SchedInspector: A Batch Job Scheduling Inspector Using Reinforcement Learning

Di Zhang
Computer Science Department, University of North Carolina at Charlotte
Charlotte, NC, USA
dzhang16@uncc.edu

Dong Dai
Computer Science Department, University of North Carolina at Charlotte
Charlotte, NC, USA
ddai@uncc.edu

Bing Xie
Oak Ridge Leadership Computing Facility, Oak Ridge National Laboratory
Oak Ridge, TN, USA
xieb@ornl.gov

ABSTRACT

Improving the performance of job executions is an important goal of HPC batch job schedulers, such as minimizing job waiting time, slowdown, or completion time. Such a goal is often accomplished using carefully designed heuristics based on job features, such as job size and job duration. However, these heuristics overlook important runtime factors (e.g., cluster availability and waiting job patterns), which may vary across time and make a previously sound scheduling decision not hold any longer. In this study, we propose a new approach to incorporate runtime factors into batch job scheduling for better job execution performance. The key idea is to add a *scheduling inspector* on top of the base job scheduler to scrutinize its scheduling decisions. The inspector will take the runtime factors into consideration and accordingly determine the fitness of the scheduled job. It then either accepts the scheduled job or rejects it and asks the base schedulers to try again later. We realize such an inspector, namely SchedInspector, by leveraging the intelligence of reinforcement learning. Through extensive experiments, we show SchedInspector can intelligently integrate the runtime factors into various batch job scheduling policies, including the state-of-the-art one, to gain better job execution performance, such as smaller average bounded job slowdown (up to 69% better) or average job waiting time (up to 52% better), across various real-world workloads. We also show that although rejecting scheduling decisions may leave the resources idle hence affect the system utilization, SchedInspector is able to achieve the job execution performance improvement with marginal impact on the system utilization (typically less than 1%). We consider one key advantage of SchedInspector is it automatically learns to work with and improve existing job scheduling policies without changing them, which makes it promising to serve as a generic enhancer for various batch job scheduling policies.

CCS CONCEPTS

• **Software and its engineering → Scheduling**; • **Computer systems organization → Parallel architectures**.

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

KEYWORDS

High Performance Computing (HPC); batch job scheduling; reinforcement learning

ACM Reference Format:
Di Zhang, Dong Dai, and Bing Xie. 2022. SchedInspector: A Batch Job Scheduling Inspector Using Reinforcement Learning. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3502181.3531470

1 INTRODUCTION

Batch jobs are still dominating in high-performance computing (HPC) platforms. End users submit batch jobs via job scripts which describe detailed job properties such as resource requests or walltime. These batch jobs will be scheduled by a centralized batch job scheduler that utilizes predefined scheduling policies, such as first-come-first-serve (FCFS) or shortest job first (SJF).

Built around such jobs, various batch job scheduling policies are designed/engineered to improve the performance of batch job executions via optimizing metrics like minimizing job waiting time, job slowdown, or completion time. In theory, designing optimal batch job scheduling policy is proved NP-hard [33]. Hence, in practice, the policies based on heuristics have been studied extensively [3–5, 9, 23, 32, 35]. Most of them define the heuristics based on known job features from the job scripts, such as job requested resources (res_j), job estimated execution time (est_j), to make scheduling decisions. Different heuristic scheduling policies may weight job features differently, such as Shortest Job First (SJF) relies only on est_j, Smallest Resource Requirement First (SQF) only on res_j [24], Smallest estimated Area First (SAF) on $est_j * res_j$, Smallest estimated Ratio First (SRF) on est_j/res_j [19], and machine learning-based policies (e.g., F1 [9]) on a non-linear combination of multiple factors such as res_j, est_j, and waiting time.

However, only considering job features overlooks critical runtime factors, such as the available resources in the cluster, the distribution of overall waiting jobs, or the arrival patterns of future jobs, which will impact the scheduling outcomes. Once these factors changed, a scheduling policy might not perform as expected anymore. In such cases, abandoning current scheduling decision and taking an opportunistic action towards future jobs may be beneficial. For example, SJF policy prioritizes the shortest jobs, aiming at reducing the average job waiting time. But, if currently the cluster is almost full and all the waiting jobs have long execution time, then even running the shortest one among them will make the cluster full for a long time, blocking other jobs and leading to a

larger job waiting time. In such case, it might be beneficial to avoid running the current job and wait until next scheduling point to schedule a different job (a detailed example is given in § 2.1).

In this study, we propose a new approach to incorporate runtime factors into a batch job scheduling policy to improve its performance. Specifically, we introduce a *scheduling inspector* to scrutinize the scheduling decisions made by the existing scheduling policy. The inspector will take runtime factors into consideration and accordingly decide whether the scheduled job fits the runtime. If it believes the current job as a good fit for the runtime, the scheduling continues as normal. Otherwise, this scheduling decision will be rejected and the job will be put back to the waiting queue and be considered again at the next scheduling point. In another word, the scheduling is 'held' and 'delayed' to a later time. At the next scheduling point, the runtime factors may have changed: new jobs might arrive and be added into the waiting queue and cluster availability might change. The same scheduling policy may generate better scheduling decision this time and eventually improve the overall job execution performance.

Beyond the potentials on performance improvement, inspecting the scheduling and rejecting the unfit jobs also introduce overheads as it delays the execution of the scheduled job and accordingly leaves the resources idle shortly. Such a delay may become a pure waste of time and resources if runtime factors do not change at the next scheduling point or the decisions made in the future reward back the same or worse. Intuitively, making the rejection beneficial relies on two facets: the precise assessment on whether a job fits to the current runtime, and the accurate predictions about the jobs and the runtime in the future. Not surprisingly, both of the facets are challenging due to the complications in the HPC environment.

In this work, we show that rejecting scheduling decisions can be consistently beneficial if done properly. To achieve this, instead of expecting benefits from every inspection and rejection, we take a *statistical approach* to identify the 'big-gain' and 'small-loss' opportunities and build high confidence on these opportunities. For instance, in SJF, if the current shortest job is still too long and requests excessive resources, and the cluster is almost full, then rejecting this job and accordingly yielding resources to shorter jobs arriving in the near future has higher possibility to be beneficial and hence should be taken (see the examples in § 2.1).

We leverage deep reinforcement learning (RL) [30] to realize such a statistical approach. In particular, we implement SchedInspector, an RL-based inspector that automatically learns how to inspect scheduling decisions via continuously trial-and-error. We introduce two new reinforcement learning designs to improve the accuracy and efficiency of SchedInspector. More details are included in §3. Different from existing batch job scheduling studies, SchedInspector can automatically learns to work with and improve existing batch job scheduling policies, called base scheduler, without altering their priority heuristics. Because of this design, SchedInspector may serve as a generic enhancer for various scheduling policies. We summarize our contributions into threefold:

- To the best of our knowledge, SchedInspector is the first study that introduces *scheduling inspector* to integrate runtime factor into existing batch job scheduling for better job execution performance.

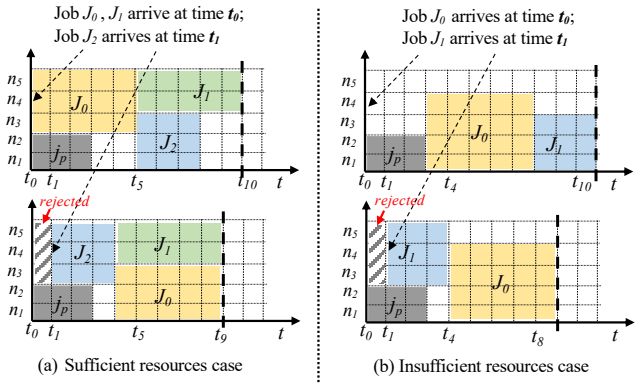

Figure 1: Scheduling jobs with/without SchedInspector. ***x-axis shows the timeline in minutes; y-axis shows the compute nodes ($n_{1\to5}$); each block represents a job that takes amount of nodes and time; J_p is the preliminary job running before the scheduling starts.***

- We leverage reinforcement learning (RL) and propose two new RL optimizations to automatically and efficiently learn inspecting job scheduling. We carefully analyze and summarize the statistical rules learned by SchedInspector.
- Via extensive experiments, we carefully discuss how SchedInspector performs on various job scheduling policies under various workloads. We also analyze its trade-off between job execution performance and system utilization.

