending HPC benchmarks. However, we assume a default five-minute execution time for the missing benchmarks in the DPB for the initial schedule construction and cost estimation. Once HPC nodes are allocated and their performance benchmarks are constructed, they are integrated into the DPB. We then update workflow schedules dynamically as discussed later. Since building such a multi-faceted DPB is expensive, we execute these runs on HPC platform resources [AWS24] listed in Table 3.1. We assume that these are part of the Cloud along with associated costs.

Cloud Cost Model (CCM): The CCM represents the hourly cost of using on-demand platforms from a Cloud provider. We use multiple platforms, as shown in Table 3.1 to build the multi-faceted DPB. Each platform has an associated per-hour cost for running workloads on it. Usually, GPU platforms have higher per-hour reservation costs compared to CPU-only platforms. The CCM is static and does not change over time as we use on-demand Cloud resources.

Table 3.1 Cloud Cost Model

HPC Platform	AWS Platform	Cost ($/hr)	Compute
Epyc Rome	c5a.4xlarge	0.616	CPU
Intel Skylake	c4.4xlarge	0.796	CPU
Intel Broadwell	c5n.4xlarge	0.864	CPU
Nvidia RTX 2060 Super	g4ad.4xlarge	0.867	GPU
Nvidia RTX 4060 ti16g	g3.4xlarge	1.14	GPU
Nvidia RTX 4060 ti8g	g5.xlarge	1.006	GPU

Building Workflow Schedule: Given the RA, CCM, and DPB, the WSB utilizes the following steps to build a schedule to meet the required deadline with a targeted task size. The output schedule contains hourly information about how many resources are required to run the executors on HPC and Cloud to meet the deadline.

In our execution model, each stage is executed as a bag-of-tasks with input data dependency between stages. This results in a pipeline execution as seen in Figure 3.3, where a stage's execution is comprised of slots from start to end. Stages with parents face a delay in execution start because of dependencies, e.g., stage j in Figure 3.3. We apply depth-first search to find such delays to decide the start time for each stage. Further, there are infeasible slots of parent stages whose output tasks cannot be processed by their children due to a lack of available slots before the deadline, as shown for stage i. We remove such infeasible slots to determine the end times of each stage and the deadlines.

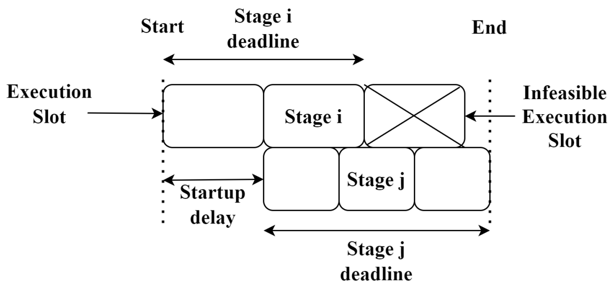


Figure 3.3 Pipelined execution of the stages in workflow

After deducing the start and end times, the next stage of schedule building is to allocate resources with constraints in Eq. 3.1 and Eq. 3.2. Eq. 3.1 states that the sum of hourly throughput, $TP(p)_i$, for each stage, p, in the workflow is greater than or equal to the task size to ensure all tasks are executed before the deadline D. This is complemented by Eq. 3.2, which ensure that two stages p and q with data dependency have almost equal hourly throughput so that the task pipeline of the workflow never remains empty to avoid resource idleness.

$$\forall p \in \mathbb{W}, \sum_{i=1}^{D} TP(p)_i \geq N \tag{3.1}$$

$$\forall p \in W, \forall q \in Parent(p),\ TP(p)_i \approx TP(q)_i \quad \text{for } i \in \{1,\ldots,D\} \tag{3.2}$$

To build the schedule for the entire workflow, we start with the stage with the highest Cloud cost per Eq. (3.3) to reduce allocation cost by preferring HPC resources. Cloud cost is derived from the sum of storage, data transfer cost, and the minimum execution cost on all the selected Cloud platforms, given by set R for stage p per Eq. 3.3.

$$storage\_cost(p) + xfer\_cost(p) + \min_{\forall r \in R} exec\_cost(p) \tag{3.3}$$

We start with the allocation of HPC throughput (number of slots available times number of nodes available) on an hourly basis in proportion to nodes in the predicted RA. Let TP_i^{HPC} be the HPC throughput during the i^{th} hour and TP^{HPC} be the total throughput available for the costliest stage. We calculate the hourly throughput target on HPC using Eq. 3.4, where T is the total number of tasks. If we cannot execute all T tasks solely on HPC, then tasks need to be offloaded to the Cloud. In that case, we distribute the Cloud throughput (i.e., the remaining tasks beyond the HPC target) among the hours where the allocated HPC throughput is under-allocated due to a lack of HPC resource availability. This is driven by our aim to have a balanced throughput during the execution. We skip the details of the Cloud target distribution for brevity. After the HPC and Cloud throughput targets are decided, the required amount of resources on HPC and Cloud can be calculated by dividing the throughput target TP_i^{HPC} by the number of available execution slots during the hour. Specifically, for Cloud, we choose the most cost-efficient resource from the DPB.

$$Target_i^{HPC} = min(T \times TP_i^{HPC} \div TP^{HPC}, TP_i^{HPC}) \tag{3.4}$$

Next, we allocate the targets for the other stages by traversing from the costliest stage and applying the conditions shown in Figure 3.4. Here, we calculate the throughput targets at the slot level for the immediate parents and children. In scenario A, a parent stage's execution slot is longer than that of its child. The tasks produced by a parent's slot should be processed next in the child's slots, which start executing during the parent's following slot (color-coded). In scenario B, the parent stage's execution slot is smaller than the child's. In this case, the child's slots should process all the tasks produced by the parent's slots that start during the child's previous slot.

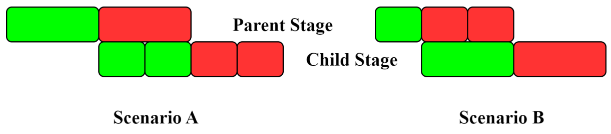


Figure 3.4 Execution slots and Throughput allocation

We calculate the hourly throughput by accumulating the throughput from the slots. Based on the hourly targets, we calculate the required allocation from HPC, if available, and Cloud. The final step is to combine HPC and Cloud allocations to form the workflow schedule. This method produces one schedule given the start time and deadline for a workflow.

Cost Estimation: After constructing a schedule, we estimate the cost of workflow execution by considering Cloud resource reservations in the schedule, deriving the Cloud filesystem operations, and data transfers between HPC and Cloud. Since HPC execution utilizes only otherwise unused single nodes via backfilling, its cost is considered free. The Cloud resource reservation cost is the sum of hourly costs of allocation for all stages in the workflow over the schedule, given the CCM. Further, we compute the amount of data read and written for the Cloud filesystem due to task execution. To estimate the data transfers and the associated filesystem operations, we compute the difference in Cloud throughput and HPC throughput between a stage and its parents. This gives us the direction and amount of data movement per stage. These calculations are done on an hourly basis and aggregated to be fed into the CCM model to get a dollar cost quote. The resource reservation and filesystem costs are modeled based on Amazon EC2 platforms and Amazon's EFS, respectively. We only account for Cloud to HPC data movement cost since data movement into the AWS cloud is free.

3.3.3 Data Preloading Strategy

Our splitting of workflow between HPC and Cloud may result in delayed execution of tasks when large input data does not reside in the same domain. This results in delayed workflow execution and increased cost because of resource idleness. As a mitigation, we built an adaptive preloading strategy to place input data near the targeted computation. To this end, we create the task-to-domain mapping as soon as a task spawns a stage in the workflow. The task-to-domain mapping indicates whether a task executes on Cloud or HPC. We rely on the domain's task load, i.e., task queue size/number of executors, for a stage to determine the mapping. If the load is less than 1, we map the task to the corresponding domain. To break ties, we prioritize Cloud to reduce resource idleness cost and data movement latency, both contributing to execution time savings. After the mapping, we copy the relevant input data to Cloud or HPC asynchronously without affecting computation. To facilitate this, we maintain two separate task queues, each for HPC and Cloud per stage in the workflow. Although the task-to-domain mapping dictates task enqueuing, our strategy is not strict with respect to the mapping. Idle resources can still execute tasks from the other domain's queue when that domain's load exceeds 1.

3.3.4 Adaptive Resource Scaler (ARS)

The second phase of workflow execution relies on the ARS to make adaptive and elastic resource requests for both HPC and Cloud per the workflow schedule selected by the user. The scaling mechanism of ARS executes at two different configurable intervals: long and short.

ARS at long intervals: ARS requests for resources on HPC and Cloud with a reservation time limit that is equal to this longer ARS interval based on the schedule constructed using the RA prediction. When a misprediction occurs, e.g., when over-estimating the available HPC resources, we may miss the deadline due to a lack of resources. Further, an imbalance in resource allocation for workflow stages can lead to resource wastage. When HPC resources are requested, there is no guarantee that they are allocated immediately, unlike Cloud resources with modeled availability at a $\approx$ two-minute delay [Mao12]. HPC resources are released automatically, whereas Cloud resources require manual termination.

ARS at short intervals: To avoid computation shortage from RA misprediction and delayed HPC allocation, we request additional Cloud allocations on top of the scheduled ones. These excess Cloud allocations mirror the scheduled HPC allocations at the longer intervals in the schedule. This allows us to begin the execution of the stages without waiting for HPC resource requests to be met and allocated. This also addresses the issue of mispredictions that overestimate the available HPC resources. After scheduled requests are made at longer intervals, ARS executes at short intervals (five minutes by default) to continuously check if HPC resource requests are allocated. As the requested HPC allocations are granted, an equivalent amount of additional Cloud resources is released as they are no longer needed during the remainder of the long interval. This incurs additional up-front Cloud reservation costs for backing up the HPC allocations in a schedule, yet only for a short time if HPC resources are granted; any remaining Cloud resources contribute to workflow allocation. This up-front Cloud cost is subject to evaluation in our study.

Rebalancing Resource Allocation: We also adapt our allocation strategy at long intervals by rebalancing the resource allocation based on the deficit (if any) on achieved throughput in the past. We keep track of the targeted throughput and the achieved throughput on an hourly basis and over-allocate Cloud resources as required.

3.4 Resource Availability Predictor (RAP)

The RAP is the key component that enables elastic workflow scheduling on HPC systems. By predicting resource availability, RAP constructs a schedule for a given workflow, providing informed decision-making capabilities regarding resource allocation and estimating the cost of execution. We describe the end-to-end build procedure of RAP on the Lassen HPC system, using data collected in Section 3.2.

3.4.1 Feature Data Set

HPC systems typically distinguish between two primary job classes for scheduling: wait and run. When a job is submitted, it first enters the wait queue. Once deemed ready to run, the scheduler removes the job from the wait queue, enqueues it in the run queue, and starts executing it. The wait queue tracks a job's requested number of nodes, submission time, and requested hours of runtime. Similarly, the run queue tracks a job's allocated number of nodes, execution start time, and time left to completion. The information in the run queue indicates the current number of nodes running and those expected to be released soon. Conversely, the wait queue reveals the current and future demand for nodes. By combining data from both queues, we can predict resource availability on an HPC system.

We record states of the run and wait queues instead of their complete snapshots, as this was sufficient in our experiments to make predictions. The captured state of the run queue represents the number of busy nodes expected to be available in the future. We use a one-dimensional vector data structure to represent this information up to 64 hours, with a maximum limit based on the fact that this is the maximum number of hours of any job requested in the data logs collected from the CSM database (likely also the upper bound per system configuration). The state of the wait queue represents the number of requested nodes for different hours of runtime. Given a maximum of 64 hours, we represent the state of the wait queue as a one-dimensional vector of length 64.

Example: Consider three jobs waiting in the Wait queue with a descriptor of (jobid, no. of nodes requested, no. of hours requested):

`(job4, 3, 1), (job5, 5, 5), (job6, 10, 2).`

Given these jobs, the wait queue state will be `[3,10,0,0,5]`. Similarly, consider three jobs in the run queue with a description (jobid, no. of nodes requested, no. of hours remaining):

`(job1, 3, 2), (job2, 5, 1), (job3, 10, 5).`

Given these jobs, the run queue state will be `[5,8,8,8,18]` with vector size 64.

The job submission and scheduling on Lassen is managed by IBM's LSF [IBM24] workload manager. LSF employs multiple scheduling policies, such as FCFS, Fair Scheduling, Backfill, SLA, and preemption, to efficiently accommodate users' resource requests. The Lassen supercomputer employs fair share-based scheduling that requires user information and their past usage. Since user information is unavailable, we assume that FCFS (primary) + Backfill (secondary) scheduling policies are applied to the jobs submitted. We simulate the production jobs history on our HPC simulator (see Section 3.5.1) and collect the feature data representing the temporal state of the wait and run queues. We also use the day of the week and the hour of the day as feature variables during ML training and prediction. Finally, we apply sampling to the collected feature data set with 30 and 60-minute frequencies.

3.4.2 Training and Results

To predict RA, we apply both time series and regression-based ML techniques to train our prediction models. Specifically, for time series-based ML, we use the Recurrent Neural Network (RNN)-based Long Short-Term Memory (LSTM) method [Yu19]. With LSTM, an input sample is comprised of the history (input steps) of states within the run and wait queues, including the day of the week and the hour of the day. The output indicates RA for up to 24 hours (output steps). Further, we use different combinations in a tuple consisting of (resample frequency, training loss function, and optimizer) during training as shown in Table 3.2.

Table 3.2 LSTM Training Parameters

Parameter	**Values**
Input Steps	[6, 12, 18, 24]
Output Steps	[6, 12, 18, 24]
Resample Frequency	[30, 60] (minutes)
Loss Function	[MAPE, RMSE, RMSLE, HUBER]
Optimizer	[SGD, ADAM]

We also trained ensembles of Decision Trees (DT) using Gradient Boosting via the XGBOOST library [Dev24]. XGBOOST parallelizes Gradient Boosting to train ensembles of DTs to improve performance while scaling. To train the DTs, we first transform the time series input samples into regression data. We do this by considering a snapshot of the run and wait queue states as an input sample and RA as an output sample (up to 24 hours). For DTs, we use combinations of output steps, resample frequencies, and loss functions (without MAPE) from Table 3.2 as training parameters.

To verify the prediction accuracy and compare the ML models, we use Mean Absolute Error (MAE) as a metric during validation. We consult Lassen's records of jobs from September 18, 2018, at 12 PM to November 18, 2020, at 11 AM. We use the records until May 18, 2020, 12 AM (18 months) for training of the models. The records from May 18, 2020, 12 AM until October 18, 2020, at 12 AM (5 months) are used for testing, resulting in an 80%/20% split between training and testing data. We observed that the MAE for LSTM (73.49-92.20) is higher than for DT (51.61-75.07) with the Huber loss function and a 60-minute resample frequency. Consequently, we choose DTs trained with Huber as the RA predictor for simulation/validation of our scheduling technique. We re-train the DTs by concatenating the training and testing data to improve accuracy. We reserve the unseen records from October 18, 2020, at 12 AM until November 18, 2020, at 11 AM (1 month) for simulation in Section 3.5.

3.5 Evaluation

3.5.1 HPC+CLOUD Simulator

To evaluate our solution, we developed a simulation framework utilizing Python-based SimPy [Tea20], a process-based discrete event simulation library. Given the prohibitive cost of conducting extensive experiments on commercial Cloud resources, we opted for simulation as an alternative. In this framework, we simulate an HPC system with wait and run queue managed by the FCFS+Backfill scheduling policy. Further, the task executors (both HPC and Cloud) are simulated using SimPy event generator functions, which pull tasks from the task queue and simulate the times of (a) execution derived from the DPB, (b) reading input files, and (c) writing output files. The scheduling algorithm on our simulator creates resource requests to the HPC scheduler for HPC executors and directly creates Cloud executors to simulate the execution of a workflow.

AWS Cloud Validation: We validated our simulator by running the OAI Analysis workflow (see Section 3.5) with our Flux+Parsl scheduling framework on an 80-node HPC cluster and AWS cloud. We ran the workflow with static resource allocations on the HPC cluster and AWS cloud with input data preloading support (see Section 3.3.2). We utilized GPUs on the HPC system (NVIDIA RTX 2060 Super) and CPUs on AWS EC2 (c4.large and c4.xlarge). We fed the traces from the execution to our simulator to run the workflow. Figure 3.5a shows the execution times (y-axis) of the workflow on the HPC-AWS Cloud hybrid platform and our simulator for task sizes ranging between 50 and 150 (x-axis), with annotations indicating the percentage difference between the two. The results show that our HPC+Cloud simulator resembles the workflow execution on the HPC-AWS cloud system with good accuracy.

3.5.2 Results

Experiments are conducted to simulate the HPC+Cloud hybrid execution of two real-world scientific workflows, OAI analysis and RNASEQ, on Lassen backed up by Amazon AWS cloud. We evaluate multiple scheduling algorithms, including ours:

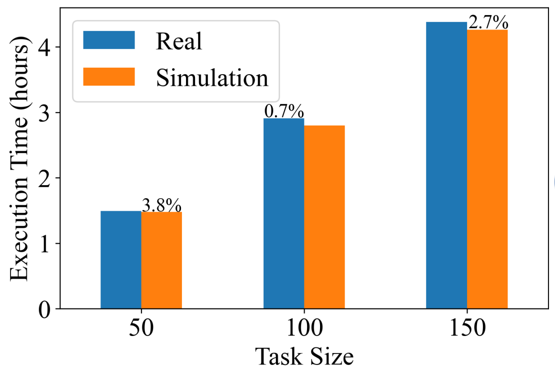
- *Pred+Adap: Our* new scheduling technique.
- *Bicer:* [Bic12] This technique dynamically allocates resources on the public Cloud and HPC by considering unprocessed tasks, data transfer latency, and available HPC nodes. *Bicer* [Bic12] developed two algorithms to complete bag-of-task applications either within a given deadline or a budget constraint. We implemented the algorithm for a given deadline. *Bicer* requests HPC nodes with runtime limit equal to the remaining of the deadline.
- *Parsl-HPC:* [al.19] We also compare it to an HPC-only solution based on Parsl's execution model. In this technique, we request a node for one hour runtime limit and assign the node, when allocated, to an incomplete stage following a topologically sorted order.

OAI Analysis: The Osteoarthritis Initiative (OAI) collected large-scale imaging data to investigate knee osteoarthritis. We build on developed analysis workflows [She19; al.22a] for the analysis of 3D magnetic resonance images (MRIs) of the knee, which include imaging data such as segmentation, thickness measurement, atlas-registration, and 3D to 2D mapping of thickness maps, all based on the knee MRIs. An input image needs to be processed by all the stages in the workflow, creating a task per stage. Figure 3.5b shows the DAG (Direct Acyclic Graph) of the workflow along with the data dependencies among its nodes, and different stages require either CPU or GPU resources for computation. The blue stages run on CPUs while green ones execute on GPUs.

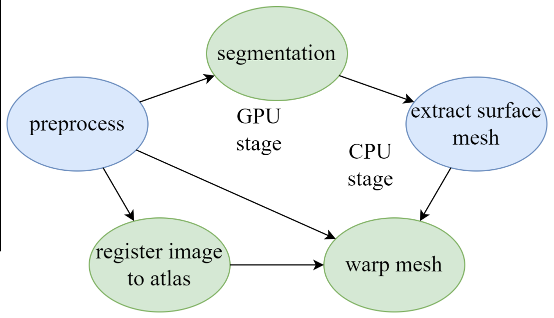
RNASEQ: RNASEQ is an RNA sequence analysis pipeline [al.20b]. We use data from the bladder cancer cells study as input [al.21a]. The workflow contains a total of 12 stages, each of which is executed on a CPU. RNASEQ is larger compared to OAI in terms of the number of stages and input/output data size.

Both workflows need significant data transfer, with some stages needing input data of size 240 MB and 14.5GB for OAI Analysis and RNASEQ, respectively. The data preloading technique and dynamic task assignment work together to minimize the impact of data transfer delays.

To simulate workflow execution, we first collected the execution traces on different platforms (see Table 3.1). On each platform, we process 200 images for OAI Analysis and 100 images for RNASEQ, and record the execution times of tasks per stage. We use this execution time distribution to resemble task execution during simulation. Further, we build multiple test cases with different combinations of the number of tasks, deadlines, and workflow execution start time (see Table 3.3). For HPC system simulation, we used job records from October 18, 2020, at 12 AM till November 18, 2020, at 11 AM (one month). Since our analysis is based on the distribution of results data, we primarily use quartiles (Q1,Q3) to describe the quantitative results unless mentioned otherwise.



(a) AWS Cloud Validation



(b) OAI Analysis Workflow

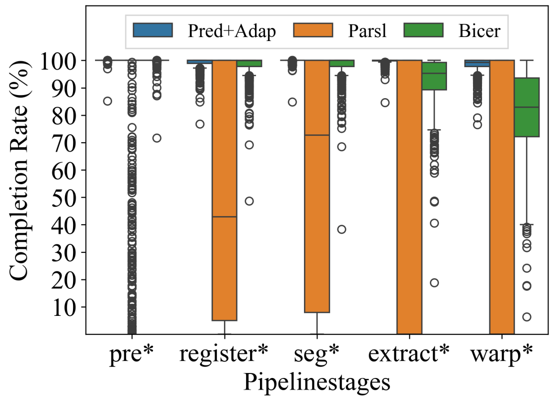
Figure 3.5 Validation and OAI Analysis Workflow

Table 3.3 Test Cases

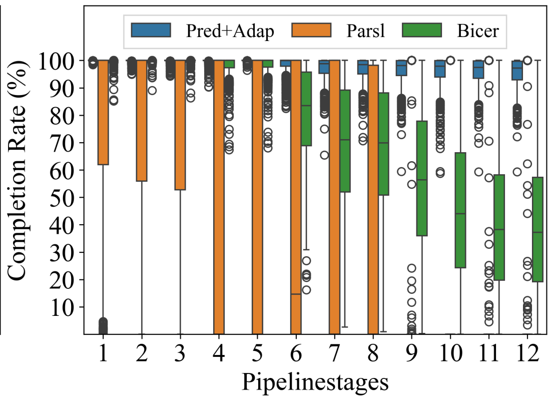
Workflow	Input Sizes	Deadlines (hour)	#testcases
OAI Analysis	12,000-96,000	6, 12, 18, 24	1,000
RANSEQ	1,000-8,000	6, 12, 18, 24	1,000

Observation 1: Our scheduling technique's ability to accurately predict RA and make adaptive resource allocation via backing up HPC allocation on Cloud achieves a high task completion rate for a given deadline.

To assess the efficiency of our scheduling technique, we compared the task completion rates of workflow runs. Figure 3.6 depicts boxplots of completion rates in percentage of total number of tasks (y-axis) for all workflow runs over the different scheduling techniques for the stages (x-axis) in the workflow. We observe that our scheduling strategy *Pred+Adap* achieves the highest completion rate of (97.79%,99.99%) and (94.44%,99.99%) for OAI and RNASEQ, respectively. Our RA prediction-based scheduling strategy not only estimates the HPC+Cloud resource requirement via RA prediction, but also handles uncertain delays in resource allocation for HPC with temporary redundant resources on Cloud, which results in a high rate of task completion for the entire workflow, with a few outliers. *Parsl-HPC*'s scheduling strategy has the lowest completion rate (0%,100%) and (0%,0%) for OAI and RNASEQ, respectively, due to a lack of HPC resource availability. Further, *Bicer*'s scheduling strategy also exhibits a lower completion rate of (72.1%,93.6%) and (36.01%, 77.77%) for OAI and RNASEQ, respectively. As *Bicer* requests longer HPC jobs and lacks the knowledge of the uncertainty of HPC allocation, it results in lower HPC throughput than required. Further, we avoid resource idleness



(a) OAI Analysis



(b) RNASEQ

Figure 3.6 Task Completion Rate

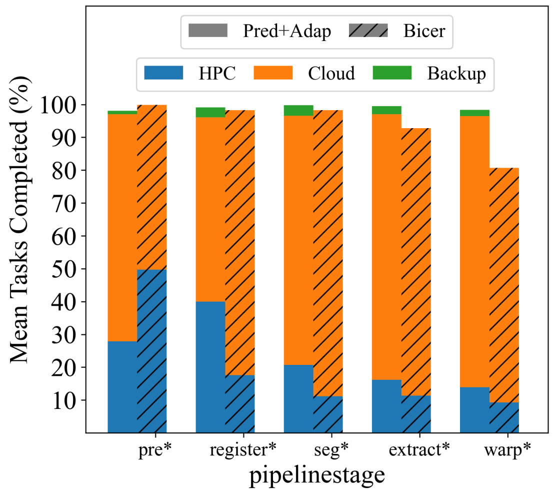
with our data preloading and resource allocation rebalancing strategies.

To better understand the importance of resource allocation strategy driven by RA prediction, we analyzed the tasks processed by HPC, Cloud, and Backup HPC allocation. Figure 3.7 shows the average percentage of total tasks (y-axis) completed by different resource groups in the applied scheduling techniques for all the stages (x-axis) of a workflow. Bicer's model can allocate the required HPC resources to execute the tasks for the initial stages of both workflows, OAI Analysis (first) and RNASEQ (first three). However, for the later stages, Bicer's model cannot obtain the necessary HPC resources, which leads to lower throughput by Cloud resources due to idleness. The incorrect assumption made by Bicer's model that the required resources are always available means it cannot allocate enough HPC resources for all the stages in the workflow. Our Pred+Adap model's ability to predict RA yields better HPC resource allocation and adapts well with backup HPC resources.

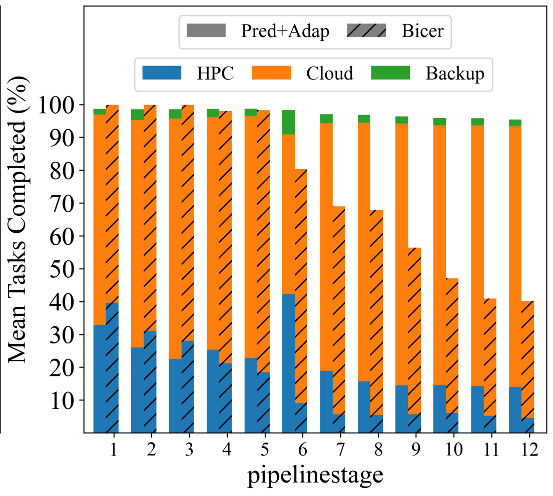
Observation 2: Estimating RA accurately and backing up HPC resource requirements via Cloud reservations minimizes the impact on the schedule of HPC production jobs.

To measure the impact of our RA prediction-based hybrid scheduling on production jobs (jobs other than our backfilled workflow), we assess the change in wait times of the production jobs that were submitted by other users on Lassen during the execution of our workflow. Figure 3.8 shows the delays in hours (y-axis) experienced before starting production jobs over different job groups (x-axis) over all 1000 workflow runs for all the scheduling strategies. A positive value means a production job was delayed because of resource usage by our workflow. Conversely, a negative value indicates an earlier start of a production job.

We observe that our scheduling technique *Pred+Adap* imposes small delays of (0, 0.12) hours in



(a) OAI Analysis



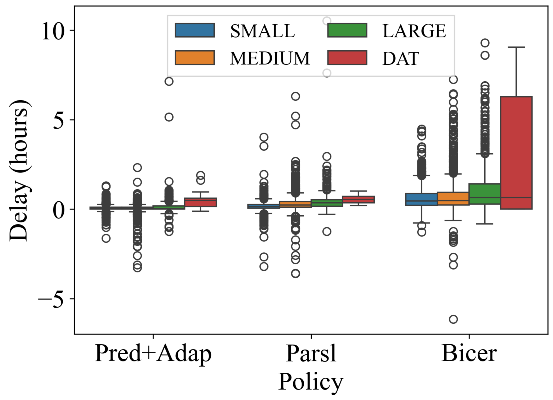
(b) RNASEQ

Figure 3.7 Percentage of tasks completed by resources

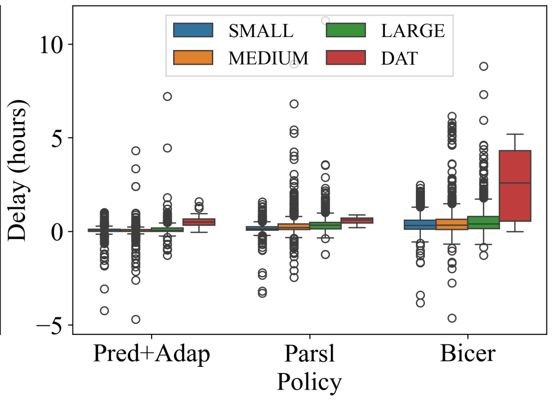
most cases on production jobs across all job groups. However, *Bicer* affects the production HPC jobs by causing a larger delay of (.23, 0.99) hours for all jobs, as it does not limit the number of HPC resources due to a lack of RA prediction. *Parsl-HPC*'s delay impact on production jobs is a little higher than *Pred+Adap* with (.08, 0.40) hours. This impact is lower than that of *Bicer* as we request an HPC resource only after the previous request is complete, resulting in shorter wait queues. In contrast, *Bicer*'s requests for resources with maximum runtime limit (up to the deadline), while we benefit from our one-hour runtime limit for both *Pred+Adap* and *Parsl-HPC*. We also observe that the delay in start times increases with the node size of production jobs for all the scheduling techniques. This is because larger jobs like DAT and LARGE are more sensitive to the FCFS + backfilling HPC scheduling strategy.

Observation 3: We provide cost predictions of workflow execution in the HPC+Cloud hybrid environment with a mean underestimation of 14.75% and 7.11% for OAI and RNASEQ, respectively.

To measure the accuracy of our cost estimator, we measured the percentage difference between predicted and actual costs relative to (divided by) actual cost for all workflow runs. A positive value indicates cost underestimation. In contrast, a negative value indicates cost overestimation. Figure 3.9 shows the distribution of cost error in percentage (y-axis) for all workflow runs. Figure 3.9a provides the cost estimation error with respect to the workflow deadline. Figure 3.9b depicts results for the underestimated cost relative to input size. We removed 45 outliers out of 2000 data points for better visibility of the data.



(a) OAI Analysis



(b) RNASEQ

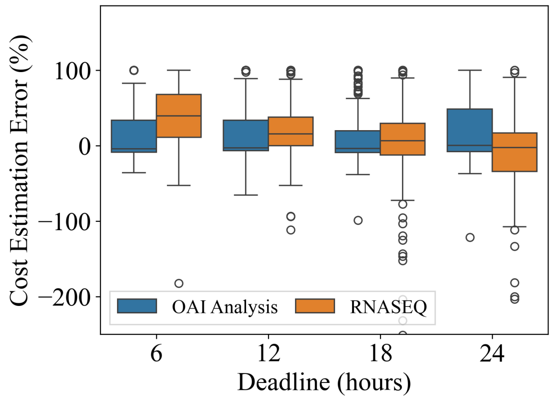
Figure 3.8 Delay of Production Jobs

Our cost predictor results in errors that most commonly lie between (-7.95%, 38.47%). However, we observe a cost estimation error of (-6.69%, 44.21%) for RNASEQ compared to OAI's (-8.22%, 30.38%). Cost underestimation primarily happens because we do not account for the backup HPC resource allocations on the Cloud. In contrast, cost overestimation is primarily due to overestimating the Cloud allocation cost that results from an incomplete DPB. For a larger workflow, its DPB construction may remain incomplete for the stages with a longer execution start. As we assume a five-minute default execution time for the missing stages (mostly with less than 5 minutes of execution times) in DPB, we may end up overestimating the cost.