The remainder of this paper is organized as follows: In §2 we motivate SchedInspector via a detailed example and discuss its challenges. We also briefly introduce the concept of deep reinforcement learning. In §3 we present SchedInspector and its key designs and optimizations. We present the main results in §4. We summarize and discuss the learned rules in §5. We compare with related work in §6, conclude this paper and discuss the future work in §7.

2 MOTIVATION AND BACKGROUND

2.1 An Example of SchedInspector

We use a simple example to demonstrate the potentials of SchedInspector. Here, we schedule job sequences on a 5-node cluster using Shortest Job First (SJF) as the base scheduling policy (without backfilling). We discuss two scheduling cases based on whether the shortest job has enough resources to run or not at the scheduling point. We report performance of both cases without (*top*) and with SchedInspector (*down*) in Figure 1.

Figure 1(a) shows the case when the selected shortest job has sufficient resources to run immediately. Without SchedInspector (*top*), when jobs J_0 and J_1 arrive at time t_0, SJF schedules J_0 as it is the shortest one (break tie by smaller id). Later, J_2 arrives at time t_1. It has higher priority than J_1, hence will run first when J_0 finishes. Note that, J_1 cannot run when J_p finishes because its priority is lower than J_2. In this way, the entire job sequence finishes at t_{10}. Comparatively, the *bottom figure* shows the results when SchedInspector is enabled. In this case, J_0 is rejected at t_0, SJF scheduler will put it back into waiting queue and try again at t_1 (when J_2 arrives). At this point, SJF will select J_2 due to its shortest

Table 1: Performance metrics of the example cases.

Scheduling Cases	*Waiting time*	*Bounded job slowdown*
Case(a)-NoInspect	(0+5+4)/3=3	(1+2+2.3)/3=1.77
Case(a)-Inspected	(4+4+1)/3=3	(1.8+1.8+1)/3=1.53
Case(b)-NoInspect	(3+7)/2=5	(1.6+3.3)/2=2.45
Case(b)-Inspected	(4+0)/2=2	(1.8+1)/2=1.4

execution time. After finishing J_2, the other two jobs (J_0 and J_1) can start at t_4. As we can see, although rejecting J_0 delays the scheduling and introduces an idle time (marked as gray), the entire job sequence completes earlier at t_9.

Figure 1(b) shows the case when the selected shortest job cannot run immediately due to insufficient resources. Without SchedInspector (*top*), J_0 arrives first and will be scheduled by SJF. But it needs to wait until enough resources are released. J_1 runs after J_0 and finishes at t_{10}. If SchedInspector is enabled and J_0 is rejected at t_0, the scheduler will then observe both J_0 and J_1 at the next scheduling point when new job arrives at t_1. At this time, SJF will schedule J_1 due to its shorter execution time. Accordingly, the future job J_1 finishes earlier and eventually leads to an earlier completion of the whole job sequence.

In addition to the completion time of the whole job sequences, we also calculate two more performance metrics: average job waiting time (*wait*) and average bounded job slowdown (*bsld*) to validate our results. Both metrics are also used in our later evaluations. The results are in Table 1, which again show SchedInspector improves both metrics.

- **average waiting time (*wait*)**: the average duration between the job's submission and its start time.
- **average bounded slowdown (*bsld*)**: the job slowdown relative to the given 'interactive thresholds' (10 seconds) calculated from $max((wait_j + exe_j)/max(exe_j, 10), 1)$ (waiting time $wait_j$ and execution time exe_j).

2.2 Challenges and Opportunities

When taking a closer look into the aforementioned two examples, we notice that the better performance comes from the cases: 1) new jobs arrived and were added into the waiting queue before the next scheduling point; 2) the new jobs match the cluster availability better and are scheduled to improve the performance. Then, it is expected, small variations, such as J_2 in *Case(a)* requests more resource, may lead to a totally different result. In a real system, we cannot eliminate such variations nor accurately predict the future, which makes implementing the inspector extremely challenging.

In this work, we argue that, although obtaining perfect result on each inspection is impractical, identifying the *statistically* 'big gain' and 'small loss' opportunities is still achievable. For instance, assuming the scenario where SJF is used as the base scheduling policy and the current shortest job requests more execution time and more resources than the historical jobs do on average. Moreover, if we assure that the average job arrival interval is small, then rejecting the current job is likely more beneficial than scheduling/running it since the rejection may yield the resources to the upcoming jobs and achieve better overall performance. Although we cannot accurately predict the exact features or the arrivals of future jobs, from the historical job and environmental statistics, we still can learn when rejection has higher chance to win.

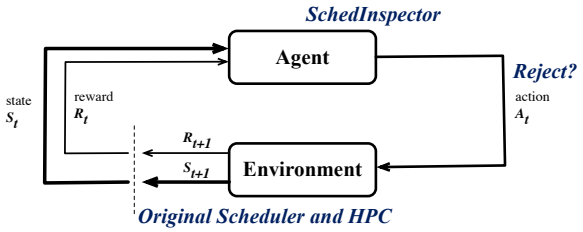

Figure 2: General framework of reinforcement learning.

In summary, the key of SchedInspector is to learn a statistically winning strategy. Such a strategy will be closely relevant to the historical job trace statistics (such as average job size and arrival interval), the real-time scheduling environment (such as currently scheduled job and waiting jobs), and the currently used scheduler. Such information is highly dynamic and complex, and can hardly be modeled using traditional heuristic methods. In this study, we leverage deep reinforcement learning to capture such knowledge automatically.

2.3 The Basics of Reinforcement Learning

Reinforcement learning is a type of machine learning technique that enables an agent to autonomously learn in an interactive environment by trials and errors using feedback from its own actions and experiences [17, 30].

Figure 2 shows a general RL framework and how SchedInspector fits into it. Here, SchedInspector will be the agent. At each time step t, it observes a corresponding state S_t (include the original scheduling decision and other scheduling relevant states), and takes an action A_t ('reject' or not) based on its internal *policy*. Consequently, the action will transfer the environment (HPC system) state from S_t to S_{t+1} and the agent will receive the reward R_{t+1} (measured based on the given performance metrics). RL repeats this until reaches the final state. The goal of reinforcement learning is to learn a policy that can maximize the expected cumulative discounted reward collected from the environment. More details about RL and its training methods can be seen in [30].

3 SCHEDINSPECTOR DESIGN

Figure 3 presents the SchedInspector architecture and its major components. It adheres to the general reinforcement learning framework shown in Figure 2, with the key components of *State* (Environment State), *Action* (RL Agent), *Reward* and *Environment* (Simulated Environment).

SchedInspector performs training following the typical policy-gradient RL training workflow [31]. It starts from the simulated environment simulating job arrivals. The arrived jobs are from a job sequence randomly sampled from the job trace file (❶). Next, at a scheduling point, the base scheduling policy (e.g., SJF) picks a job. Accordingly, the full scheduling contexts (e.g., the scheduled job, waiting queue, cluster status) will be formulated as environmental state (❷), which will be further summarized into concise *State Features* (❸) and fed to the RL agent to generate the inspection

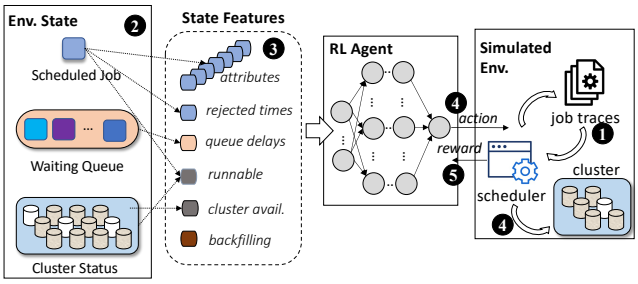

Figure 3: Architecture of SchedInspector.

result. The result will be applied to the simulated environment (discussed in Section §3.2) and transforms its internal state from S_t to S_{t+1} (❹). Finally, the *reward* for this action is calculated (❺) and returned to the agent for training.

Since most of the job execution performance metrics (e.g., average waiting time or bounded job slowdown) can only be calculated after the entire sequence of jobs get scheduled, during reward calculation, we hold the intermediate rewards of each individual action to be 0 and only calculate the final reward after finishing the last job in a job sequence. We discuss more details about the reward calculation in §3.4. Once the entire job sequence is scheduled, we build a trajectory that contains multiple inspection actions and a final reward. After creating a batch of such trajectories, SchedInspector trains its neural networks from these actions and their associated rewards using policy-gradient algorithm. For inference, SchedInspector acts similarly as it does in the training process but outputs its actions to actual HPC clusters.