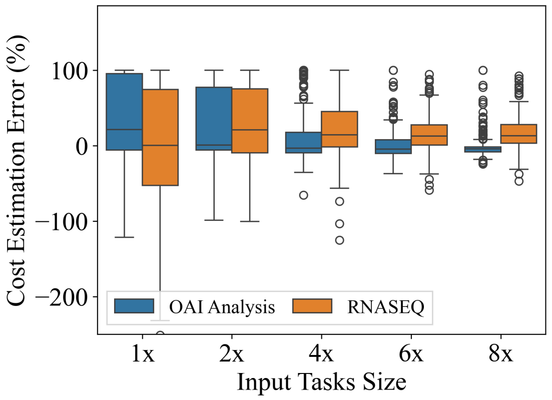
Figure 3.9a shows that the cost estimation error (y-axis) distribution for the OAI Analysis workflow remains steady as the deadlines become longer (x-axis). However, for RNASEQ, cost overestimation grows with the deadline. Larger HPC usage due to longer deadlines can also result in cost overestimation when the DPB for the HPC resources is incomplete and Cloud allocation cost is overestimated, as it happens for RNASEQ. Figure 3.9b indicates that cost estimation error (y-axis) decreases as input task size increases (x-axis). This is due to the cost of backup HPC resources that we do not include in our predicted cost. For smaller input task sizes, the predicted dollar cost is so small that it is often close to the cost of backing up HPC resources on the Cloud (i.e., approaching 100%), which can be further amplified by data movement cost during execution.

Observation 4: *Pred+Adap*'s scheduling approach in an HPC+Cloud hybrid environment yields significant cost savings.

To assess the dollar-cost benefits of our proposed scheduling technique, we compared the cost of executing a workflow run completely on the Cloud against our *Pred+Adap* but exclude *Parsl-HPC*



(a) Over Deadline



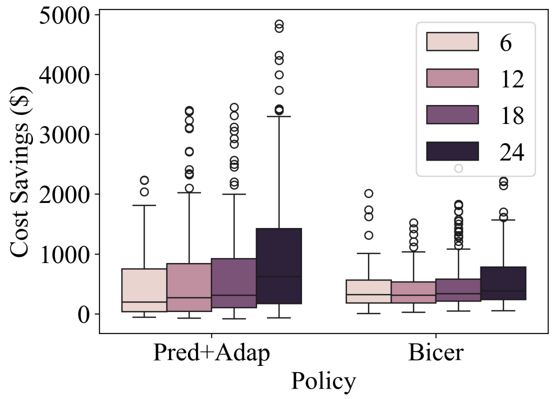
(b) Over Tasks Size

Figure 3.9 Difference between Predicted and Actual Costs

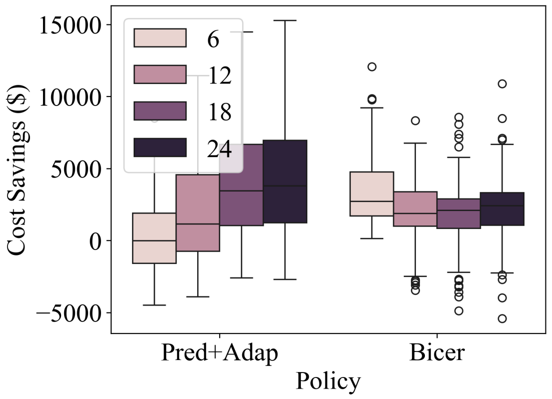
since it executes only on HPC. Figure 3.10 shows the distribution of cost savings in dollars (y-axis) for all workflow runs for different deadlines. A positive value suggests a lower cost for a scheduling technique compared to Cloud-only scheduling; conversely, a negative value implies a higher cost. As can be seen, *Pred+Adap* yields better cost savings with a mean savings of 23.96% and 28.59% for OAI and NASEQ, respectively, compared to *Bicer*'s 16.82% and 26.5% for OAI and RNASEQ, respectively. The higher savings for *Pred+Adap* culminate from better usage of HPC resources than *Bicer*. Our scheduling technique predicts RA to better understand how much Cloud resources will be required and also addresses the uncertainty in HPC resource allocation with backup resources on Cloud. Further, we observe that savings increase with longer deadlines as a larger number of unused HPC nodes (void of dollar cost) is allocated. Overall, from Observations 3 and 4, we deduce that the higher RA of HPC can yield better cost savings, but can also result in a higher cost overestimation error.

Observation 5: Accurate RA prediction leads to lower cost of backing up HPC resources via the Cloud.

We back up requested HPC resources via Cloud reservations (see Section 3.3.2) to avoid RA mispredictions, which may delay workflow execution. To measure the accuracy of our RA prediction, we calculated the reservation cost of the additional Cloud resources due to backing up delayed HPC resources. Figure 3.11 shows this backup cost (y-axis) as a percentage of the total Cloud cost. Figure 3.11a depicts this cost for different deadlines (x-axis) while Figure 3.11b plots the cost over different input task sizes (x-axis). We observe that the additional backup cost is only ≈0-4% and ≈0-14% for 75% of the workflow runs for OAI and RNASEQ, respectively. The overall average additional backup cost is ≈10.5%. This suggests that our HPC resource requests are immediately satisfied



(a) OAI Analysis



(b) RNASEQ

Figure 3.10 Cost Savings

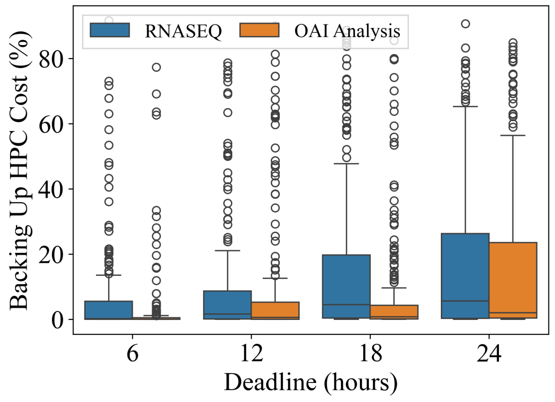
thanks to accurate RA predictions and the lower runtime limit of one hour, which contributes to a better utilization of the backfilling capacity for the HPC scheduler. Considering this accuracy, combined with the minimal impact our scavenging method has on production HPC jobs, our RAP predictor and adaptive scheduler can be deemed highly effective in executing large-scale workflows in an HPC+Cloud hybrid manner without perturbing normal HPC operations.

We further observe that larger deadlines lead to small increases in backup cost (see Figure 3.11a). This is due to increased HPC resource usage with longer execution times. The results further indicate that backup cost decreases as the input size grows (Figure 3.11b). This is due to the increment in Cloud resource requirements as task sizes become larger, leading to a lower HPC-to-Cloud resource usage ratio.

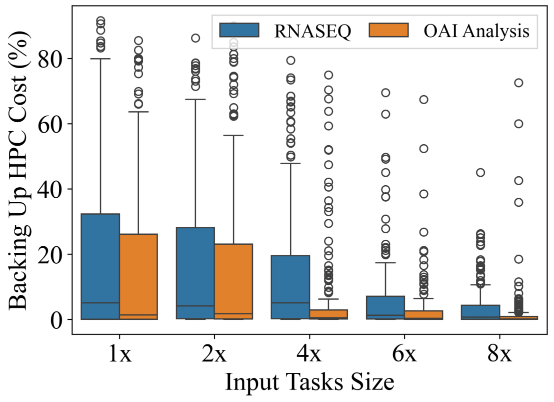
Table 3.4 Task completion rates (Q1, Q3) without Data Preloading, Rebalancing and Backup HPC

Model	OAI Analysis	RNASEQ
Full Model	(97.79%, 99.99%)	(94.44%, 99.99%)
w/o Data Preloading	(93.27%, 99.97%)	(85.17%, 99.28%)
w/o Rebalancing	(89.11%, 97.15%)	(84.11%, 97.3%)
w/o Backup	(97.18%, 99.99%)	(91.32%, 99.9%)

Observation 6: The importance of techniques such as data preloading, backing up HPC, and



(a) Backup Cost in % over Deadline Length



(b) Backup Cost in % over Tasks Sizes

Figure 3.11 Cost of Backing up HPC in the Cloud

rebalancing resource allocation increases as workflow size grows.

Although our scheduling approach is primarily based on RA prediction, it is supported by multiple techniques, including data preloading, backing up of HPC resources, and rebalancing of resource allocation. We performed an ablation study to measure the impact of these techniques on our model. Specifically, we compare the task completion rates (Q1, Q3) for our full model against models without one of these techniques. The results, as shown in Table 3.4, show that all the techniques have a significant performance impact, improving the task completion rate to meet deadlines. More importantly, the impact of these techniques grows as the workflow size (both node and data) grows, with lower completion rates of (85.17%, 99.28%), (84.11%, 97.3%) and (91.32%, 99.9%) without data preloading, rebalancing and backup HPC, respectively, for RNASEQ compared to (93.27%, 99.97%), (89.11%, 97.15%) and (97.18%, 99.99%) for the OAI Analysis workflow. Further, data preloading and resource rebalancing techniques are more impactful than the backup HPC technique for both workflows.

Observation 7: Backup resources on Cloud for HPC allocations have minimal impact on task processing.

To measure the impact of backup resources on the workflow scheduling, we analyze their average reservation time and the number of tasks processed by them. A backup resource for HPC has an average reservation time of 0.4 hours (1 hour time limit) and contributes only to 2.7% of total tasks on average for OAI Analysis. For RNASEQ, the reservation time is 0.5 hours and the processing contribution is 3.26%. This suggests that our RA prediction is effective in minimizing the impact on HPC production jobs while providing a higher rate of task completion.

Observation 8: Our dynamic approach to building performance benchmarks with the DPB is economical and avoids a bloated budget.

Instead of relying on expensive benchmarks, we built the DPB with a cumulative target of only 5% of total task size or 1 hour of run time, whichever finishes earlier, for all the platforms. The cost and duration of building the DPB are included in the final cost and the set deadline, respectively. The average cost of the DPB is only 3.24% and 3.13% for OAI and RNASEQ, respectively. The (Q1, Q3) values are (.26%, 1.48%) and (0.25%, 1.45%) for OAI and RNASEQ, respectively.

3.6 Related Work

Workflow Management Systems (WMS), such as Pegasus [al.15], Nextflow [al.17], Parsl, and Snakemake [Kös12], provide support for scaling and scheduling workflows across HPC and Cloud Environments. While they perform several scheduling optimizations, such as task clustering and data-aware scheduling, the decision to offload tasks from HPC to Cloud is left to the user's discretion. WMSs do not provide insights or heuristics for informed decision-making regarding scheduling and scaling based on factors such as RA, deadline, and cost.

Several techniques have been developed to schedule workflows in an HPC+Cloud or private-public Cloud hybrid environment to meet deadlines. Aneka [Too18] and Bicer et al. [Bic12] employ a dynamic approach where resources are scaled up/down at run time by periodically calculating available resources from the local cluster, the pending tasks, data coordination overhead and the required additional resources from Cloud. However, their assumption of on-demand HPC resources seems unrealistic as it often results in delayed allocation and insufficient computational capacity, leading to idle Cloud resources and lower completion rates. In contrast, our technique mitigates the resource availability issue through RA prediction and backing up of HPC resources on Cloud with temporary (until HPC requests are met) redundancy ensuring the necessary computational capacity is maintained on both HPC and Cloud platforms. Another difference is that both Aneka and Bicer overallocate resources to compensate for data movement overhead, whereas our technique applies data preloading to avoid such overhead, thus not allocating additional Cloud resources.

Guo et al. [Guo14] primarily performs CB on private Cloud by offloading requests to public Cloud while predicting when the private Cloud is overloaded using a workload forecaster. Further, it optimizes the cost of offloading by selecting applications with lower Cloud costs and primarily focuses on resource scaling under high workloads without a deadline constraint for workflows. Their work does not focus on complex workflows and relies on workload behavior to scale up/down public Cloud resources on demand. Gupta et al. [al.16] utilizes HPC jobs' performance metrics and resource requirements for optimal job placement on HPC and Cloud. Assuncao et al. [Ass09] evaluated the

Table 3.5 Hybrid Scheduling Comparison

Contributions	**Model**	**Scheduling Strategy**	**Predictor**	**Deadline Constraint**	**Cost Reduction**	**Cost Estimation**
Our Work	HPC+Cloud	scavenge unused HPC nodes	HPC Resource Availability	✓	✓	✓
[Bic12], [Too18]	HPC/Private+Public Cloud	offload to Cloud	N/A	✓	✗	✗
[Guo14]	Private+Public Cloud	offload to Cloud	future workload	✗	✓	✗
[Ass09]	Local Cluster+Cloud	optimized placement	N/A	✗	✓	✗
[al.16]	HPC+Cloud	optimized placement	N/A	✗	✓	✗

cost of using additional Cloud resources to supplement the lack of resources on a local cluster. They evaluated multiple job placement strategies based on various performance metrics such as job wait times, deadline violations, and the cost of Cloud. These studies primarily focus on optimizing the local cluster scheduler by determining the optimal placement (local or cloud) for all jobs on an HPC system. They do not account for workflows with complex data dependencies or data movement costs. Additionally, they lack mechanisms for estimating future resource availability and rely on users to estimate the required resources. In comparison, our work focuses on large-scale workflows to split them across HPC and Cloud while taking advantage of unused resources on HPC.

Existing studies either do not consider deadline constraints, assume that resources are available on demand, which is not true in HPC, or do not consider data dependencies between jobs. Therefore, none of them is directly applicable to solve the deterministic scheduling problem. Deterministic scheduling of large-scale workflows with deadline requirements has the following requirements: estimation of required resources, estimation of resource availability, considering data overhead in case of Cloud offloading, and estimation of cost of execution. Our work satisfies all of them. Table 3.5 compares our work with other models and illustrates the uniqueness of our approach.

3.7 Summary

We identified the challenges of scheduling large-scale workflows with deadlines on HPC systems instead of solely utilizing costly Cloud systems. A lack of knowledge about future RA and complex data dependencies in workflows limits the ability of WMS/users to make informed decisions on scheduling workflows with respect to cost and deadline. To mitigate these challenges, we developed a novel scheduling framework to orchestrate large-scale workflow execution in an HPC+Cloud hybrid environment guided by the user's choice of a schedule. To this end, we developed RAP and WSB to generate multiple schedules of workflow execution, which helps users make informed decisions utilizing factors such as cost and deadline. Further, our adaptive scheduler addresses issues such as misprediction of RA and delayed HPC resource allocation by temporarily backing up HPC resources on the Cloud. We use simulations to assess our techniques, as actual execution on Cloud resources would incur prohibitively high costs for our study. The evaluations show that our scheduling framework yields a mean 98%-99.4% rate of task completion, and with a mean 7.11% to 14.75% cost estimation error of the final cost of execution in comparison to a mean of 74.77% to 93.98% and 45.35% to 51.1% task completion rates by Cloud bursting and HPC-only solutions, respectively. Furthermore, our framework saves cost for more than 75% of diverse workflow runs. Combining minimal impacts on production jobs and minimal backup cost of HPC resources on the Cloud, our HPC+Cloud co-scheduling methodology shows good accuracy of our RAP model while keeping cost at bay and considering deadlines.

CHAPTER

4 WHESL: A DISTRIBUTED EXCEPTION MANAGEMENT FRAMEWORK FOR HPC

4.1 Introduction

Modern HPC systems rely on scale, hierarchy, and heterogeneity in both software and hardware stack for the reliable and efficient execution of workflows ranging from large-scale scientific simulation and AI/ML-guided drug discovery to large language model (LLM) training and inference [Mor18; Lua25; Nar21; DN19]. However, failures continue to be a performance bottleneck for such large-scale applications as the Mean Time Between Failures (MTBF) is in the order of hours for large systems [Kok25; Jia24; Zu24; Gup17]. In the era of exascale computing, FT support for such large-scale workflows presents multiple challenges.