3.1 RL Agent

RL agent serves as the brain of SchedInspector. It takes the top-priority job and the current scheduling context as inputs and generates a binary *action* (reject or not) as the decision. In SchedInspector, the RL agent uses a 3-layer fully connected neural networks, structured the same as 3-layer perceptron (MLP) [8], as its core network. As shown in Figure 3, besides the input layer and the one dimensional output layer (reject or not), the three hidden layers in our MLP have 32, 16, and 8 neurons respectively. The total parameter size of the network is 938. We choose MLP for two reasons: 1) it has been widely used in RL-based optimization research for its flexibility and simplicity [20, 21, 25]; 2) as we will show in the following section, our feature selection mechanism highly simplifies the inputs of RL agent and enables us to use simple networks such as MLP to achieve high training accuracy.

SchedInspector uses Actor-Critic model [18] to accelerate and stabilize the training. The core idea is to introduce an extra *value network* as a critic to work with the original *policy network*. These two networks use the same architecture and take the same inputs, but output different values. The policy network outputs delay action; the value network outputs the expected cumulative rewards of current state, which serves the policy network as the baseline reward of similar states to guide its training. Such a baseline will help stabilize the policy network. Without the value network, we observed high variations during the training. As these two networks are the same, we do not differentiate them in Figure 3.

3.2 Simulated Environment

Training an RL model often needs enormous interactions between the agent and the environment. It is impractical to perform such training in a real HPC cluster. Hence SchedInspector conducts training using a simulated HPC environment (*Simulated Env*). In SchedInspector, we build our simulated environment based on SchedGym, a RL-compatible simulator implemented in RLScheduler [39]. We extended it to acknowledge the reject decisions as well as to support more base job schedulers. We added state trackers to record job states such as how many times a job has been rejected.

The simulator works as Figure 3. Each time the base job scheduling policy selects a top-priority job, it will run the RL agent to inspect it. If RL returns 'reject', the simulator will cancel the scheduling, put the job back into the waiting queue, and move forward to the next scheduling point (cut-off by the parameter 'maximal retry interval' `MAX_INTERVAL`). Otherwise, the simulator will provision resources for that job as usual. If there are not sufficient resources to run the job now, the simulator will wait until enough resources are released. During waiting, if backfilling is enabled, it may schedule other waiting jobs if that does not affect execution of the currently scheduled job. There is another parameter (`MAX_REJECTION_TIMES`) to control the maximal number of rejections that one job can receive. If a job has been rejected too many times, SchedInspector will not reject it again.

Note, in our simulator, we differentiate two job execution times carefully. We use the *actual execution time* (exe_j) to calculate the job's finishing time. We use the *estimated execution time* (est_j) to conduct scheduling for schedulers and SchedInspector. Unless explicitly stated, 'job execution time' always refers to the estimated execution time.

3.3 Feature Building

As discussed earlier, SchedInspector inspects scheduling decisions based on the runtime environmental factors. Hence, all these scheduling related contexts should form the state for RL agent to observe. However, such a raw state does not work efficiently for RL training. First, the raw state is large and contains features that are irrelevant to training but generate overhead, such as job IDs. Second, many state features are correlated and can be better modeled if manually pre-processed. In SchedInspector, we manually build following features, designed for the inspection tasks, to maximize the training efficiency.

Scheduled job. SchedInspector needs to observe the the scheduled job waiting for inspection. Out of all job features, we select three of them: *job waiting time* ($wait_j$, the duration the job has been waiting in the queue); *job execution time* (est_j, the estimated execution time of the job); and *job requested computing nodes* (res_j, the number of nodes requested by the job). We eliminated other job features such as *job id* to improve the training efficiency.

Rejected times. We track how many times each job has been rejected to avoid rejecting a job too many times. Each time, SchedInspector will compare the job's current rejected times with the threshold `MAX_REJECTION_TIMES` to determine whether further rejections can be granted or not.

Queue delays. Rejecting scheduling decision leaves resources in idle. It delays not only the execution of currently scheduled job, but also that of the waiting jobs in the queue. Intuitively, the overhead or penalty increases when more jobs are waiting in the queue. This value impacts the inspection and rejections significantly and should be observed by the agent.

But, simply addressing the number of waiting jobs is not accurate as the cost is relevant to the metrics used to evaluate the schedulers. For instance, if the metrics is *bsld* (average bounded slowdown), then introducing a Δt idle is expected to increase each waiting job's bsld by roughly $\Delta t/max(exe_j, 10)$. If the metrics is *wait* (average waiting time), then the increment for each waiting job will be directly Δt. In SchedInspector, we iterate all of the waiting jobs, calculate their expected delays according to the given performance metrics, and add them together as the value of *queue delays*. We use it as a key factor to help RL agent understand the cost of its rejection decision.

Runnable and ***Cluster availability***. These two features are relevant as they reflect the available resources in the cluster and whether the currently top-priority job can run or not. The cluster availability is calculated as the ratio of free computing nodes (n_{free}) and total computing nodes (n_{total}). Runnable is calculated by comparing the required resources of the job (res_j) and the current available computing nodes (n_{free}) in the cluster. If res_j is smaller, runnable value is 1 meaning the job can run immediately; otherwise its value is 0.

Backfilling Contributions. If *Runnable* is 0, the cluster does not have enough resources to run the selected job. It needs to wait until more resources are released. During waiting, the cluster can schedule other waiting jobs if backfilling is enabled. As backfilling changes the scheduling behaviors significantly, the RL agent needs to know whether it is enabled or not. We use this feature to represent it. If backfilling is not enabled, this feature would be 0. If it is enabled, we scan the waiting jobs and calculate the number of waiting jobs that can be backfilled as the final value of this feature. We use it as a factor to help RL agent work with backfilling.

To summarize, we manually build features for SchedInspector for better training accuracy and efficiency. Many of these manual features are aggregated from multiple jobs instead of their raw individual values (e.g., *queue delays* and *backfilling contributions*). We intentionally do so to prevent SchedInspector from pursuing the subtle optimal that can only be identified by predicting the future and exhaustively examining each individual job and their possible combinations. These opportunities are not stable, hence not our focus in SchedInspector. Our evaluation results in Section §4 also confirms that these manually built features not only accelerate the training, but also help SchedInspector perform consistently well towards unseen job traces.

3.4 Reward Function

In RL, reward is the feedback from the environment to the agent. The goal of RL training is to maximize the cumulative rewards after a sequence of actions. In SchedInspector, the reward should be relevant to the performance metrics that the job schedulers try to optimize for, such as minimizing average bounded job slowdown or average job waiting time.

Intuitively, the reward can simply be the difference between values of the performance metrics *with* and *without* SchedInspector after scheduling the same sequence of jobs. For instance, if the metric is minimizing *bsld* (*average bounded slowdown*), then the reward can be $bsld_{orig} - bsld_{inspect}$. The RL agent will maximize the reward by minimizing $bsld_{inspect}$. If the performance metric is minimizing *average waiting time* ($wait$), then the reward can be $wait_{orig} - wait_{inspect}$. We call this direct subtraction *Native reward*.

The major issue of native reward, however, is the high variances of some performance metrics, which may decrease the training accuracy. For example, the average bounded slowdown (*bsld*) of scheduling 256 continuous jobs sampled from a real-world job trace (SDSC-SP2) ranges from 1 to 2414. The improvements in a *bsld=2414* job sequence can be easily larger than the improvements in a *bsld=2* job sequence, which may confuse the RL agent during training. Such a bias needs to be eliminated. One bias free reward could be just counting how many times SchedInspector wins the base scheduler. The reward function then can be defined as $count(bsld_{inspect} < bsld_{orig})$. This reward, namely *Win/Loss reward*, is easy to calculate and will not be affected by the variances of the metric values. However, it treats all improvements the same, hence does not reward the big-gain actions and can not guide the RL agent to identify high-gain opportunities limiting its training performance.