First, the commonly used C/R-based solutions focus narrowly on monolithic parallel applications [Di17; Sat12; Beh20; Beh22; Mau24]. Workflows are composed of multiple jobs with different computation abstractions and dependencies. An exception in one of the jobs can have repercussions in other components of a workflow. Since traditional C/R is primarily designed for MPI-based parallel applications, it cannot address the dependencies and complexities associated with advanced workflows. Although redundancy can be used with workflows, it is not scalable and efficient. Modern

HPC systems need FT solutions that handle system complexities and hierarchical dependencies in workflows.

Second, no WMS currently supports advanced exception management. Any WMS plays a central role in effectively automating and managing large-scale executions via coordination of complex tasks such as workflow construction, efficient resource management, reproducible execution, and monitoring. Such a WMS generally supports FT via automatic job re-execution, workflow check-pointing or exception handlers in workflow scripts. However, scheduling and execution of workflows involve multiple independent software entities, e.g., the system scheduler, the WMS itself, individual jobs, the I/O subsystem, all within a complex (and non-uniform) ecosystem of HPC computing. Multiple such entities can detect failures in isomorphic form and may take individual actions to recover. These entities lack knowledge of each other. They do not coordinate to reach a consensus for efficient recovery. Hence, the resultant state of execution may become either redundant or non-recoverable. Furthermore, FT solutions provided by a WMS, apart from C/R, are not efficient. This includes widely used automatic job re-execution, where individual jobs are retried up to a predefined number of times upon failure. For each restart, a WMS resubmits the failed jobs to the system scheduler, which adds additional overhead such as requeuing and reallocation of resources. Also, exception handling via programming, e.g., *try...except* blocks in Python, requires FT expertise by application/workflow developers, which is uncommon. Finally, FT is not portable across the plethora of WMS frameworks available to users. Each WMS provides a different interface for workflow construction and FT support. Managing and porting workflows from one to another WMS or HPC system adds even more overhead. Overall, the HPC ecosystem lacks a common and standard FT solution compatible with multiple WMS frameworks.

In short, today's WMS frameworks lack in FT support for *scalability*, *coordination*, *usability*, and *portability* on modern HPC systems. We address these gaps via WHESL, a standalone exception management/runtime system that integrates with WMS frameworks to provide advanced FT solutions for efficient recovery. WHESL's holistic system and workflow view helps manage exceptions efficiently, specifically to detect, propagate, and perform recovery by letting users define exceptions in a programmable fashion. To this end, we isolate exceptions from being detected by different subsystems to eliminate redundant actions during recovery when taken independently. To achieve that, we *fluxify* (schedule tasks via Flux [al.18a] on individual allocations) the jobs submitted to the system scheduler by the WMS. This allows us to detect and isolate exceptions locally. After isolation, we propagate the exceptions adhering to the workflow hierarchy and coordinate with multiple subsystems to ensure efficient and orchestrated recovery. Without WHESL, when an exception happens, a job fails and may be restarted redundantly by the WMS and the system scheduler with new allocations. In contrast, with WHESL, the failed jobs are restarted locally whenever possible to eliminate the inefficiency due to new resource allocations and the lack of coordination between the WMS and the system scheduler.

At the core of WHESL lie three modules: ESL, EMIL, and ERM. The ESL module is comprised of a user interface (UI) supported by a domain-specific language (DSL) that allows users to define exceptions in terms of detection and recovery. The EMIL module isolates and propagates exceptions and coordinates recovery among various workflow and system components to provide efficient recovery, as it avoids redundant recoveries by different workflow and system components. The ERM library is composed of multiple recovery methods with an HPC resource-centric view that ensures ease of access and promotes simplicity for users. The ESL and ERM components are implemented as standalone software packages. In contrast, the EMIL component is built on top of Flux, a hierarchical and scalable RJMS to take advantage of its scalable and nested scheduling capability.

In summary, we make the following contributions:

• We identify and discuss the gaps present in the current Fault Tolerance systems for workflows that need to be addressed.

• We propose, design, and develop a novel Exception Management System, WHESL, that can efficiently recover from failures in a coordinated manner, leveraging a scalable and nested scheduler such as Flux while relying on the workflow and system view.

• We demonstrate two important use cases of managing node failures and disk quota expiry exceptions via WHESL.

• Experimental results indicate that WHESL provides low recovery time from failures in complex workflows. It performs well in both failure scaling and task scaling with a ≈1.6%-4.3% and ≈-2.09%-2.53% mean difference, respectively, against failure-free runs. Further, WHESL causes minimal interference with application execution.

4.2 Background

Let us we briefly discuss the the relevant entities on HPC systems and their approaches to support FT for workflows.

4.2.1 Resource and Job Management System

HPC systems are generally managed by a single instance of a central RJMS such as SLURM [Yoo03], PBS [Nit04], Cobalt [Lab25], or Moab/TORQUE [Com25]. The key responsibilities of the RJMS include resource allocation, job scheduling, execution, monitoring, failure handling, and recovery. A RJMS usually supports FT for node failure or non-responsiveness via heartbeat-based monitoring to

Table 4.1 Fault Tolerance support in Workflow Management Systems

Workflow Management System	**Job C/R**	**Workflow C/R**	**Task Retry**	**Custom Exception Handler**
Snakemake [**Moelder2021**], Cylc [Oli19], RADICAL-EnTK [Bal16], Cromwel [dev25a], RADICAL-AsyncFlow [dev25b], SmartSim [Par22], Fireworks [Jai15], Makeflow [Alb12], AiiDA [Hub20], Toil [Viv17], Covalent [Cun23], libEnsemble [Hud22], Pegasus [al.15]	✗	✓	✓	✓
Dask [Roc15], Globus-compute [All12], Swift/T [Wil11], Parsl [al.19], TaskVine [SD23]	✗	✗	✓	✓
Nextflow [al.17], Maestro [DN25], PyCOMPSs/COMPSs [Tej17], Galaxy [Com24]	✗	✓	✓	✗
pyiron [Jan19]	✗	✓	✗	✓

detect and isolate individual faulty nodes and then requeue/reschedule the canceled/failed job on a new set of healthy nodes. The requeuing policy depends on job submission parameters and the system site policy. However, this mechanism does not consider the hierarchy or dependency within workflows, and it ignores how node isolation and job requeue policy may impact the workflow's other pending and running jobs with data dependencies. These shortcomings can result in catastrophic failure for workflows during uninformed recovery attempts.

4.2.2 Workflow Management System

A WMS enables users to automate and orchestrate the execution of multiple tasks that are interconnected within a workflow. The WMS is also responsible for efficiently managing the execution of workflows on heterogeneous HPC systems. To achieve that, a WMS needs to support multiple FT techniques to provide resiliency for the workflows. Such methods include automatic job restart upon failures, checkpointing at the workflow level, and error handling via exception handlers in workflow scripts.

We surveyed the FT support provided by publicly available WMS frameworks. Due to the high number of available WMS solutions, we narrow our focus to those listed on the WCI (Workflow Community Initiative) website [Ini25]. WCI is a collaborative hub for driving workflow technologies and methodologies [FDS21; FDS23; FDS24] to unite the workflow ecosystem. WCI lists 37 WMS packages, out of which 25 provide WCI-metadata that describes the capabilities of a workflow (structure, execution, data, and provenance), which indicates how workflows ensure the Findable, Accessible, Interoperable, and Reusable (FAIR) properties on computational platforms. Furthermore, we found that 23 of these WMS solutions provide some form of FT (see Table 4.1). Most of the listed frameworks support custom exception handling via either programming constructs such as *try..catch* blocks or domain-specific event hooks, allowing users to steer workflow execution upon failures. However, managing such execution requires both workflow and FT expertise. Automatic task retry is also a common feature. Automatic job restart enables individual tasks in a workflow to be re-executed upon failure up to a given number of reties specified by the user. However, the resources allocated for the failed task by the system scheduler are released, and the HPC job for the task is requeued for new resource allocation and scheduling. Furthermore, applying automatic job restart to all types of failures is inefficient. Workflow checkpointing allows a WMS to save the progress of the whole workflow so that it does not re-execute the already completed part of the workflow upon complete workflow restart from failure. However, restarting the entire workflow adds more overhead due to re-planning of execution, resubmission of jobs that had been running at the time of failure. Restarting either the entire workflow or a failed job has similar overheads at different scales. Further, each WMS requires users to learn and use different FT interfaces to add such support.

Overall, popular WMS frameworks with their existing FT techniques do not focus on the key challenges presented by scalability, coordination, usability, and portability, all of which are essential to the ecosystem of HPC for efficiency and resiliency. In contrast, our solution, WHESL, addresses these challenges as described next.

4.3 System Design

Let us discuss the overall architecture and technique of WHESL along with its core components in detail.

4.3.1 Execution Model

Workflows can be run as a pilot job by reserving the required amount of resources before execution begins. In such a scenario, the WMS has dedicated access to on-demand resources for executing individual tasks and can perform optimizations for better resource utilization. However, resource underutilization is a common side effect of pilot jobs. For a pilot job, the WMS relies on locally nested RJMS such as Flux to request resources within the already allocated resources from the top-level RJMS, e.g., SLURM. To avoid resource underutilization, the WMS dynamically creates workflow tasks and requests resources as needed. Our solution applies to both scenarios. However, we establish the dynamic resource allocation model for workflow task execution to show the performance benefits of our solution.

Further, we assume that workflows are a mix of heterogeneous workloads ranging from single-node jobs to MPI-based parallel applications. Workflows in general can be represented as a Directed Acyclic Graph (DAG) displaying a unidirectional data/control flow due to dependencies. However, we also consider coupled workflows where individual applications are running simultaneously, producing data and control for each other at run time.

4.3.2 Failure Model

In our model, failures are detectable and manifest at the application level. A failure can be detected by multiple entities, such as the WMS, RJMS, and an application's runtime system. We also allow failures in a task of a workflow to impact the other dependent tasks, resulting in the propagation of failures. But failures are confined to workflow task execution, only, and are not observed in or propagated to the application runtime system, system scheduler, and WMS. Further, a failure may or may not result in an application crash. After detection, we refer to a failure as an exception, and our work primarily focuses on managing such exceptions efficiently.

4.3.3 Design

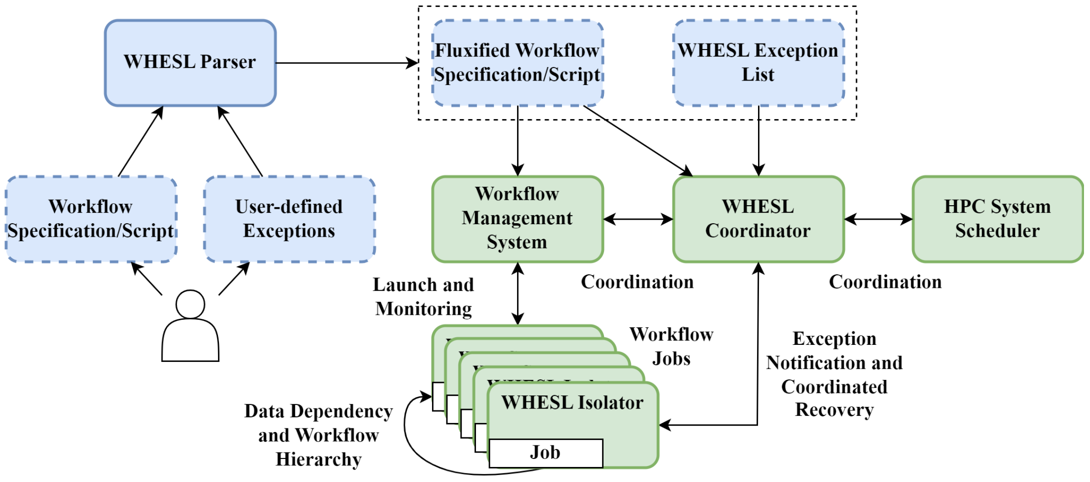

WHESL Parser
Fluxified Workflow Specification/Script
WHESL Exception List
Workflow Specification/Script
User-defined Exceptions
Workflow Management System
WHESL Coordinator
HPC System Scheduler
Launch and Monitoring
Coordination
Coordination
Workflow Jobs
Exception Notification and Coordinated Recovery
Data Dependency and Workflow Hierarchy
WHESL Isolator
Job

Figure 4.1 Overall WHESL Architecture

Figure 4.1 shows the overall architecture of WHESL, where the blue boxes represent static components executed before workflow execution, and the green boxes represent dynamic components executed at runtime. The dashed boxes represent inputs to or outputs of a subsystem, whereas the solid boxes represent a subsystem or an application. WHESL has primarily three subsystems: the *WHESL Parser* built on top of the ESL, *WHESL Isolator*, and the *WHESL Coordinator*; both built as part of the EMIL. Both the *WHESL Isolator* and the *WHESL Coordinator* use the ERM library for efficient recovery from exceptions.

WHESL Parser: Users provide two input files to the parser to execute the workflow with WHESL-enabled FT support. The workflow specification/script defines the tasks and their resource requirements, input files, and the dependencies among the tasks. The exceptions list provides details about exceptions and their detection and recovery steps defined using the ESL. The parser primarily has two responsibilities. It first validates the exception list, which must follow the grammar defined by the ESL. The ESL is a domain-specific language developed using Antlr4 [Par14]. It provides users with syntax to define exceptions, their detection, and recovery mechanisms in a declarative manner. Users also define additional details, such as different entities and resources in the HPC ecosystem.

Listing 4.1 shows the important rules for representing exceptions, resources, and entities while omitting the details for brevity. The ESL primarily defines three major constructs: entities, exceptions, and resources. Entities are the stakeholders involved in workflow execution, namely, the WMS, the system scheduler, and the running job. Entities (lines 8-9) are the actors that perform recovery actions. Resources (lines 10-14) are the objects in an HPC ecosystem on which different operations, such as add, remove, or replace, are performed to recover from exceptions. With the exception-related rules in the ESL, users define exceptions (line 16), their detection process (line 17), and a set of operations on different resources by multiple entities (line 18) to recover from them. The ESL essentially provides mechanisms to describe exception management in a holistic manner.

After validation, the *WHESL Parser* produces the exception details in JSON format that is accepted by the *WHESL Isolator* and *WHESL Coordinator* to manage them at runtime. The parser further modifies the workflow specification file or script to embed isolator instantiation code and identification details for the workflow tasks. We refer to this process as *fluxification*, which serves as a means to create a Flux instance first, followed by a *WHESL Isolator* instance for a newly allocated job. The parser also preserves all other aspects in a workflow, such as input data, dependencies, and resource allocation configuration for the tasks. Users submit or run the *fluxified* file to the WMS to start execution of the workflows.

Listing 4.1 Rules subset of Exception Specification Language

```
grammar whesl2;

expression: declare_entity declare_exception_type \
                exception_statement  EOF ;
exception_statement: declare_resource
                   | declare_exception
                   | exception_detection
                   | exception_recovery ;
declare_entity: 'DECLARE ENTITY' entity_list ;
entity_name: 'SCHEDULER' | 'WMS' | 'JOB' | 'DEPENDEND_JOB' ;
declare_resource: 'DECLARE RESOURCE' resource_name \
                      '[' resource_op_handle_list ']' ;
resource_name: 'NODE' | 'JOB' | 'FILESYSTEM' ;
resource_op_handle_list: resource_op_handle (',' resource_op_handle)  ;
resource_op_handle: resource_op custom_handler? ;
resource_op: 'ADD' | 'REMOVE' | 'REPLACE';
declare_exception_type: 'DECLARE EXCEPTION_TYPE' exception_type_list ;
declare_exception: 'DECLARE EXCEPTION' exception_name exception_type \
                      'JOB' '[' jobs_list ']' ;
exception_detection:  exception_detection_log_match
                      | exception_detection_log_recurring_match
                      | exception_detection_log_multiple_match
                      | exception_detection_log_hang
                      | exception_detection_exception;
exception_recovery:  'EXCEPTION RECOVERY STEP' exception_name \
                      entity_name resource_name '.' resource_op args?;
exception_type: 'JOB_LEVEL' | 'WORKFLOW_LEVEL' ;
```

WHESL Isolator: An isolator instance primarily encapsulates one workflow task at runtime and performs the following operations. It manages the job's life cycle in terms of launching the workflow task as a Flux job, subscribing to the Flux job's logs to parse for exceptions, performing relevant exception handling per the exception specification provided by the user, while planning execution together with the *WHESL Coordinator*. These operations are performed per workflow task by one isolator, managing exceptions locally whenever possible. Because the workflow tasks are *fluxified*, when the Flux jobs fail, it does not translate to an HPC or workflow job failure, thus isolating exceptions to the local allocation. When an exception is detected by an isolator, it begins orchestrating with the coordinator to recover by relaunching the failed job with a new configuration locally. If an exception cannot be managed locally, then the isolator exits, leading to an HPC or workflow job failure. Such failures are managed by transforming the local exception to a higher-level exception and handling it via the coordinator. During initialization, the isolator registers its workflow task details with the coordinator to build the workflow hierarchy in terms of task dependencies.

WHESL Coordinator: Our framework operates with one instance of the coordinator functioning as the central management system for exceptions, which can operate on multiple workflow instances. It primarily coordinates the tasks within a workflow to maintain and update its hierarchy in the presence of exceptions. To do that, it identifies and stores the workflow hierarchy via both the workflow specification/script and the runtime workflow task details received from the isolator at registration time. Using the hierarchy and past exception details, it dynamically updates the workflow tasks to avoid future failures. Further, it coordinates with other subsystems such as the WMS and the system scheduler as and when required for exception recovery, following the recipe defined by the user.

Both the coordinator and isolator are built on top of Flux to leverage its ability for nested/hierarchical scheduling within itself and other RJMS frameworks. We leverage Flux's rich communication APIs, event and log management features to perform multiple tasks, such as to communicate between *WHESL Isolators* and *WHESL Coordinator*, manage workflow tasks as Flux jobs locally, and subscribe to logs from the jobs to parse for exceptions. Next, we discuss the details of detecting and isolating exceptions using WHESL.

4.3.4 Exception Detection and Isolation

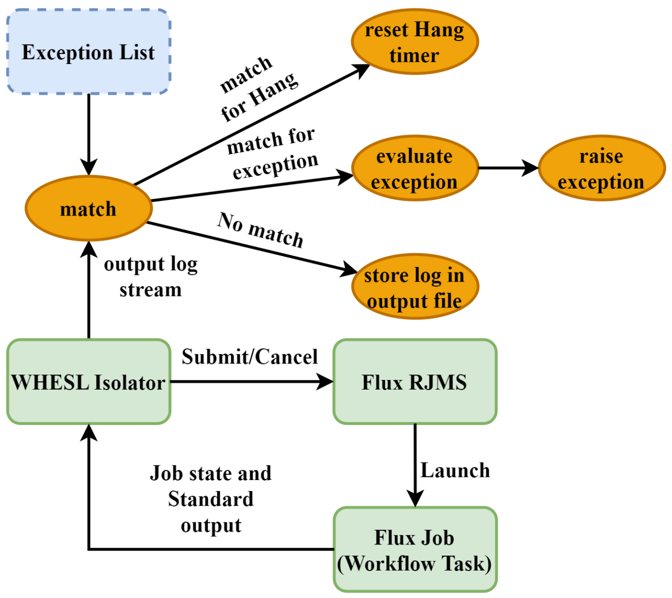


Figure 4.2 Exception Detection in WHESL

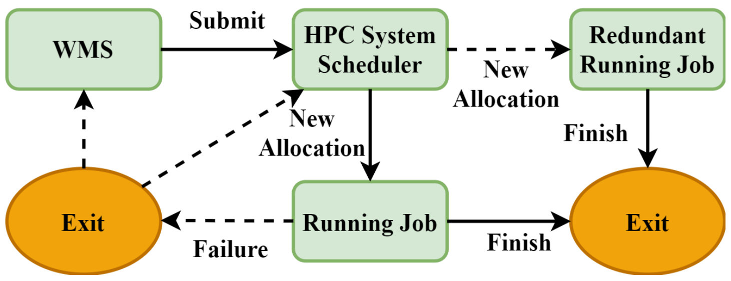


Figure 4.3 Inefficient Recovery without WHESL

Figure 4.2 depicts the overall design of the exception detection technique. The oval-shaped

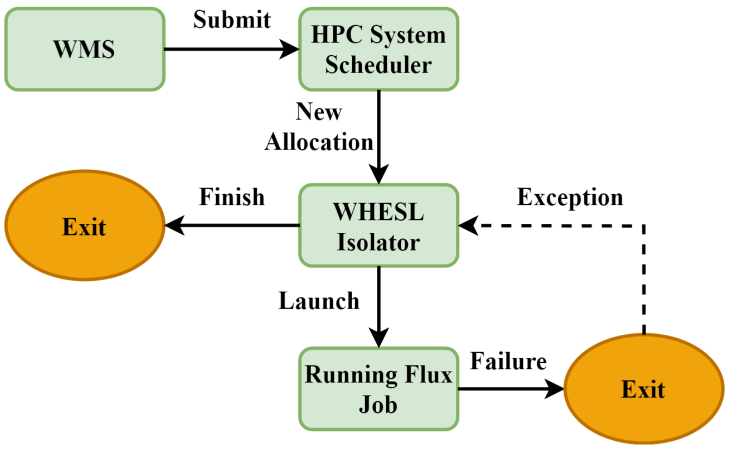


Figure 4.4 Exception Isolation with *WHESL Isolator*

orange objects represent different operations performed by the isolator, and the green colored square boxes represent running subsystems or applications. The blue-dashed boxes represent input to a subsystem. WHESL primarily relies on application logs to detect exceptions. However, other advanced failure detection techniques [Das18b; Das18a] and tools can be easily integrated with via RPC calls to raise exceptions. The isolators detect exceptions at the workflow task level by parsing the output logs from individual jobs. An isolator leverages Flux's built-in features to subscribe to various events related to Flux jobs, including job state changes and standard I/O logs. The isolator subscribes to these events, manages the assigned workflow task's life cycle as a Flux job, and parses its output logs to detect exceptions. A task's standard output is usually directed to a log file. By subscribing to the job's events, the isolators intercept logs and perform a match with the patterns provided by users to detect exception conditions. Upon a match, the relevant exception is raised. Otherwise, the Isolator stores the output to the log file.

The *WHESL Parser* enables users to match logs with different patterns for exceptions (line 17), as shown in Listing 4.1. For example, a pattern can be a single log match or multiple log matches, potentially over a recurring number of retries, which are tracked. These types of matches are needed when an exception manifests in a job's output logs. WHESL also provides mechanisms to detect a hang exception by leveraging the timers provided by Flux. The *WHESL Isolator* raises a hang exception when a particular log pattern does not appear within a given time. Finally, WHESL allows users to raise a new exception that is based on the occurrence of another exception. For example, users can raise a job failure exception for an MPI task if MPI-related failures happen a certain number of times. Both isolator and coordinator store past exceptions and evaluate an exception to determine whether it transforms to another before raising it to identify the final exception to handle. Furthermore, users can define different recovery methods for both MPI and job failures resulting from different MPI failures. WHESL also requires exceptions to be tied to specific tasks in the workflow per ESL rules (line 16).

Failures can be detected by multiple entities in the HPC ecosystem, such as the WMS, the HPC system scheduler, and the application runtime system. Upon detection, they perform recovery independently, which can be either redundant or cause conflicts. Consider the case where a node becomes non-responsive during an MPI-based workflow task, and the submitted HPC job fails. In such a situation, the WMS observes that the HPC workflow task has failed and will reschedule it via the HPC system scheduler per the workflow configuration. Simultaneously, the HPC system scheduler via the *slurmd* daemon will detect a node failure and may reschedule the job on a new set of healthy nodes. Such independent actions are redundant and may result in future catastrophic failures for reasons such as data overriding or lack of resource availability.

To avoid such problems, we *fluxify* each workflow task that is scheduled via the system scheduler. With *fluxification*, we start a Flux instance as the SLURM/HPC job. The Flux instance spawns a non-interactive Bash shell, where a *WHESL Isolator* is executed within it. The isolator receives the actual workflow task command, the coordinator's communication address, and other details related to role and workflow identification. During initialization, isolator and coordinator exchange information to establish a communication channel and the workflow hierarchy. With *fluxification*, we achieve exception isolation where job failures are shielded within the allocation by not emitting the failure signals to other subsystems (e.g., WMS and system scheduler). Further, the *fluxified* workflow tasks are reconfigured in such a way that the system scheduler, even though they identify the node failures within the allocation, do not resubmit the job automatically. Figure 4.3 and Figure 4.4 show two contrasting scenarios, without and with WHESL, respectively, depicting how exceptions are isolated. The squared blocks represent a job or a subsystem. Oval-shaped blocks represent actions. The dotted directed lines represent control flow during failures, whereas the solid lines represent control flow during normal operation. In Figure 4.3, upon node and job failures, both the WMS and the system scheduler get notified, resulting in redundant job resubmissions due to a lack of coordination. In Figure 4.4, due to the *WHESL Isolator*, the workflow tasks are run as Flux jobs and managed via local Flux schedulers. Any job failures are constrained by their isolators to avoid any redundancy. Next, we discuss the details of the coordinated recovery performed by WHESL.

4.3.5 Coordination and Recovery

After an exception is detected by an isolator, it initiates recovery by orchestration with the coordinator. WHESL manages exceptions that are categorized as *job-level* or *workflow-level* (line 19 of Listing 4.1). For *job-level* exceptions, only the isolator performs the recovery actions. In contrast, both isolator and coordinator share the recovery actions for *workflow-level* exceptions.

Since the recovery operations of *job-level* exceptions are limited to local allocation, the corresponding isolator performs all required actions and notifies the coordinator about the local job reconfiguration and its restart at the end. The coordinator reconfigures future jobs dependent on the failed job and sends a restart notification to running dependent jobs along with the reconfiguration details. However, for *workflow-level* exceptions, additional operations are required by the coordinator via orchestration. Figure 4.5 depicts the sequence diagram for the coordination and recovery steps performed by different entities in WHESL.

The following are the sequence of steps for coordinated recovery:

- *Step 1:* Once the detected exception is determined to be *workflow-level* type, the isolator notifies the coordinator to execute its own set of recovery actions and waits for its response. After the coordinator executes its set of actions, it notifies the Isolator to execute its set of recovery actions.

- *Step 2:* The failed workflow task is restarted with a new configuration. The new configuration is determined via the recovery steps defined in the exception specification file. In this work, we primarily focus on resource allocation and I/O configurations. Depending on the exception and the new configuration, the failed workflow task has to be restarted either on the already allocated previous set of resources, i.e., by restarting the Flux job, or on a newly allocated set of resources for which an HPC job restart is needed.

- *step 3:* Next, the coordinator updates the configuration of dependent jobs so that they become compatible with the restarted job. However, the coordinator does not update the configuration immediately after exception recovery. Instead, it stores the revised configuration and updates it when the dependent jobs' Isolators are initialized. For already running jobs, updates are performed immediately.

A WMS primarily provides mechanisms to define exception recovery in a declarative format that are specific to an action, i.e., restart of the failed job. For example, users can declare the number of times the WMS restarts a job before stopping the workflow execution. Or, if a job failed a certain number of times, then WMS migrates the job to a different HPC system, per user specification.

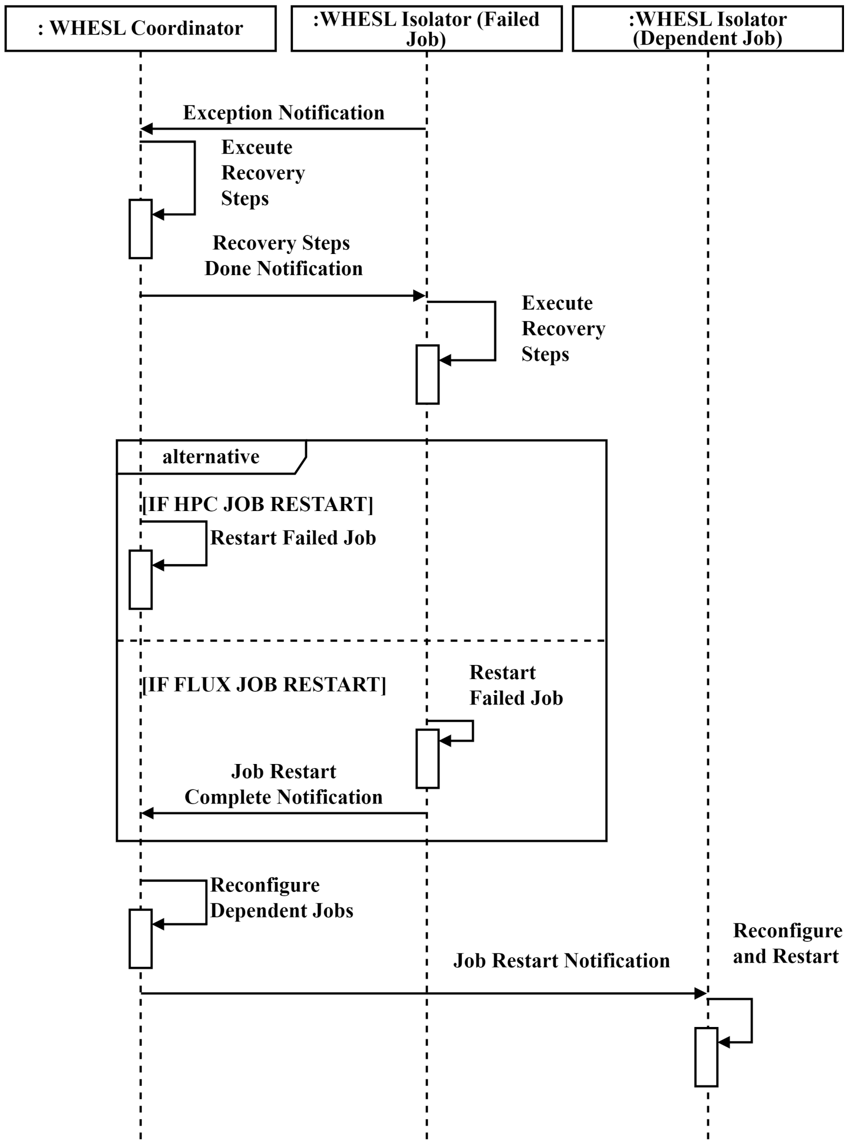


Figure 4.5 Sequence Diagram for Recovery Coordination in WHESL for *workflow-level* exceptions

Further, language-based exception handlers lack a generic model that makes it easy for users to define recovery steps for any exception. To address these challenges, WHESL developed a resource-centric action model to define recovery steps for any exception. Users can define recovery steps in a declarative format that is extensible and allows customization.

WHESL's recovery model is defined in two sections of the ESL in Listing 4.1: resource management and exception recovery. The resource management section (lines 10-14) lets the user define different types of resources and the operations that are performed on them. A resource is an object in the HPC ecosystem that is created or used during workflow execution. For example, the parallel file system, an HPC job, a node, etc. Exceptions are found in workflow tasks primarily due to failures in these resources and can be recovered by managing these resources during workflow execution in a coordinated manner. Managing resources includes operations such as the addition of more resources, removing the faulty ones, and replacing the faulty resources with healthy ones. For example, in the case of a node failure, the failed node can be either removed or replaced with a healthy node in the allocation. Similarly, if the primary filesystem is non-responsive, it can be replaced with a new one. Furthermore, if the job throughput does not meet the expectation, then additional nodes can be added to improve processing speed. To support such management, we currently support three operations: add, remove, and replace. Users can use such resource-operation constructs to build recovery mechanisms for each exception (line 18) per Listing 4.1. WHESL currently has support for resources such as *job*, *filesystem*, and *node* with add, remove, and replacement operations. However, WHESL's recovery model is extensible as users can add more resources and their supported operations.