In SchedInspector, we design a new reward function called *Percentage Reward* to address the issues of previous two reward functions. For instance, if the performance metric is minimizing *bsld*, then the reward is defined as $(bsld_{orig} - bsld_{inspect})/bsld_{orig}$. Compared to the previous two rewards, this function eliminates the high variance of $bsld_{orig} - bsld_{inspect}$ by dividing it by $bsld_{orig}$; it also awards the big-gain actions as the percentage could be higher in that case. In §4, we compared different reward functions and show the advantage of the *percentage reward*.

4 EVALUATION

This section evaluates the performance of SchedInspector. In particular, we focus on answering the following questions:

- Can SchedInspector learn how to inspect scheduling decisions and improve the performance of the base scheduling policies? If yes, how efficiently does it learn?
- How much SchedInspector could improve the job execution performance upon different scheduling policies on different workloads? When does SchedInspector not perform well and why? How does backfilling impact the results? How would SchedInspector impact the system utilization?
- Does SchedInspector still work in more realistic scheduling settings? Are the overheads of SchedInspector acceptable in production system?

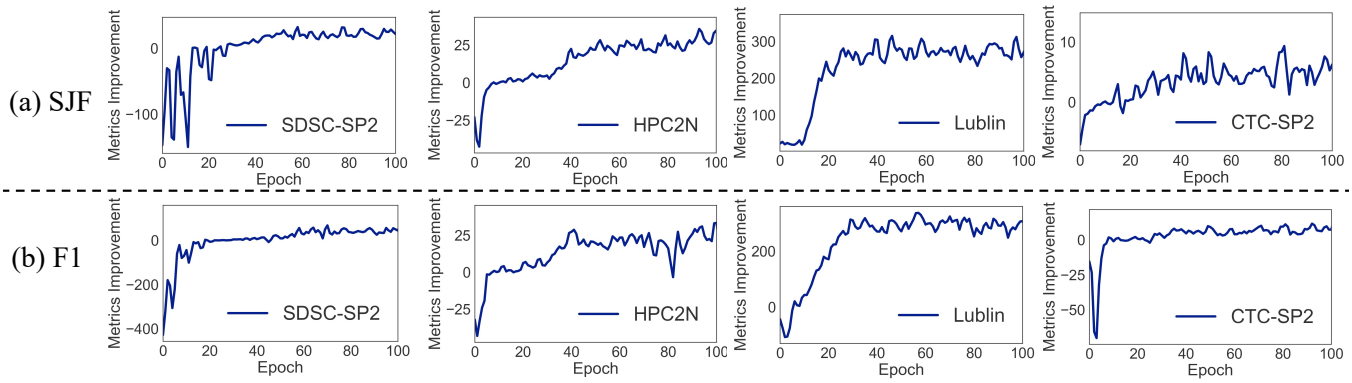

Figure 4: Training curves of SchedInspector on four job traces using two schedulers. ***The x-axis shows the training epoch, the y-axis shows the metrics improvements on the selected metrics (bsld). Larger than 0 means SchedInspector performs better than the base schedulers.***

Table 2: List of job traces in use.

Name	*cluster size*	*interval* (sec)	est_j (sec)	res_j
CTC-SP2	338	379	11277	11
SDSC-SP2	128	1055	6687	11
HPC2N	240	538	17024	6
Lublin	256	771	4862	22

Table 3: List of base batch job scheduling policies.

Abbr.	*Full Name*	*Priority Setting*
FCFS	First Come First Served	$max(wait_j)$
LCFS	Last Come First Served	$min(wait_j)$
SJF	Shortest Job First	$min(est_j)$
SAF	Smallest estimated Area First	$min(est_j * res_j)$
SRF	Smallest estimated Ratio First	$min(est_j / res_j)$
F1	Carastan-Santos et. al [9]	$min(log_{10}(est_j) * res_j + 870 * log_{10}(s_j))$

4.1 Implementation and Configuration

We implement SchedInspector based on OpenAI SpinningUp library [2] using Tensorflow [1]. SchedInspector uses Proximal Policy Optimization (PPO) algorithm, a state-of-the-art policy-gradient RL algorithm, to train [28].

SchedInspector contains several hyper parameters. Two are relevant to the inspection itself. `MAX_INTERVAL` means the maximal waiting time for the base schedulers to try again after being rejected. We set it to be 600 seconds or 10 minutes to avoid idling resources for too long. `MAX_REJECTION_TIMES` means the maximal number of rejections that one job can receive. We empirically set it to be 72. In this way, a job might not get scheduled in $600 * 72$ seconds or 12 hours, which is a relatively long range as we want to promote SchedInspector to explore. The policy-gradient reinforcement learning algorithm also contains several hyper parameters. Its batch size is 100, which means the agent collects 100 trajectories before updating its model. Each time the agent updates the model, we call it an *epoch*. Each trajectory contains 128 sequential jobs selected from a random start index of the job trace. The learning rate of the network updating is 10^{-3}. More details about the implementation and configuration can be found in our code repo[1].

4.2 Evaluation Setup

We evaluate SchedInspector using job traces from Parallel Workloads Archive [12]. Table 2 shows the job traces in use and their key features, such as the total number of processors in the cluster (*cluster size*), average job arrival interval (*interval*), average estimated execution runtime (est_j), and average requested processors (res_j). To be generic, we include both synthetic (Lublin in the bottom part) and real-world (others in the top part) traces. We select these job traces for two reasons. First, they are commonly used in relevant research [9, 39], hence popular for evaluating schedulers. Second, from their key features reported in Table 2, we can observe these job traces are diverse. We expect SchedInspector works well with various job traces.

In addition to job traces, we also evaluated SchedInspector on different job scheduling policies. Table 3 shows these scheduling policies and their heuristic priority functions. Here, parameter s_j is job submission time; $wait_j$ is job waiting time; est_j is job estimated runtime; res_j is job requested resources; $A_t = est_j * res_j$ indicates the estimated area, and $R_t = est_j / res_j$ indicates estimated ratio. More details about these scheduling policies can be found in [9, 19]. We do cover a wide range of representative scheduling policies in Table 3. FCFS, LCFS, and SJF are policies that focus on one job attribute; SAF and SRF are policies built on two job attributes; F1 [9] is the state-of-the-art heuristic scheduling policy which leverages machine learning to build the non-linear regression of multiple job attributes. We will exam how SchedInspector works with all these base scheduling policies.

4.3 Evaluations on SchedInspector RL Designs

In this set of experiments, we first evaluate 1) whether the RL-based SchedInspector can successfully learn the inspection and rejection strategy to gain better job execution performance than the base scheduling policies; 2) how efficient the learning is; and 3) how our new RL designs impact the learning efficiency.

To answer these questions, we directly show the training curves of SchedInspector on four job traces listed in Table 2 using two

[1] https://github.com/DIR-LAB/SchedInspector

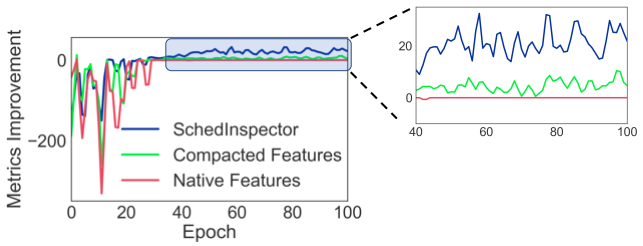

Figure 5: The comparison of the training curves of SchedInspector with different feature building mechanisms. *y-axis shows the improvements of SchedInspector over the base scheduling policy on bsld. Larger is better.*

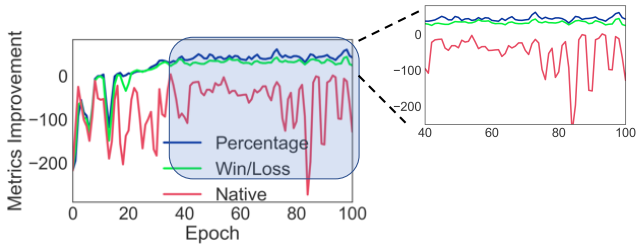

Figure 6: The comparison of the training curves of SchedInspector with different reward functions.

different base scheduling policies: SJF and F1 [9]. Both policies optimize the average bounded job slowdown (*bsld*). F1 actually achieves the state-of-the-art performance in *bsld*. Hence, we used *bsld* as the performance metric and evaluated whether SchedInspector can further improve it. Figure 4 presents the results. We discuss how SchedInspector performs with other job scheduling policies and job execution performance metrics in details in §4.4.

The results in Figure 4 show SchedInspector starts from performing worse (larger *bsld*) than the base scheduling policies, but improves quickly and finally achieves better *bsld* than the base policies do (both SJF and F1) on all tested job traces (converged to a value larger than 0). It is interesting to see that SchedInspector can further reduce *bsld* value of F1 in such a significant way (e.g., 40% better in SDSC-SP2 trace and 95% better in Lublin trace) since F1 is already highly optimized for minimizing *bsld* [9]. This result shows the effectiveness of SchedInspector during training.

In addition, from these results we observed SchedInspector delivers an efficient learning process (mostly converged within the first 40 epoch). We believe that the Feature Building mechanism and Reward Function contribute to the efficiency. To prove it, we further conducted micro-benchmark evaluations towards the two new designs in the next two subsections.

4.3.1 Impacts of Feature Building. To show the impacts of new *Feature Building* mechanism, we compare the features built in SchedInspector with two naive ways to build features: *native features* and *compacted features*. The *native features* directly use the whole environmental state as the inputs. Such a strategy is commonly used in other RL-based system optimization work [21, 39] as they expect the deep neural network to learn the useful features out of raw inputs automatically. The *compacted features* downselects the features. It includes only the current job and the cluster state, and ignores the *queue delay* and *backfilling contributions*. We use this mechanism to show how SchedInspector would perform if it misses the important aggregated features.

We compared the training curves of these three cases in Figure 5. To save the space, we only report the result of running SJF base scheduling on SDSC-SP2 trace using metrics *bsld*. Other evaluations show similar trends. From the figure we observed that across three feature mechanisms, the manually built features outperform the others clearly. The improvements over base SJF converge to 8.7 using *compacted feature*. While at the same time, *SchedInspector Feature* converges to 25.1, which is 2.9x better. The *native feature* performs the worst and can not converge to a positive value. After checking its decisions in more details, we found that the RL agent learns simply not to reject any scheduling at all. We believe this is because the raw state often leads to subtle and unstable corner cases and confuses the RL agent. Such a result confirms that our Feature Building mechanism prevents SchedInspector from chasing corner cases and assures its high training accuracy.

4.3.2 Impacts of Reward Function. We further compare the performance of SchedInspector using different *Reward* functions. Specifically, we compared the *native reward*, *win/loss reward*, and the *percentage reward* in the same scheduling scenario as discussed earlier (i.e., running SJF base scheduling on SDSC-SP2 trace using metrics *bsld*). The results are shown in Figure 6.

Here, the y-axis is the direct difference between $bsld_{orig}$ and $bsld_{inspect}$. Intuitively, such a y-axis should favor the *native reward*, because it is exactly what native reward optimizes for. However, the results are counter-intuitive: among the three reward functions, the *percentage reward* actually performs the best. This reflects the benefits of percentage reward as it stabilizes the highly variant reward values as well as captures the big gains. While the highly variant reward values cause problems in the native reward function.

4.4 Evaluation on SchedInspector Generality

This section discusses how SchedInspector works with different base scheduling policies on different job traces and optimize towards different job execution metrics. We also show its performance with backfilling enabled and disabled cases. Beyond the training curves, we further report the performance of SchedInspector when tested in scheduling the actual job sequences sampled from job traces. To avoid over-fitting, for each trace we use the first 20% for training and the remaining 80% of the data for testing.

4.4.1 Working with Various Scheduling Policies. To understand how SchedInspector works with various base job scheduling policies, we trained SchedInspector based on the scheduling heuristics listed in Table 3 and checked their performance. To save spaces, we reported the results of using SDSC-SP2 as job trace and *bsld* as job execution performance metrics. Results on other job traces show similar trend.

The training of SchedInspector with different job scheduling policies are shown in Figure 7. We do not include the results on SJF and F1 since they were already reported in Figure 4. This time, in addition to show how the training converges on the performance

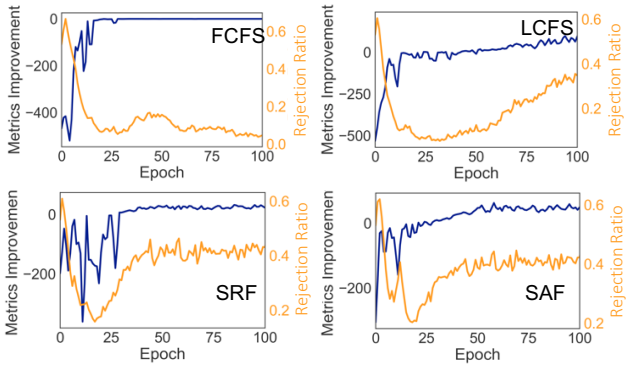

Figure 7: SchedInspector training with different base job scheduling policies. *Blue curve is the bsld improvements using the left y-axis; orange curve is the rejection ratio using the right y-axis.*

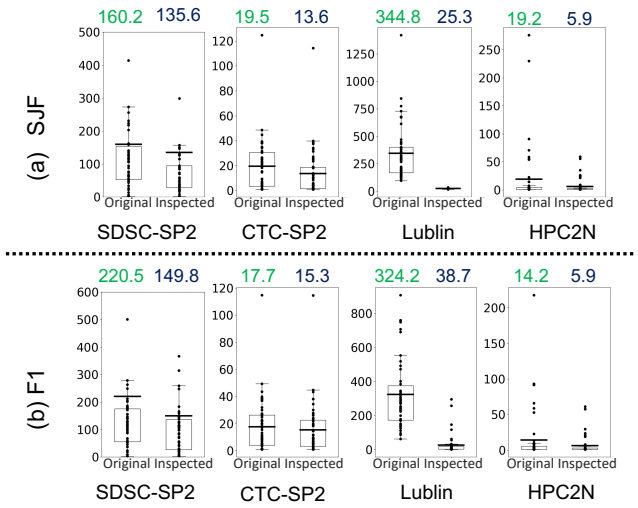

Figure 8: The scheduling performance of SchedInspector and base scheduling policies on four different job traces. *y-axis is the bsld value. The averages are shown on the top of each bar. Smaller is better.*

metrics, we show another important indicator about the training: how the *Rejection Ratio* changes. Rejection Ratio is how many times the rejections are made v.s how many times the inspection happens. The right y-axis measures the ratio, ranging from 0 to 1.

From these results, we can observe that among these job schedulers, FCFS is the only one did not converge to positive improvement, which indicates SchedInspector introduces no benefits to it. When taking a closer look into its Rejection Ratio, we can notice its value slowly reduces to 5% starting from around 50%. Comparatively, on other scheduling policies (LCFS, SRF, SAF), SchedInspector can converge to 144.9, 52.9, and 34.5 metrics improvements (*bsld*); and their Rejection Ratios converge to 39.1%, 41.0%, and 48.2% respectively. These results show although SchedInspector works well with many scheduling policies, not all of them benefit from SchedInspector.

In fact, it is expected that SchedInspector does not benefit all scheduling policies. The key is whether rejecting current job can lead to a different job being scheduled in the future. If nothing changes after some idle time, then the rejection becomes a pure waste. Since FCFS always prioritizes the oldest job, any future job will not impact its decision. Inspecting and rejecting FCFS then is a pure waste of time and resources. From Figure 7, we can observe SchedInspector converges to a very low rejection rate (5%) in FCFS. The converged rejection rate is not exactly 0 because of the explorations in the reinforcement learning as well as the low cost of certain rejections. For example, rejecting a job that needs to wait for resources does not impact the performance. Comparatively, F1 and SJF try to prioritize jobs for the minimal *bsld*. The future jobs added may fit better to that goal and hence change the next scheduling decision. So, SchedInspector has the chance to improve F1 and SJF. Interestingly, we did not embed such knowledge into SchedInspector design. But it learns that automatically. A low converged Rejection Ratio (less than 10%) is a strong signal for SchedInspector to disable any delaying attempts for that scheduling policy.

4.4.2 Working with Different Job Traces. Figure 4 already presents the good training convergence of SchedInspector toward the performance metric *bsld* on various job traces. In this section, we further evaluated how the trained SchedInspector model performs when applied to schedule actual job sequences. We focus on the performance based on SJF and F1 base scheduling policies and *bsld* job execution performance metrics due to space limitation. We will discuss other job execution performance metrics in later sub-sections.

Table 4: The performance of SchedInspector (*bsld* as the metrics) on different job trace.