4.4 Evaluation

We demonstrate WHESL's efficient recovery management for three types of exceptions: Disk Quota Exceeded (DQE), Application Crash due to node failure, and Application Hang due to communication failure. We implemented WHESL on top of Flux version 0.60.0. We conducted the experiments on an 80-node institutional cluster featuring heterogeneous mixes of CPUs and GPUs. For application crash and hang exceptions, we use only CPU (AMD Epyc Rome) platforms with the LULESH application. For the DQE exception, we use both CPU and GPU (NVIDIA RTX 4060 Ti) platforms as required by the OAI workflow described next. The HPC cluster also supports two different filesystems: Maestro WMS manages the execution of both LULESH and OAI on the cluster.

4.4.1 Managing Disk Quota Exceeded Exception

Workflow: The Osteoarthritis Initiative (OAI) provides a plethora of publicly available imaging data of knees to investigate the causes of osteoarthritis. The OAI Analysis workflow is built on top of analysis workflows focusing on 3D magnetic resonance images (MRIs) of the knee [She19; al.22a]. The workflow comprises analysis techniques such as segmentation, thickness measurement, atlas registration, and 3D-to-2D mapping of cartilage thickness maps for knee MRIs. It is built as a bag-of-tasks structure consisting of various applications, each identified as a stage in the workflow's Directed Acyclic Graph (DAG) (see Figure 4.6). Green stages run on GPUs, while the blue ones utilize CPUs. Workflow edges represent the data flow. As a bag-of-tasks workflow, each scanned image needs to be processed by all the stages in the DAG. The OAI Analysis workflow runs on large-scale HPC systems, generating a significant amount of data. The OAI data repository contains ≈27 million images combining MRI and X-ray scans. Running analysis for 10,000 images will produce ≈3.5 terabytes of output data. For one image, Table 4.2 shows the total number of files and their cumulative size in megabytes generated per stage of the workflow.

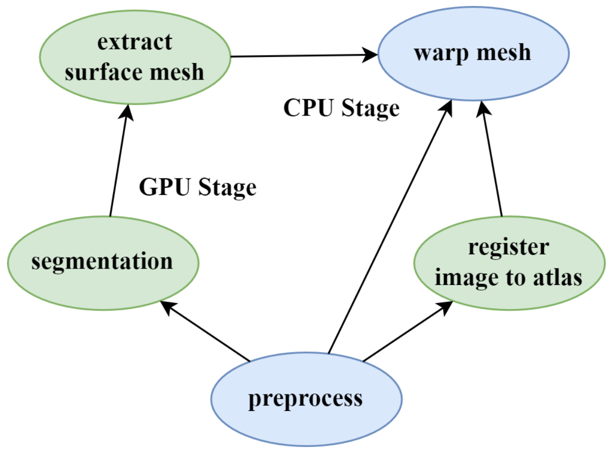


Figure 4.6 OAI Analysis Workflow

Exception: To meet this I/O load, HPC systems usually adopt tier-based storage systems, where top-tier filesystems provide lower latency and higher bandwidth, but are smaller in terms of capacity. Further, users are provided a personal disk quota limit on all the filesystems to allow fair use of resources. Running a large-scale OAI Analysis can create a large amount of data, resulting in an exception due to exceeding a user's quota. In such scenarios, workflows should be able to trans-

Table 4.2 OAI Analysis workflow's output per image

Pipeline Stage	Output Size (MB)	# Output Files
preprocess	28.20	1
segmentation	76.80	2
extract_surface_mesh	5.99	1
register_image_to_atlas	234.75	1
warp_mesh	2.09	6

parently switch their output location to another filesystem, possibly a slower, but larger-capacity one. Furthermore, such changes to input/output paths must be updated for the relevant dependent stages in a workflow, which the existing WMSes do not support. For example, with the Snakemake WMS, users can set up rules for job re-execution with a secondary filesystem on disk quota failures and even move files from the primary to the secondary filesystem. However, this update is not propagated throughout the jobs across the workflow, which will lead to future job failures. Similar behavior can be achieved with Nextflow, but users need to manually restart the workflow execution. Pegasus can automatically switch to a secondary filesystem, but cannot migrate already produced files and requires full re-execution to produce the complete output from the failed job on the secondary filesystem. In addition, none of the WMSes support such failure recovery for workflows under a dynamic paradigm where both the producer and consumer applications are running simultaneously exchanging data and control.

Listing 4.2 Exception Specification for Disk Quota Exceeded

```
DECLARE ENTITY WMS, JOB
DECLARE EXCEPTION_TYPE WORKFLOW_LEVEL
DECLARE RESOURCE FILESYSTEM[REPLACE]

DECLARE EXCEPTION DISKQUOTAEXPIRED WORKFLOW_LEVEL JOB [run−preprocess]
EXCEPTION DETECTION DISKQUOTAEXPIRED LOG "Disk Quota Expired" stdout
EXCEPTION RECOVERY STEP DISKQUOTAEXPIRED JOB FILESYSTEM.REPLACE \
        {"filesystem":"<filesystem root>", "defaultpath": \
        "default path for workflow execution"}
EXCEPTION RECOVERY STEP DISKQUOTAEXPIRED WMS FILESYSTEM.REPLACE \
        {"filesystem":"filesystem root", "defaultpath": \
        "default path for workflow system output"}
```

Recovery: The desired recovery from a DQE exception is to move data or create soft links from the partial output of the failed job to the secondary filesystem, and to ensure that all the dependent jobs (both running and pending) of the failed job are updated to read outputs of the failed jobs from the secondary filesystem. To add such recovery support to their workflows, users need to provide an exception specification file (see Listing 4.2). As per specification (lines 1-3), there are two entities in the ecosystem, the WMS and jobs, and only the filesystem resource needs to be managed upon failure. The DQE exception is categorized as a workflow-level exception (line 5) as output staging areas for both jobs, and the WMS may need to be moved to a secondary filesystem. Here, the exception is to be detected and recovered only for the *run-preprocess* job instances. WMSes such as Maestro and Parsl can parameterize tasks to create multiple instances of them. WHESL will enable exception management for all the job instances of the task. Further, line 6 shows the exact log that needs to be matched against to detect this exception. For recovery, WHESL needs to perform two updates. First, the *WHESL Coordinator* needs to make sure that the staging area for the WMS is moved to the new filesystem (line 8). Second, the *WHESL Isolator* of the failed job needs to move its output from the older staging area to a new location (line 7). As per the ESL, exception recovery statements accept user-defined arguments that are passed as key-value pairs to the resource management operations (line 18, Listing 4.1). In case of an exception due to DQE, the root of the new filesystem and the path of the current filesystem are passed as arguments to the handler governing a filesystem resource's replacement. In our implementation, the replacement operation automatically creates soft links from the already produced output files within the new filesystem before it restarts just the job (but not the entire workflow) with a new configuration that includes the new filesystem path as the output staging area. Further, the *Coordinator* updates the running dependent jobs' configurations immediately by coordinating with their *Isolator* instances, including updates of the future dependent jobs' configuration as and when they are started.

Analysis: To evaluate the impact of integrating WHESL with workflows, we ran the OAI Analysis workflow with and without WHESL support. In this experiment, the OAI Analysis workflow comprises five tasks (see DAG in Figure 4.6). Each workflow task was further partitioned into two instances, each processing 5 MRI-based images from the OAI repository on one node. In short, the workflow creates 10 HPC jobs, two per workflow task. First, the workflow was integrated with the Maestro WMS without WHESL support and executed without any failures. We refer to this testcase as the *Baseline* execution. Next, we scaled the number of DQE failures from 20% to 100% for the jobs in the workflow while executing with WHESL support. That means for each HPC job, the DQE failure was induced and then recovered by WHESL. Each testcase was executed five times, and without any resource allocation delay, as only 2-4 nodes were required for the whole workflow execution at any point in time. We compare the distribution of makespan for each testcase to analyze the efficiency of WHESL's exception management. Figure 4.7 depicts the makespan for all the testcases. As can be seen, the additional recovery time due to failures remains almost constant with increasing number of failures. When failures are induced, WHESL's support helps complete execution with an additional ≈1.6%-4.3% overhead even when every file access for writing results in an exception once for each job. This suggests that WHESL can provide robust FT against false failure detections.

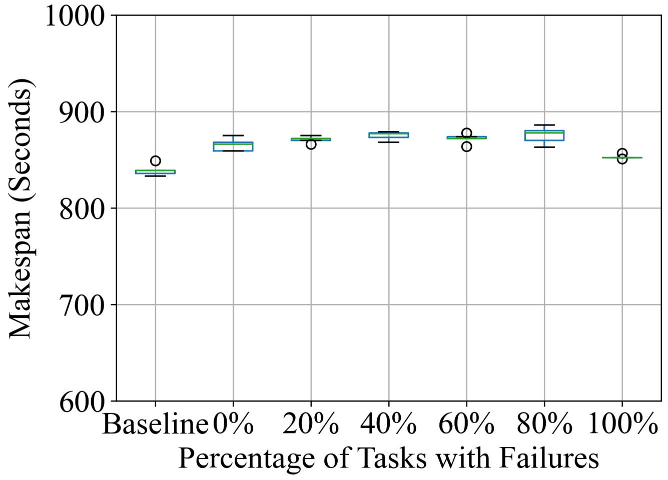


Figure 4.7 Impact of WHESL during Failure scaling without any queue wait times for jobs

To compare WHESL's recovery with the methods from existing WMSes, we introduced the additional overhead of waiting in the queue for HPC job submission. This is the wait time for the HPC jobs before the HPC RJMS grants the requested resources to begin their execution. With traditional recovery in WMSes, jobs are re-executed via resubmission of a failed job as a new job to the RJMS. This additional overhead is considered every time an HPC job is submitted. To model the wait queue for small jobs (in this case, 1 node), we analyzed the production jobs [LLN24b] of the LASSEN supercomputer and considered the median wait queue time. Figure 4.8 depicts the makespan of the OAI Analysis workflow with and without WHESL support. With increasing failure rate and without WHESL support, the makespan of the workflow increases linearly because of the added wait time overhead in the HPC job queue. As WHESL can isolate exceptions to the local allocation and restart the jobs in-place, workflow executions do not experience additional wait overhead.

Next, we analyze WHESL's performance under task scaling. To this end, we scale up the number of tasks per stage in the OAI workflow from 2 to 10, which results in 10-50 tasks for the complete workflow. We also induce one DQE exception per task at a 100% failure rate. We then compare the

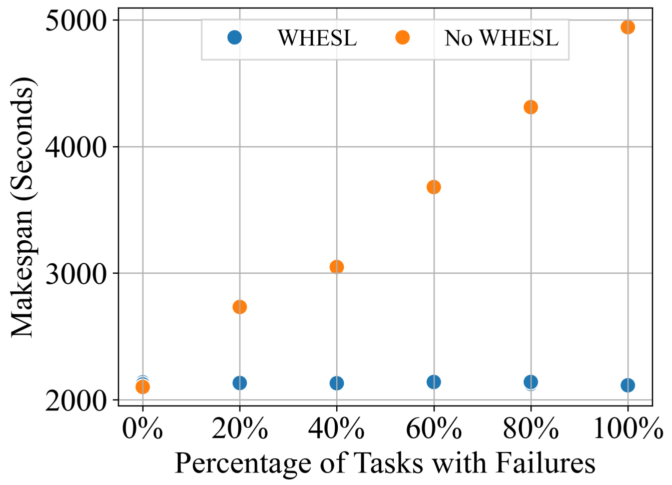


Figure 4.8 Impact of WHESL during Failure scaling with expected queue wait time for jobs

performance of WHESL in the presence of failures (WHESL runs) for larger workflows against the baseline runs, where workflow runs do not experience any failure while executing without WHESL support. Figure 4.9 shows the distribution of makespan (y-axis) of the baseline, and WHESL runs at 100% failure rate. We observe that makespan increases steadily for both baseline and WHESL runs with the task scaling due to the scheduling overhead at the WMS and HPC scheduler level. However, even with a 100% failure rate, WHESL runs' makespan closely match the baseline runs with a mean difference range between -2.09% and 2.53%.

We induce failures in the tasks around the same time when 50% of the execution is complete. As the *segmentation* and *registerimagetoatlas* stages run simultaneously, there can be twice the number of tasks per stage failures that can happen at the same time, and WHESL needs to handle 4-20 failures quickly. To analyze WHESL's efficiency, we measured the recovery time per failure. The recovery time is the duration between when a failure is detected and the replacement(restarted) job starts executing. Figure 4.10 shows the distribution of recovery times (y-axis) in seconds against different total numbers of tasks (x-axis) in the workflow. We observe that the recovery time distribution remains fairly similar as we scale the total number of tasks with the mean time in the range from 4.08 to 4.74 seconds. This shows the efficiency of WHESL in terms of recovery latency in handling failures for a large number of tasks.

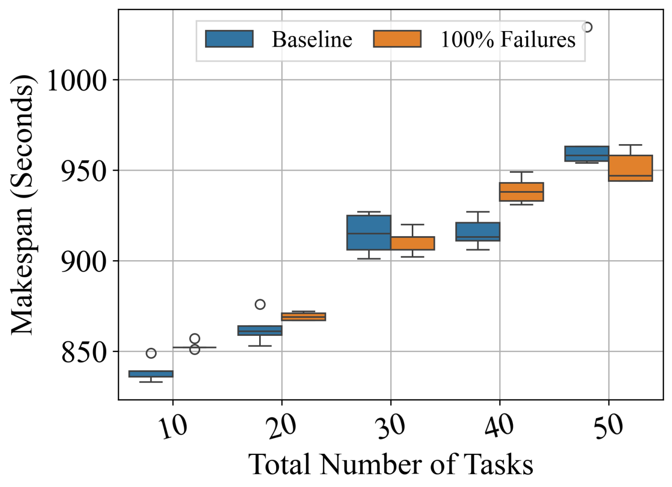


Figure 4.9 Impact of WHESL during task scaling without any queue wait time for jobs

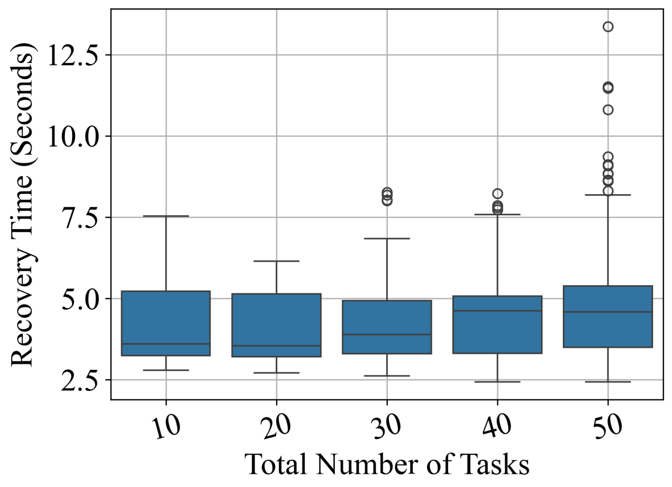


Figure 4.10 Recovery times for OAI from DQE

4.4.2 Managing Application Crash and Hang Exceptions

Application: LULESH [Kar13] is hydrodynamics proxy application developed by Lawrence Livermore National Laboratory (LLNL) to help evaluate performance of large-scale HPC systems. It has been ported to a number of programming models and we utilize the MPI-based LULESH for our evaluation purpose. The MPI model can execute only with a process of size p^3, where $p \in N$.