	Base→Y	'SDSC-SP2'→Y	Y→Y
SDSC-SP2	149.5	130.75	130.75
CTC-SP2	13.36	10.79	10.1
Lublin	333.19	320.39	27.97
HPC2N	8.26	4.39	3.27

The results are reported in Figure 8. In these experiments, for each trace, we randomly sampled 50 different job sequences (each contains continuous 256 jobs) from the testing dataset. We scheduled them using both the base schedulers (SJF, F1) and SchedInspector enabled counterparts, and calculated their metrics (*bsld*). We plot all of the 50 results and their box-and-whisker with the averages in Figure 8. From these results, we can observe SchedInspector indeed improves the base scheduling policies across different job traces: the metrics *bsld* is becoming better (smaller) from 13.6% (F1 running on CTC-SP2 case) to 91.6% (SJF running on Lublin case).

The previous results (Figure 8) show the performance of SchedInspector when trained from and applied to the same job trace (using separate training and testing datasets). The results suggest the SchedInspector model is stable across training and testing datasets. Next, we further evaluated the stability of SchedInspector when trained on a trace-X but applied to a totally different trace-Y. Due to space limit, we only report the results of using SDSC-SP2 as the trace-X, SJF as the base scheduling policy, and *bsld* as the metrics. Other scheduling policies share the similar trends.

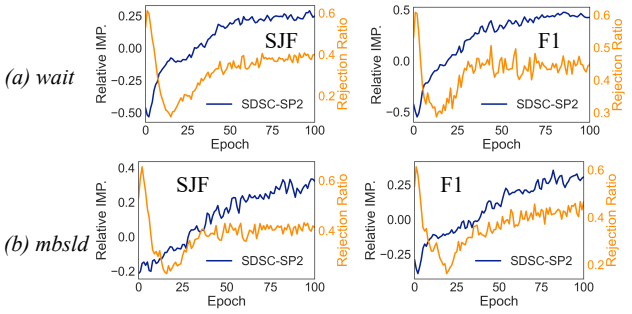

Figure 9: SchedInspector training with different job execution performance metrics. *Blue curve is the relative improvement using the left y-axis; orange curve is the rejection ratio using the right y-axis.*

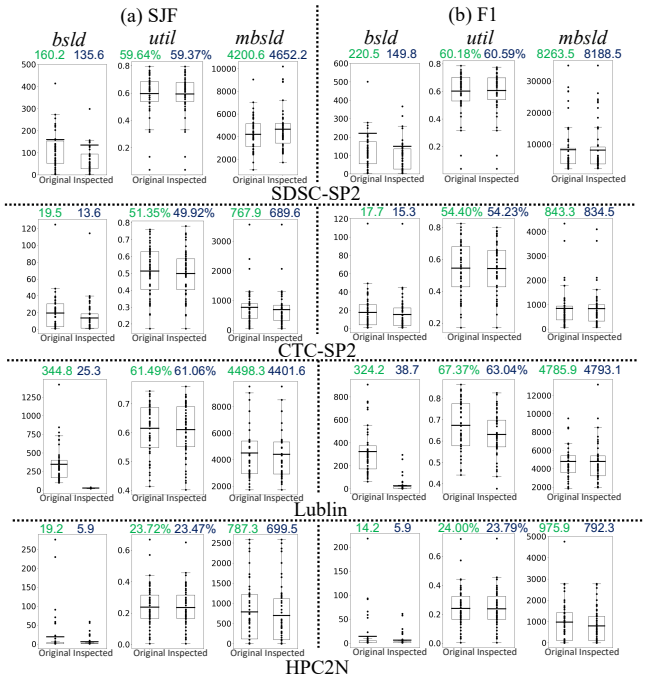

Figure 10: The performance of SchedInspector on different metrics using SJF and F1 as base scheduling policy on different job traces. *lower are better for* $bsld$ *and* $mbsld$*; higher are better for* $util$*.*

Table 4 summarizes the results. Each row represents a job trace-Y being scheduled under one of the three scheduling scenarios (columns). In particular, the first column (Base→Y) shows the performance of using base scheduling policy (SJF) to schedule each job trace Y. The second column ('SDSC-SP2'→Y) shows the performance of applying SchedInspector trained using SDSC-SP2 (trace-X) to each job trace Y. The third column (Y→Y) shows the results of applying SchedInspector trained using each job trace Y to the same trace Y. Each time, we scheduled the same 50 randomly sampled job sequences from each job trace and report their average *bsld* for comparison. From the table we can observe that, although 'SDSC-SP2'→Y performs not as good as Y→Y does, it still outperforms the base scheduler, suggesting a stable and positive impact of SchedInspector across workloads.

4.4.3 Working with Different Job Execution Metrics. In the previous evaluations, we mainly used *bsld* (average bounded job slowdown) as the job execution performance metric. In this section, we further examine how SchedInspector performs towards different job execution metrics. In particular, we conducted evaluations towards two additional job execution performance metrics, each addressing an aspect of the system:

- **average waiting time (*wait*)**: the average duration between the job's submission and its start time. It does not consider job length in its calculation.
- **maximal bounded job slowdown (*mbsld*)**: the maximal *bsld* of a job sequence instead of the average. It emphasizes on the fairness and effectively avoids starving long jobs.

Figure 9 reports the training curves of SchedInspector towards these two performance metrics. Due to the space limitation, we only present the results on job trace SDSC-SP2 with SJF and F1 serving as the base scheduling policies due to their representativeness. The results in Figure 9 show that, for both *wait* and *mbsld*, SchedInspector starts performing not as well as the base scheduling policy does. However, it learns fast and converges stably, and achieve 25% to 50% better performance than the base scheduler does at the end. These results show SchedInspector does perform well and stably across different job execution metrics, such as *wait* and *mbsld*.

4.4.4 Trade-off Among Different Metrics. In real world, more performance metrics in addition to job execution metrics may be important to end users. They may further care about the trade-off among different metrics when a scheduling policy is applied. In this section, we show how SchedInspector performs if gets trained toward one job execution performance metric but evaluated on other metrics. Specifically, we trained SchedInspector towards *bsld* and evaluated it towards *util* and *mbsld*. Here *util* indicates the system resource utilization, calculated as the resources used for executing jobs v.s all resources available. We examine *util* to show whether SchedInspector will significantly affect the resource utilization because of its actively rejecting and delaying scheduled jobs. We also show the performance of maximal bounded job slowdown (*mbsld*) to examine whether SchedInspector will blindly push back long jobs and starve them to get better *bsld*.

Figure 10 presents the results. In these experiments, for each trace, we randomly sampled 50 different job sequences (each contains continuous 256 jobs) only from the testing dataset. We scheduled them using both the base schedulers (SJF, F1) and SchedInspector enabled counterparts. Each time, we calculated all three performance metrics: *bsld*, *util*, and *mbsld* and plot all of the 50 results and their box-and-whisker with the averages in Figure 10. From these results, we can observe that the SchedInspector trained on *bsld* still performs well towards *mbsld*, indicating that SchedInspector does not blindly push back long jobs to gain better *bsld*. We also observe SchedInspector does introduce slight overhead

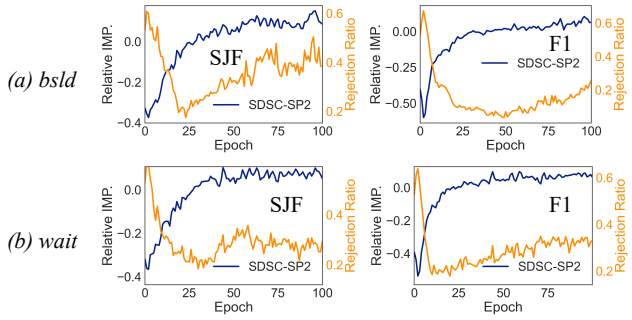

Figure 11: The training curves of SchedInspector with backfilling. ***y-axis shows the relative improvement and rejection ratio.***

on system utilization (less than 1% in most of the cases), showing the rejections introduced by SchedInspector do not break the system utilization. We will more systematically discuss SchedInspector's impacts on system utilization with backfilling counted in Section 4.4.6.

4.4.5 Working with Backfilling. Previously, we have shown SchedInspector improves various base job scheduling policies with backfilling disabled. It is interesting to see whether SchedInspector can still improve the performance of job scheduling policies with backfilling enabled. Figure 11 reports the training curves of SchedInspector with backfilling enabled using *bsld* and *wait* as the job execution performance metrics. Due to the space limitation, we only present the results of SDSC-SP2 trace on SJF and F1 scheduling policies. Other cases share the similar patterns. Similar to the backfilling-disabled cases, SchedInspector can effectively learn and achieve better performance than the base scheduler. But we do notice that the converged metrics improvements are smaller. This is expected as backfilling has improved the schedulers significantly, leaving limited optimization space for SchedInspector. Even though, we can still see that SchedInspector converges to about 10% improvements.

4.4.6 Impacts on System Utilization. This subsection addresses the concerns about whether SchedInspector and its rejection decisions hurt system utilization after introducing idles. In this experiment, we randomly sampled 50 job sequences (each contains continuous 256 jobs) from different job traces, scheduled them using the base SJF and F1 schedulers (*BASE*) and their SchedInspector enabled counterparts (*INSP*), then calculate their system utilization. We consider both with and without backfilling cases. The results are in Table 5. We can observe that, in all these cases, SchedInspector introduces barely noticeable reduction ($\Delta \approx 1\%$) on system utilization except for (Lublin, F1) case, which has a 4.3% difference. But, relative to Figure 8, we know for this exact case, SchedInspector also brings 88% improvement on average bounded job slowdown, overshadowing its utilization reduction.

4.5 SchedInspector in Realistic Settings

So far, we have examined the effectiveness and generality of SchedInspector based on various heuristic scheduling policies. But, real-world batch job schedulers are more complicated and need to consider fair sharing among users, project quotas, and queue priorities

Table 5: System utilization with/without SchedInspector.

	SJF			F1		
	BASE	INSP	Δ	BASE	INSP	Δ
Scheduling ***without*** *Backfilling*						
SDSC-SP2	59.64%	59.37%	**-0.27%**	60.18%	60.59%	**+0.41%**
CTC-SP2	51.35%	49.92%	**-1.43%**	54.40%	54.23%	**-0.17%**
Lublin	61.49%	61.06%	**-0.43%**	67.37%	63.04%	**-4.33%**
HPC2N	23.72%	23.47%	**-0.25%**	24.00%	23.79%	**-0.21%**
Scheduling ***with*** *Backfilling*						
SDSC-SP2	78.45%	78.37%	**-0.08%**	76.71%	76.93%	**+0.22%**
CTC-SP2	74.98%	74.89%	**-0.09%**	75.47%	76.05%	**+0.58%**
Lublin	79.38%	77.71%	**-1.67%**	80.38%	78.08%	**-2.30%**
HPC2N	56.81%	57.10%	**+0.29%**	57.11%	56.57%	**-0.54%**

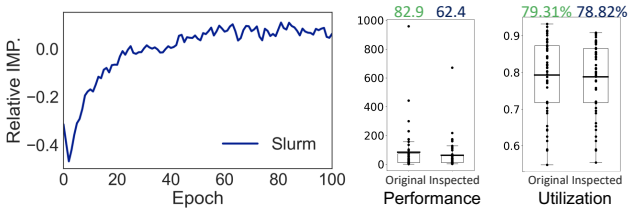

Figure 12: The performance of SchedInspector working with Slurm. ***y-axis on the right two charts are*** *bsld* ***and util percentage.***

in their policy. It is then interesting to see whether SchedInspector still works with them. In this evaluation, we used Slurm multifactor priority scheduler with backfilling as the base scheduling policy. The job priority is calculated as follow [37]:

$$\begin{aligned} Job\_Priority = &(weight_{age}) * (age\_factor) + \\ &(weight_{fairshare}) * (fairshare\_factor) + \\ &(weight_{jattr}) * (job\_attribute\_factor) + \\ &(weight_{partition}) * (partition\_factor) + \ldots \end{aligned}$$

Here, age_factor is job waiting time (normalized using 7 days). The $fairshare\_factor$ calculates the offset between a user's actual CPU usage and her assigned share. We used the normal model to do the calculation [7] and use a user's actual CPU usage as her assigned shares. We used job requested execution time as $job\_attribute\_factor$. The $partition\_factor$ is the priority of each queue in the system and is not given in our job trace file. We calculated it by counting the CPU usages of each queue across the whole trace and used such an actual share to estimate the queue priority. Although a real system may consider more factors, such as QoS_factor. We eliminated them as existing data traces do not capture such information. For the weights, we assigned all $weight_{(*)}$ as 1000 to consider all of these factors equally. We used SDSC-SP2 to conduct this experiment because it contains the needed user and queue information. We used *bsld* as the job execution performance metric to optimize.

Figure 12 reports the results. We report the actual training curve of SchedInspector, *bsld* performance and *util* performance when the trained model is applied to schedule 50 randomly sampled job sequences from testing dataset. Since backfilling is by default

enabled in Slurm, we only report the backfilling results. Here, we can observe file SchedInspector still learns how to delay Slurm scheduler in this case. It performs 24.7% better on *bsld* (62.4 vs. 82.9, smaller is better) and only introduces 0.49% utilization reduction.

4.6 SchedInspector Computational Cost

The computational cost consists of *training* and *inference* cost. SchedInspector is trained based on the simulated environment, so it can be trained in an offline environment using the historical job traces collected from the cluster. Our previous evaluations show SchedInspector still works well on partially seen or even unseen job traces. So, it is reasonable to re-train the SchedInspector every week or month instead of keeping up with the job traces in real time. This means SchedInspector is not sensitive to the training time. Based on the current implementation, the training time for SchedInspector is about 35 minutes, which is far smaller than the necessary model updating intervals (weeks or months). The inference cost is more important as each job scheduling decision made by the scheduler will need to go through SchedInspector. Because of the feature-building mechanism and the simple 3-Layer MLP neural networks, we are able to control the inference cost to 0.7ms, which is negligible in batch job scheduling.

5 WHAT SCHEDINSPECTOR LEARNS

In this section, we discusses what SchedInspector actually learns. Because SchedInspector takes a statistical approach, we conducted statistical analysis on the learned model. To do so, we first trained a model based on the following combination: [SJF, *bsld*, SDSC-SP2], then used the trained model to schedule the whole SDSC-SP2 job trace from beginning to the end. During scheduling, we recorded each inspection decision (reject or not) and its corresponding input state features. In this experiment, we totally collected 24,044,629 *Total Samples*, each representing an inspection. Among them, 7,351,608 samples yield rejection (≈30%), which are called *Rejected Samples*. We then analyze how the inputs features impact rejection decisions to reveal what SchedInspector learns.

Specifically, we plot the cumulative distribution function (CDF) of rejected samples and total samples for different features. Figure 13 shows the results. To interpret these figures, we focus on the gradient of the curves during certain range of x-axis. For instance, in 'Waiting Time' figure, the curve of rejected samples increases much faster at the beginning of x-axis ([0, 0.3]), which means SchedInspector delays more often when jobs have smaller waiting time. Similarly, in 'Job Execution Time' figure, the curve of rejected samples increases faster in the larger x-axis, which means SchedInspector rejects more jobs with longer execution time. Note that, since this model is trained without backfilling, we ignore the *Backfilling Contributions* feature; also we did not plot *Runnable* as its value is either 0 or 1.

From these results, we can briefly summarize what SchedInspector learns. First, SchedInspector tends to delay the jobs with shorter waiting times, longer execution times, and higher resource requirements. This is intuitive as delaying short-waiting jobs costs less and delaying high-demanding or long-running jobs could lead to more gains on *bsld*. Second, when looking into the scheduling environment (i.e., free nodes, queue delays), we find both of them have

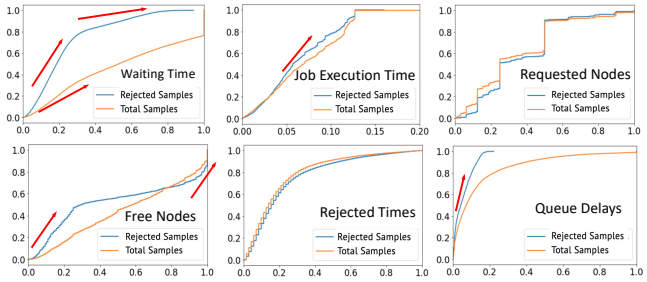

Figure 13: The CDF distribution of input features. Rejected Samples vs. Total Samples. *x-axis shows the normalized value of the feature.*

profound impacts on the rejection decisions. For 'Free Nodes', it is interesting to see 'having many free nodes' and 'having few free nodes' both lead to more rejections than moderately loaded case. This seems counter-intuitive but actually makes sense: when the cluster is busy, immediately scheduling an unfit job may saturate the cluster, so rejecting it leads to big gains; when the cluster is mostly idle, rejecting a job may not lead to big gains, but it means less cost because the rejection will affect less waiting jobs. This also means SchedInspector may be more useful in real-world HPC, as they are often in full or idle states. Also we noticed that the hard cap for 'Queue Delays': it has a maximal value 0.22. When the queue delays is larger than that, SchedInspector will never delay job anymore, regardless of the other factors.