Exception: With LULESH, we evaluate WHESL with two exceptions: Application Hang (AC) and Application Crash (AH). An AH exception causes applications to get stuck permanently. AC causes applications to fail by crashing during execution. There are many root causes behind AC exceptions, with hardware failures, illegal device access by applications, and network-related errors being some of them. Similarly, AH exceptions happen because of deadlocks, communication-related problems such as network degradation and resource starvation. In this work, we focus on AC and AH exceptions resulting from network degradation. LULESH follows the execution of a typical parallel application on HPC systems, performing computation and communication/data synchronization iteratively. To cause the AC and AH exceptions, we deactivate and then reactivate the InfiniBand (IB) interface of one nodes. This causes IB link flaps during the library calls within the application to synchronize and exchange results for the simulation update. The IB link flaps cause LULESH to either get stuck in MPI blocking calls with an AH exception, or they experience an AC exception due to a crash. In our experiments, the resulting exception was non-deterministic. We performed repeated experiments to collect a sufficient number of samples for evaluation. The faults are injected at random intervals during the application execution. Further, we added SCR [Moo10] checkpointing support to LULESH every 10 timesteps.

Listing 4.3 Exception Specification for Application Crash

```
DECLARE ENTITY WMS, JOB
DECLARE EXCEPTION_TYPE JOB_LEVEL
DECLARE RESOURCE NODE[ADD, REMOVE, REPLACE]

DECLARE EXCEPTION CRASH JOB_LEVEL JOB [run-lulesh]
EXCEPTION DETECTION CRASH LOG "An error occurred in MPI_" stdout
EXCEPTION RECOVERY STEP CRASH JOB NODE.REPLACE
```

Listing 4.4 Exception Specification for Application Hang

```
DECLARE ENTITY WMS, JOB
DECLARE EXCEPTION_TYPE JOB_LEVEL
DECLARE RESOURCE NODE[ADD, REMOVE, REPLACE]
DECLARE EXCEPTION HANG JOB_LEVEL JOB [run-lulesh]
EXCEPTION DETECTION HANG LOG "cycle = " stdout TIMEOUT 120
EXCEPTION RECOVERY STEP HANG JOB NODE.REPLACE
```

Recovery: The desired recovery for AC and AH exceptions is to isolate the faulty node with IB link flaps and replace it with a healthy reserved node. A failed job should be restarted locally by the *WHESL Isolator* without a resubmission by the WMS. To achieve this, users provide exception specifications (see Listing 4.3 and Listing 4.4 for AC and AH exceptions, respectively). Lines 1-3 in both the listings define the execution ecosystem's entities, resources, and exception types managed by WHESL. Similarly, line 5 defines the exceptions AC and AH, indicating which jobs should be subject to such exceptions. Line 6 in Listing 4.3 indicates the log monitored for AC exceptions. In contrast, Line 6 in Listing 4.4 specifies the exact log pattern whose appearance *within* the specified time limit does *not* raise any AH exception. Conversely, if the given log appearance *exceeds* the timeout, WHESL raises an AH exception. The recovery step for both exceptions (line 7) is to replace the faulty node with a healthy one for a restart.

To this end, WHESL reserves two additional nodes per job (in our experiment). When Flux is instantiated, the broker services run on each node of the allocation. These brokers form a hierarchical overlay network and collectively provide services such as job scheduling, resource management, and RPC routing. By default, the rank 0 broker is the leader among the brokers, which is where WHESL runs. For that reason, WHESL reserves that node from being allocated for the job and fault injection. Further, an additional node is reserved for replacing the faulty node.

Analysis: To evaluate the impact of WHESL on LULESH, we executed it using Maestro WMS with and without WHESL's support. Each testcase was run with AC and AH fault injection, as well as without any fault injection. This serves as the baseline case. We repeated experiments for each test case five times and collected execution times. For testcases without WHESL support (e.g., OAI Analysis) we added additional overheads, such as resource allocation wait time, checkpoint read time, recomputation time for lost work, and failure detection time, to the baseline case. We also assume that a fault detection technique is integrated with Maestro WMS to detect AH exceptions. When an AC exception occurs, jobs exit with failures, and Maestro automatically detects it.

Figure 4.11 demonstrates the overall impact of WHESL on application performance in terms of makespan under AC and AH exceptions. Without any fault injection, the addition of WHESL support adds only a 0.6% to the mean makespan. However, when AC and AH faults are injected, WHESL's ability to manage exceptions locally within an allocation reduces makespan by 16.9% and 10.5%, respectively. Without WHESL support, Maestro needs to resubmit the failed job, which introduces additional overhead in waiting for new resources.

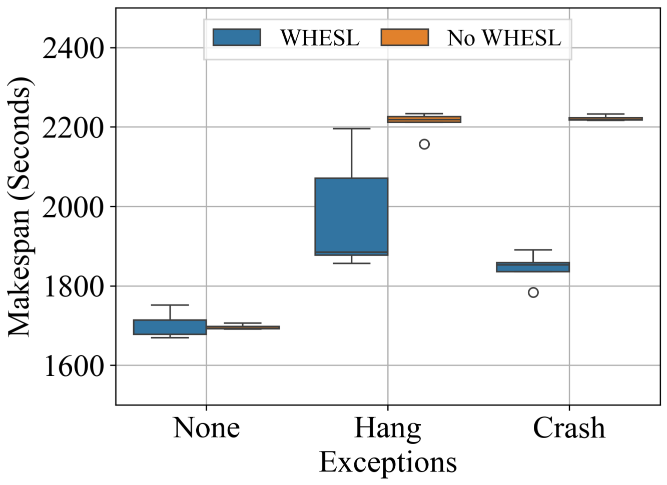


Figure 4.11 Impact of WHESL on LULESH Application

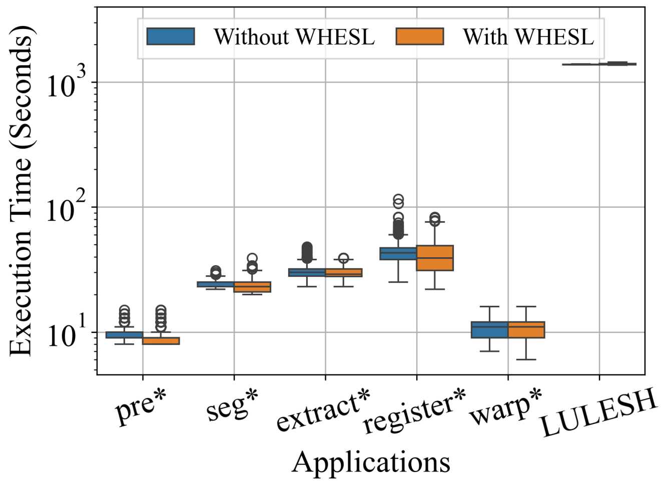


Figure 4.12 Impact of WHESL on HPC Jobs

4.4.3 Impact of WHESL on Application Performance

WHESL subscribes to the Flux job's output log events and parses them to detect failures. This may interfere with the execution of the actual workload running on a node. Further, Flux brokers run as an event-driven process on each node. To measure their impact on workload execution, we measured and plotted the individual applications' execution times from OAI Analysis and LULESH when run with and without WHESL's support (see Figure 4.12). We do not observe a significant difference in execution times, which is good news. We conclude that running WHESL in the background on actual workloads in the absence of exception has minimal performance impact.

4.5 Related Work

Prior research on fault tolerance for workflow systems has largely overlooked the hierarchical structure and operational complexity of modern HPC ecosystems. Existing approaches typically restrict failure recovery to task dependencies defined within the workflow, without considering broader system-level interactions. Grid Workflow [Hwa03] addresses this limitation in part by separating failure-handling logic from application code and allowing users to manage recovery policies within the WMS. The framework supports multiple recovery mechanisms, including retries, checkpointing, replication, and alternative task execution, enabling adaptability to heterogeneous and unreliable Grid resources. Unlike WHESL, it does not account for dependencies within workflow tasks or propagate the side effects of recovery actions to downstream dependent tasks. Moreover, the framework does not consider inefficiencies arising from redundant or conflicting recovery actions initiated by entities outside the WMS.

[TC08] enabled propagation of failures within the workflow hierarchy by adaptively restructuring workflows at runtime. Recovery patterns are defined within the workflow structure or specification based on the Reference-Nets-within-Nets formalism, enabling workflows to self-modify their structure when specific failure conditions are detected. Even after facilitating such propagation of failures, its failure management scope is limited to the workflow structure. Multiple WMSes discussed in Section 4.2 support similar failure management techniques. Our investigation indicates that Pegasus WMS provides the most comprehensive failure management among the systems listed in Table 4.3. Pegasus can propagate failures to dependent tasks within a workflow and dynamically reconfigure the dependent tasks. Unlike WHESL, it does not address the inefficiency that may arise due to multiple entities performing recovery from failures independently. Pegasus also requires each recovered task to be re-executed completely. Moreover, existing designs are limited to individual WMSs and lack portability across different workflow systems. [NVI25] supports in-process restart

Table 4.3 Exception Management Systems Comparison

Exception Model	**Specification**	**Scope**	**Workflow Hierarchy**	**Residence**	**User-defined Recovery**	**Coordination**	**Portability**
Tolosana-Calasanz et al.	Imperative	Workflow	✓	WMS	✗	✗	✗
Hwang et al.	Declarative	Sub-workflow	✗	WMS	✓	✗	✗
Pegasus	Declarative	Workflow	✓	WMS	✗	✗	✗
MCEM	✗	Workflow+	✗	Distributed	✗	✓	✗
Our Work	Declarative	Workflow	✓	Distributed	✓	✓	✓

for MPI applications, but are not applicable to workflows.

MCEM [Her19] introduced a conceptual model for hierarchical exception management, bringing cooperation between different subsystems to address redundant and inefficient recovery processes. Our work builds on this model to provide a practical and novel solution aimed at standardizing fault tolerance in HPC systems. The primary difference between WHESL and MCEM lies in their design choices. MCEM proposed embedding coordinators within various subsystems (namely the Network Manager, File System Manager, WMS, and system scheduler) to detect exceptions and coordinate recovery. However, MCEM lacked a practical coordination and recovery mechanism. In contrast, WHESL uses an exception isolation mechanism to avoid the performance overhead of communication between multiple subsystems. Exceptions are first isolated locally and then coordinated through a central coordinator to execute user-defined recovery steps, managing HPC resources via add, remove, and replace operations. Furthermore, MCEM does not consider workflow hierarchy for propagating exceptions, omitting a key requirement for managing failures in hierarchical workflows. Finally, MCEM has not been validated through practical evaluation. Table 4.3 compares WHESL with other systems and shows their unique differences.

4.6 Summary

We developed a distributed exception management system to mitigate the inefficiencies in recovery mechanisms of existing WMSes, runtime systems, and RJMSes. These inefficiencies are primarily due to a lack of coordination among the disparate components of the workflow ecosystem. Wastage of resources and loss of work occur as the existing recovery methods often result in unwanted redundancy or non-recoverable failures. Contrary to prior solutions, our developed system, WHESL, provides scalability for large workflows by leveraging Flux's nested scheduling capability and improves the efficiency of recovery from exceptions via coordination among the independent system entities. Its resource-centric recovery model allows users to define steps for exception management with ease. Furthermore, support for other WMSes can be added to WHESL without significant effort. Evaluations of WHESL demonstrate that for exceptions such as exceeding disk quota, it can withstand a 100% failure rate to provide near-baseline performance for the OAI Analysis workflow. Moreover, WHESL has negligible overhead under task scaling, with a mean difference ranging from -2.09% to 2.53%. For exceptions, Crash and Hang, WHESL can reduce the makespan by 16.9% and 10.5% for the LULESH application. Moreover, WHESL's impact on workload performance is negligible, which, in summary, is unprecedented.

CHAPTER

5 CONCLUSION

The extreme scale and heterogeneity of today's HPC systems pose significant challenges to achieving effective fault tolerance and deterministic execution for large-scale applications and workflows. Failure to address these challenges results in resource wastage, increased work loss, and inadequate support for time-critical workloads. In this thesis, we demonstrated that these challenges can be effectively addressed through three orthogonal but complementary solutions that leverage prioritization, predictability, and scalability across the HPC ecosystem.

First, we identified the limitations of existing failure-aware C/R solutions when failures have short lead times. Under such conditions, vulnerable nodes in large parallel applications cannot reliably store their program state to a persistent PFS due to I/O congestion. We further investigated the performance of state-of-the-art failure-aware C/R solutions under varying lead times to highlight their shortcomings. To address these limitations, we applied prioritization and coordinated the parallel ranks of the application, enabling vulnerable nodes to access the PFS without congestion. We also integrated complementary techniques, such as live migration, to reduce checkpoint frequency and overhead. We evaluated our hybrid C/R model against state-of-the-art solutions to measure its efficiency in terms of application execution overhead under multiple failure distributions. Furthermore, we analyzed the results to explain why the proposed technique outperforms previous approaches and discussed the trade-offs associated with these solutions.

Next, we investigated the challenges of scheduling large-scale workflows on HPC systems under deadline constraints. We found that the non-deterministic resource allocation policies adopted by HPC system schedulers prevent cloud-bursting–based solutions from effectively allocating and utilizing on-demand resources, resulting in resource idleness and missed deadlines. To address this issue, we explored predicting resource availability on HPC systems to construct predictive and timely workflow execution schedules that enable users to trade off cost and timeliness. Execution of such schedules is supported by techniques including dynamic resource allocation based on the workflow schedule, preloading of input data, and backing up HPC resources with cloud capacity. We evaluated and analyzed our framework against state-of-the-art solutions and found that it achieves higher task completion rates and cost savings, incurs lower impact on HPC production jobs, and exhibits low cost estimation error.

Finally, we developed a distributed exception management framework to address inefficient failure recovery caused by a lack of coordination among different entities in HPC systems. Without coordination, components such as workflow management systems and HPC schedulers detect and recover from failures independently. This can result in redundant recovery actions, resource wastage, and, in the worst case, data corruption or catastrophic failures. Our solution, WHESL, coordinates disparate components of the HPC ecosystem, mitigates redundant recovery actions through failure isolation, and provides scalable and portable fault tolerance.

Overall, we show that our approaches significantly reduce execution overhead, improve resource utilization, increase workflow completion rates, and provide reliable guarantees for time-critical workloads. Through rigorous evaluation and analysis, this thesis establishes that applying prioritization, predictability, and scalability to resource management is essential for resilient and efficient execution on next-generation HPC systems and provides practical foundations for future HPC workflow and system designs. In summary, we have empirically shown that our hypothesis holds.

BIBLIOGRAPHY

[al.16] al., A. Gupta et. “Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud”. *IEEE Transactions on Cloud Computing* **4**.3 (2016), pp. 307–321.

[al.19] al., B. Yadu et. “Parsl: Pervasive Parallel Programming in Python”. *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 2019.

[al.22a] al., C. Huang et. “DADP: Dynamic abnormality detection and progression for longitudinal knee magnetic resonance images from the Osteoarthritis Initiative”. *Medical Image Analysis* (2022), p. 102343.

[al.22b] al., C. Misale et. “Towards Standard Kubernetes Scheduling Interfaces for Converged Computing”. *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation*. Cham: Springer International Publishing, 2022, pp. 310–326.

[al.22c] al., D. J. Milroy et. “One Step Closer to Converged Computing: Achieving Scalability with Cloud-Native HPC”. *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 2022, pp. 57–70.

[al.18a] al., D. Ahn et. “Flux: Overcoming Scheduling Challenges for Exascale Workflows”. 2018, pp. 10–19.

[al.15] al., E. Deelman et. “Pegasus, a workflow management system for science automation”. *Future Generation Computer Systems* **46** (2015), pp. 17–35.

[al.18b] al., E. Deelman et. “The Future of Scientific Workflows”. *Int. J. High Perform. Comput. Appl.* **32**.1 (2018), 159–175.

[al.21a] al., J. L. Green et. “Molecular characterization of type I IFN-induced cytotoxicity in bladder cancer cells reveals biomarkers of resistance”. *Molecular Therapy-Oncolytics* **23** (2021), pp. 547–559.

[al.23] al., L. Ward et. “Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources”. *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2023, pp. 32–41.

[al.20a] al., N. Besaw et. “Cluster system management”. *IBM Journal of Research and Development* **64**.3/4 (2020), 7:1–7:9.

[al.17] al., P. Di Tommaso et et al. “Nextflow enables reproducible computational workflows”. *Nature biotechnology* **35**.4 (2017), pp. 316–319.

[al.20b] al., P. A. Ewels et. “The nf-core framework for community-curated bioinformatics pipelines”. *Nature biotechnology* **38**.3 (2020), pp. 276–278.

[al.21b] al., R. F. da Silva et. “A Community Roadmap for Scientific Workflows Research and Development”. *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*. 2021, pp. 81–90.

[Alb12] Albrecht, M. et al. “Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids”. *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. SWEET ’12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012.

[All12] Allen, B. et al. “Software as a service for data scientists”. *Commun. ACM* **55**.2 (2012), 81–88.

[Ass09] Assuncao, M. D. de et al. “Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters”. *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*. HPDC ’09. Garching, Germany: Association for Computing Machinery, 2009, 141–150.

[AWS24] AWS. 2024. URL: `https://aws.amazon.com/ec2/instance-types`.

[Bal16] Balasubramanian, V. et al. “Ensemble toolkit: Scalable and flexible execution of ensembles of tasks”. *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE. 2016, pp. 458–463.

[BG11] Bautista-Gomez, L. et al. “FTI: High Performance Fault Tolerance Interface for Hybrid Systems”. *Supercomputing*. 2011.

[Beh20] Behera, S. et al. “Orchestrating Fault Prediction with Live Migration and Checkpointing”. *Symposium on High Performance Distributed Computing*. 2020.

[Beh22] Behera, S. et al. “P-ckpt: Coordinated Prioritized Checkpointing”. *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2022, pp. 436–446.

[Ben15] Ben Belgacem, M. & Chopard, B. “A hybrid HPC/cloud distributed infrastructure: Coupling EC2 cloud resources with HPC clusters to run large tightly coupled multiscale applications”. *Future Generation Computer Systems* **42** (2015), pp. 11–21.

[Ben17] Benoit, A. et al. “Towards Optimal Multi-Level Checkpointing”. *IEEE Transactions on Computers* **66**.7 (2017), pp. 1212–1226.

[Bhi16] Bhimji, W. et al. “Accelerating Science with the NERSC Burst Buffer Early User Program”. 2016.

[Bic12] Bicer, T. et al. “Time and Cost Sensitive Data-Intensive Computing on Hybrid Clouds”. *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. 2012, pp. 636–643.

[Bou13] Bouguerra, M. et al. “Improving the Computing Efficiency of HPC Systems Using a Combination of Proactive and Preventive Checkpointing”. *International Parallel and Distributed Processing Symposium*. 2013, pp. 501–512.

[Com24] Community, T. G. “The Galaxy platform for accessible, reproducible, and collaborative data analyses: 2024 update”. *Nucleic Acids Research* **52**.W1 (2024), W83–W94. eprint: `https://academic.oup.com/nar/article-pdf/52/W1/W83/58436305/gkae410.pdf`.

[Com25] Computing, A. *Moab HPC Suite*. Advanced workload and resource orchestration platform often used with TORQUE/PBS. Adaptive Computing. 2025.

[Cun23] Cunningham, W. et al. *AgnostiqHQ/covalent: v0.228.0-rc.0*. Version v0.228.0-rc.0. 2023.

[Das20a] Das, A. “Aarohi: Making Real-Time Node Failure Prediction Feasible”. *International Parallel and Distributed Processing Symposium*. 2020.

[Das18a] Das, A. & Mueller, F. “Aarohi: Automaton-based Low-cost Online Failure Prediction”. *SC Poster Session*. 2018.

[Das17] Das, A. et al. “Desh: Deep Learning for HPC System Health Resilience”. *SC Poster Session*. 2017.

[Das18b] Das, A. et al. “Desh: Deep Learning for System Health Prediction of Lead Times to Failure in HPC”. *Symposium on High Performance Distributed Computing*. 2018, pp. 40–51.

[Das18c] Das, A. et al. “Doomsday: Predicting Which Node Will Fail When on Supercomputers”. *Supercomputing*. 2018, 9:1–9:14.

[Das20b] Das, A. et al. “Aarohi: Making Real-Time Node Failure Prediction Feasible”. *International Parallel and Distributed Processing Symposium*. 2020, pp. 1092–1101.

[dev25a] developers, C. *Cromwell documentation*. Web page. `https://cromwell.readthedocs.io/en/latest/` (accessed 2025-12-08). 2025.

[dev25b] developers, R. *RADICAL.AsyncFlow — Workflow & Asynchronous Execution*. Web page. `https://radical-cybertools.github.io/radical.asyncflow/` (accessed 2025-12-08). 2025.

[Dev24] Developers, S. 2024. URL: `https://xgboost.readthedocs.io/en/stable/`.

[Di14] Di, S. et al. “Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications”. *International Parallel and Distributed Processing Symposium*. 2014, pp. 1181–1190.

[Di17] Di, S. et al. “Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model”. *IEEE Transactions on Parallel and Distributed Systems* **28**.1 (2017), pp. 244–259.

[DN25] Di Natale, F. *Maestro Workflow Conductor (maestrowf)*. Version 1.1.11. accessed 2025-12-07. 25, 2025.

[DN19] Di Natale, F. et al. “A massively parallel infrastructure for adaptive multiscale simulations: modeling RAS initiation pathway for cancer”. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’19. Denver, Colorado: Association for Computing Machinery, 2019.

[Fan] Fang, A. & Chien, A. A. “How Much SSD Is Useful for Resilience in Supercomputers”. *Workshop on Fault Tolerance for HPC at eXtreme Scale*, pp. 47–54.

[FDS21] Ferreira Da Silva, R. et al. “A Community Roadmap for Scientific Workflows Research and Development”. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States). 2021.

[FDS23] Ferreira Da Silva, R. et al. *Workflows Community Summit 2022: A Roadmap Revolution*. Tech. rep. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), 2023.

[FDS24] Ferreira Da Silva, R. et al. *Workflows Community Summit 2024: Future Trends and Challenges in Scientific Workflows*. Tech. rep. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States). Oak Ridge Leadership Computing Facility (OLCF), 2024.

[Gai12a] Gainaru, A. et al. “Fault Prediction under the Microscope: A Closer Look into HPC Systems”. *Supercomputing*. 2012.

[Gai12b] Gainaru, A. et al. “Taming of the Shrew: Modeling the Normal and Faulty Behaviour of Large-scale HPC Systems”. *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012), pp. 1168–1179.

[Gar18] Garg, R. et al. “Shiraz: Exploiting System Reliability and Application Resilience Characteristics to Improve Large Scale System Throughput”. *International Conference on Dependable Systems and Networks*. 2018, pp. 83–94.

[Gei03] Geist, A. & Engelmann, C. “Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors” (2003).

[Geo15] George, C. & Vadhiyar, S. “Fault Tolerance on Large Scale Systems using Adaptive Process Replication”. *IEEE Transactions on Computers* **64**.8 (2015), pp. 2213–2225.

[Guo14] Guo, T. et al. “Cost-Aware Cloud Bursting for Enterprise Applications”. **13**.3 (2014).

[Gup17] Gupta, S. et al. “Failures in large scale systems: long-term measurement, analysis, and implications”. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–12.

[Her19] Herbein, S. et al. “Mcem: Multi-level cooperative exception model for hpc workflows”. *Proceedings of the 9th International Workshop on Runtime and Operating Systems for Supercomputers*. 2019, pp. 27–32.

[Hub20] Huber, S. P. et al. “AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance”. *Scientific Data* **7**.1 (2020), p. 300.

[Hud22] Hudson, S. et al. “libEnsemble: A Library to Coordinate the Concurrent Evaluation of Dynamic Ensembles of Calculations”. *IEEE Transactions on Parallel and Distributed Systems* **33**.4 (2022), pp. 977–988.

[Hwa03] Hwang, S. & Kesselman, C. “Grid workflow: a flexible failure handling framework for the grid”. *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*. IEEE. 2003, pp. 126–137.

[IBM24] IBM. 2024. URL: `https://www.ibm.com/docs/en/spectrum-lsf/10.1.0`.

[Ini25] Initiative, W. C. 2025. URL: `https://workflows.community/`.

[Isk08] Iskra, K. et al. “ZOID: I/O-forwarding Infrastructure for Petascale Architectures”. *Symposium on Principles and Practice of Parallel Programming*. 2008, pp. 153–162.

[Jai15] Jain, A. et al. “FireWorks: a dynamic workflow system designed for high-throughput applications”. *Concurrency and Computation: Practice and Experience* **27**.17 (2015), pp. 5037–5059. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3505`.

[Jan19] Janssen, J. et al. “pyiron: An integrated development environment for computational materials science”. *Computational Materials Science* **163** (2019), pp. 24 –36.

[Jia24] Jiang, Z. et al. “{MegaScale}: Scaling large language model training to more than 10,000 {GPUs}”. *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 745–760.

[Kar13] Karlin, I. et al. *Tuning the LULESH mini-app for current and future hardware*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[Kok25] Kokolis, A. et al. "Revisiting reliability in large-scale machine learning research clusters". *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2025, pp. 1259–1274.

[Kös12] Köster, J. & Rahmann, S. "Snakemake—a scalable bioinformatics workflow engine". *Bioinformatics* **28**.19 (2012), pp. 2520–2522.

[Lab25] Laboratory, I. A. N. *Cobalt Scheduler*. Resource management and job scheduling system used on Blue Gene supercomputers; includes multi-dimensional scheduling capabilities. 2025.

[Liu12] Liu, N. et al. "On the role of burst buffers in leadership-class storage systems". *Symposium on Mass Storage Systems and Technologies (MSST)*. 2012, pp. 1–11.

[Liu08] Liu, Y. et al. "An optimal checkpoint/restart model for a large scale high performance computing system". *International Parallel and Distributed Processing Symposium*. 2008, pp. 1–9.

[LLN24a] LLNL. 2024. URL: `https://hpc.llnl.gov/hardware/compute-platforms/lassen`.

[LLN24b] LLNL. 2024. URL: `https://github.com/LLNL/LAST/tree/main`.

[Lua25] Luan, F. S. et al. "The streaming batch model for efficient and fault-tolerant heterogeneous execution". *arXiv preprint arXiv:2501.12407* (2025).

[Luc14] Lucas, R. et al. "DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges" (2014).

[Mao12] Mao, M. & Humphrey, M. "A Performance Study on the VM Startup Time in the Cloud". *2012 IEEE Fifth International Conference on Cloud Computing*. 2012, pp. 423–430.

[Mau24] Maurya, A. et al. "DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models". *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '24. Pisa, Italy: Association for Computing Machinery, 2024, 227–239.

[Moo10] Moody, A. et al. "Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System". *Supercomputing*. 2010, 1–11.

[Mor18] Moritz, P. et al. “Ray: A distributed framework for emerging {AI} applications”. *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 2018, pp. 561–577.

[Nar21] Narayanan, D. et al. “Efficient large-scale language model training on gpu clusters using megatron-lm”. *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2021, pp. 1–15.

[Nit04] Nitzberg, B. et al. “PBS Pro: Grid computing and scheduling attributes”. *Grid resource management: state of the art and future trends*. Springer, 2004, pp. 183–190.

[NVI25] NVIDIA. *nvidia-resiliency-ext: NVIDIA Resiliency Extension*. `https://github.com/NVIDIA/nvidia-resiliency-ext`. Accessed: 2025-01-07. 2025.

[Oli19] Oliver, H. et al. “Workflow Automation for Cycling Systems”. *Computing in Science & Engineering* **21**.4 (2019), pp. 7–21.

[ORN20] ORNL. *Spectral Library*. 2020. URL: `https://www.olcf.ornl.gov/spectral-library/`.

[Par14] Parr, T. et al. “Adaptive LL(*) parsing: the power of dynamic analysis”. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &amp; Applications*. OOPSLA ’14. Portland, Oregon, USA: Association for Computing Machinery, 2014, 579–598.

[Par22] Partee, S. et al. “Using Machine Learning at scale in numerical simulations with SmartSim: An application to ocean climate modeling”. *Journal of Computational Science* **62** (2022), p. 101707.

[Roc15] Rocklin, M. “Dask: Parallel Computation with Blocked algorithms and Task Scheduling”. *Proceedings of the 14th Python in Science Conference*. Ed. by Huff, K. & Bergstra, J. 2015, pp. 130 –136.

[Roy22a] Roy, R. B. et al. “DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts”. *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2022, pp. 1–18.

[Roy22b] Roy, R. B. et al. “Mashup: making serverless computing useful for HPC workflows via hybrid execution”. *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, 46–60.

[Sat12] Sato, K. et al. “Design and modeling of a non-blocking checkpointing system”. *Supercomputing*. 2012, pp. 1–10.

[Sch07] Schroeder, B. & Gibson, G. A. “Understanding failures in petascale computers”. *Journal of Physics: Conference Series* **78** (2007), p. 012022.

[She19] Shen, Z. et al. “Networks for joint affine and non-parametric image registration”. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4224–4233.

[SD23] Sly-Delgado, B. et al. “TaskVine: Managing In-Cluster Storage for High-Throughput Data Intensive Workflows”. *Proceedings of the SC ’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W ’23. Denver, CO, USA: Association for Computing Machinery, 2023, 1978–1988.

[Tea20] Team, S. *SimPy: Discrete-Event Simulation for Python*. 2020. URL: `https://pypi.org/project/simpy/`.

[Tej17] Tejedor, E. et al. “PyCOMPSs: Parallel computational workflows in Python”. *The International Journal of High Performance Computing Applications* **31**.1 (2017), pp. 66–82. eprint: `https://doi.org/10.1177/1094342015594678`.

[Tiw14] Tiwari, D. et al. “Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems”. *International Conference on Dependable Systems and Networks*. 2014, pp. 25–36.

[TC08] Tolosana-Calasanz, R. et al. “Exception handling patterns for hierarchical scientific workflows”. *Proceedings of the 6th international workshop on Middleware for grid computing*. 2008, pp. 1–6.

[Too18] Toosi, A. N. et al. “Resource provisioning for data-intensive applications with deadline constraints on hybrid clouds using Aneka”. *Future Generation Computer Systems* **79** (2018), pp. 765–775.

[Top] “Top 500 List”. http://www.top500.org/. 2023.

[Vaz18] Vazhkudai, S. S. et al. “The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems”. *Supercomputing*. 2018.

[Viv17] Vivian, J. et al. “Toil enables reproducible, open source, big biomedical data analyses”. *Nature Biotechnology* **35**.4 (2017), pp. 314–316.

[Wan17] Wan, L. et al. “Optimizing checkpoint data placement with guaranteed burst buffer endurance in large-scale hierarchical storage systems”. *Journal of Parallel and Distributed Computing* **100** (2017), pp. 16 –29.

[Wan08] Wang, C. et al. “Proactive Process-Level Live Migration in HPC Environments”. *SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 2008.

[Wan12] Wang, C. et al. “Proactive Process-Level Live Migration and Back Migration in HPC Environments”. *Journal of Parallel Distributed Computing* **72**.2 (2012), pp. 254–267.

[Wil11] Wilde, M. et al. “Swift: A language for distributed parallel scripting”. *Parallel Computing* **37**.9 (2011). Emerging Programming Paradigms for Large-Scale Scientific Computing, pp. 633–652.

[Yoo03] Yoo, A. B. et al. “SLURM: Simple Linux Utility for Resource Management”. *Job Scheduling Strategies for Parallel Processing*. Ed. by Feitelson, D. et al. 2003, pp. 44–60.

[You74] Young, J. W. “A first order approximation to the optimum checkpoint interval”. *Commun. ACM* **17**.9 (1974), pp. 530–531.

[Yu19] Yu, Y. et al. “A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures”. *Neural Computation* **31**.7 (2019), pp. 1235–1270. eprint: `https://direct.mit.edu/neco/article-pdf/31/7/1235/1053200/neco\_a\_01199.pdf`.

[Zu24] Zu, Y. et al. “Resiliency at Scale: Managing {Google’s}{TPUv4} Machine Learning Supercomputer”. *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 761–774.