6 RELATED WORK

In HPC, batch job scheduling has been a long-standing research topic. Various batch schedulers have been proposed and studied [3–5, 9, 16, 23, 32, 35]. In particular, they determine the scheduling priority by heuristically weighting different job features, from focusing on one feature to leveraging sophisticated methods such as linear programming [5, 13], non-linear algorithms [9, 15, 29], neural networks [3, 4], or even reinforcement learning [22]. Compared to the previous studies, SchedInspector is fundamentally different. It inspects and rejects existing job scheduling policies' decisions and improves their performance by taking runtime factors into considerations. Recently, there are batch job scheduling policies that consider both job features and runtime factors together via machine learning, such as RLScheduler [39], DRAS [10], DSBH [34]. However, RLScheduler-like policies often change the original scheduling policy disruptively that system administrators are often reluctant to do. Even worse, since the scheduling decisions are based on a mix of job features and runtime factors, the outcomes sometimes become arbitrary and hard to interpret and explain to end users, further hindering its application. One key advantage of SchedInspector is it works with existing scheduling policies without changing them. Its rejection decisions are also separate from the decisions made by the base scheduling policies, hence can be provided individually to end users.

Some non-work-conserving schedulers, sharing the similar principles of rejecting or delaying a ready task as SchedInspector does. They are mostly designed for the environment where contentions may occur or costs may be higher if a task has to run now, such

as disk scheduling [26, 36], real-time simultaneous multithreading (SMT) processors [11, 14, 27], tasks with locality [38], batch mode scheduling [35], rent-based cloud with dynamic price [6]. Delaying scheduling becomes intuitive in these cases. Compared to these studies, the rationale behind delaying the scheduling is quite different from SchedInspector and accordingly leads to different solutions and results.

7 CONCLUSION AND FUTURE WORK

This study presents SchedInspector, a new reinforcement learning based scheduling inspector to incorporate runtime information into batch job scheduling. Our results show SchedInspector is capable to learn high-quality inspection and rejection policies towards various job scheduling policies, including the state-of-the-art one, to gain better job execution performance, such as minimizing average bounded job slowdown, job waiting time, and maximal bounded job slowdown. The performance improvements are generic across different workloads and scheduling scenarios. We also extensively show and discuss the impacts of SchedInspector on system utilization and the scheduling policies that SchedInspector does not work well with. We demonstrate what SchedInspector learns and explain its intuition using an example. In the future, we plan to investigate further on more kinds of job workloads, especially the machine learning workloads; incorporate SchedInspector with intelligent scheduling policies, such as RLScheduler [39]; and integrate SchedInspector into Slurm [37] in real-world HPC clusters.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable feedback. This work is supported by NSF grants CCF-1910727 and CNS-1817094. This work used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, and Google Brain. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[2] Joshua Achiam. 2018. Spinning Up in Deep Reinforcement Learning.

[3] Anurag Agarwal, Selcuk Colak, Varghese S Jacob, and Hasan Pirkul. 2006. Heuristics and augmented neural networks for task scheduling with non-identical machines. *European Journal of Operational Research (EJOR'06)*.

[4] Derya Eren Akyol and G Mirac Bayhan. 2007. A review on evolution of production scheduling with neural networks. *Computers & Industrial Engineering (CAIE'07)*.

[5] Hadil Al-Daoud, Issam Al-Azzoni, and Douglas G Down. 2012. Power-aware linear programming based scheduling for heterogeneous computer clusters. *Future Generation Computer Systems (FGCS'12)*.

[6] Pradeep Ambati, Noman Bashir, David Irwin, and Prashant Shenoy. 2020. Waiting game: optimally provisioning fixed resources for cloud-enabled schedulers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*.

[7] David Bigagli. 2014. Slurm Workload Manager Introductory User Training. https://cug.org/proceedings/cug2014_proceedings/includes/files/tut102.pdf.

[8] Hervé Bourlard and Yves Kamp. 1988. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics (Biol. Cybern.'88)*.

[9] Danilo Carastan-Santos and Raphael Y. de Camargo. 2017. Obtaining dynamic scheduling policies with simulation and machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*.

[10] Yuping Fan, Zhiling Lan, Taylor Childers, Paul Rich, William Allcock, and Michael E Papka. 2021. Deep reinforcement agent for scheduling in HPC. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'21)*.

[11] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. 2006. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA (WIOSCA'06)*.

[12] Dror G. Feitelson, Dan Tsafrir, and David Krakov. 2014. Experience with using the Parallel Workloads Archive. *Journal of Parallel and Distributed Computing (JPDC'14)*.

[13] Christodoulos A Floudas and Xiaoxia Lin. 2005. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research (Ann. Oper. Res.'05)*.

[14] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. 2008. Work-conserving optimal real-time scheduling on multiprocessors. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems (ECRTS'08)*.

[15] Edwin SH Hou, Nirwan Ansari, and Hong Ren. 1994. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed systems (TPDS'94)*.

[16] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. 2011. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed systems (TPDS'11)*.

[17] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research (JAIR'96)*.

[18] Vijay R Konda and John N Tsitsiklis. 2000. Actor-critic algorithms. In *Advances in neural information processing systems (NeurIPS'99)*.

[19] Arnaud Legrand, Denis Trustram, and Salah Zrigui. 2019. Adapting batch scheduling to workload characteristics: what can we expect from online learning?. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'19)*.

[20] Yan Li, Kenneth Chang, Oceane Bel, Ethan L Miller, and Darrell DE Long. 2017. CAPES: unsupervised storage performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*.

[21] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets'16)*.

[22] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication (SIGCOMM'19)*.

[23] Ahuva W Mu and Dror G Feitelson. 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with ling. *IEEE Transactions on Parallel and Distributed Systems (TPDS'01)*.

[24] Michael L Pinedo. 2012. *Scheduling*. Springer.

[25] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. 2019. Swift machine learning model serving scheduling: a region based reinforcement learning approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*.

[26] Pedro Eugênio Rocha and Luis CE Bona. 2012. A QoS aware non-work-conserving disk scheduler. In *Proceedings of the 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*.

[27] Emilia Rosti, Evgenia Smirni, Giuseppe Serazzi, and Lawrence W Dowdy. 1995. Analysis of non-work-conserving processor partitioning policies. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'95)*.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv:1707.06347*.

[29] Harmel Singh and Abdou Youssef. 1996. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *Proceedings of the 5th IEEE heterogeneous computing workshop (HCW'96)*.

[30] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[31] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems (NeurIPS'00)*.

[32] Wei Tang, Zhiling Lan, Narayan Desai, and Daniel Buettner. 2009. Fault-aware, utility-based job scheduling on blue, gene/p systems. In *Proceedings of the International Conference on Cluster Computing and Workshops (CLUSTER'09)*.

[33] Jeffrey D. Ullman. 1975. NP-complete scheduling problems. *Journal of Computer and System Sciences (JCSS'75)*.

[34] Lingfei Wang, Aaron Harwood, and Maria A Rodriguez. 2021. A Deep Reinforcement Learning Scheduler with Back-filling for High Performance Computing. In *Proceedings of the Asia-Pacific Conference on Computer Science and Data Engineering (CSDE'21)*.

[35] Fatos Xhafa and Ajith Abraham. 2010. Computational models and heuristic methods for Grid scheduling problems. *Future Generation Computer Systems*

(FGCS'10).

[36] Yuehai Xu and Song Jiang. 2011. A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics.. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11).*

[37] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'03).*

[38] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems (EuroSys'10).*

[39] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. 2020. RLScheduler: an automated HPC batch job scheduler using reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20).*