ESCHER: Expressive Scheduling with Ephemeral Resources

Romil Bhardwaj
UC Berkeley

Alexey Tumanov
Georgia Tech

Stephanie Wang
UC Berkeley

Richard Liaw
Anyscale Inc.

Philipp Moritz
Anyscale Inc.

Robert Nishihara
Anyscale Inc.

Ion Stoica
UC Berkeley

Abstract

As distributed applications become increasingly complex, so do their scheduling requirements. This development calls for cluster schedulers that are not only general, but also evolvable. Unfortunately, most existing cluster schedulers are not evolvable: when confronted with new requirements, they need major rewrites to support these requirements. Examples include gang-scheduling support in Kubernetes [**?** **?**] or task-affinity in Spark [**?**]. Some cluster schedulers [**?** **?**] expose physical resources to applications to address this. While these approaches are evolvable, they push the burden of implementing scheduling mechanisms in addition to the policies entirely to the application.

ESCHER is a cluster scheduler design that achieves both evolvability and application-level simplicity. ESCHER uses an abstraction exposed by several recent frameworks (which we call *ephemeral resources*) that lets the application express scheduling constraints as resource requirements. These requirements are then satisfied by a simple mechanism matching resource demands to available resources. We implement ESCHER on Kubernetes and Ray, and show that this abstraction can be used to express common policies offered by monolithic schedulers while allowing applications to easily create new custom policies hitherto unsupported.

ACM Reference Format:
Romil Bhardwaj, Alexey Tumanov, Stephanie Wang, Richard Liaw, Philipp Moritz, Robert Nishihara, and Ion Stoica. 2022. ESCHER: Expressive Scheduling with Ephemeral Resources. In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3542929.3563498

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563498

1 Introduction

With the end of Moore's law and Dennard scaling, developers are forced to distribute their applications to process an ever growing amount of data. As a result, the past decade has seen a proliferation of new distributed frameworks [**?** **?** **?**] to handle a variety of workloads from big data (e.g., batch jobs, interactive query processing) to AI applications (e.g., model training and serving).

As the number of data and AI applications grows, so do their scheduling requirements. Some examples of scheduling policies are *affinity* (i.e., co-locate computation with data to avoid costly data transfers), *anti-affinity* (i.e., schedule tasks on different machines to avoid interference), and *gang scheduling* (i.e., schedule a group of interdependent tasks simultaneously). For example, a hyperparameter search application [**?** **?**] consists of multiple distributed training jobs, each consisting of multiple parallel tasks. This requires anti-affinity between jobs for high throughput, affinity within a job to avoid unnecessary data transfers, and gang scheduling to ensure multi-node jobs are not starved.

This diversity of policy requirements makes designing schedulers for distributed frameworks challenging. There is an inherent trade-off between *simplicity* and *flexibility* in exposing different policies to applications. Different cluster managers occupy different points in this trade-off space.

At one end of the spectrum (Figure **??**a), monolithic cluster managers like YARN [**?**] and Kubernetes [**?**] provide several out-of-the-box policies for the application to chose from. This simplifies the application's task, but it compromises the flexibility, as adding a new policy requires changes to the scheduler and the cluster manager itself. Implementing a new policy requires the developer to understand and modify the source code of the cluster manager, not always an easy task given their inherent complexity. And, once the new policy is implemented, the developer is on the horns of a dilemma: either fork the project and pay the cost of maintaining it up-to-date as the project evolves, or wait many months for the change to be merged in the main branch. Worse yet, if the cluster manager is closed source, the application developer has no choice but to wait and hope that the company behind the cluster manager will implement the desired policy.

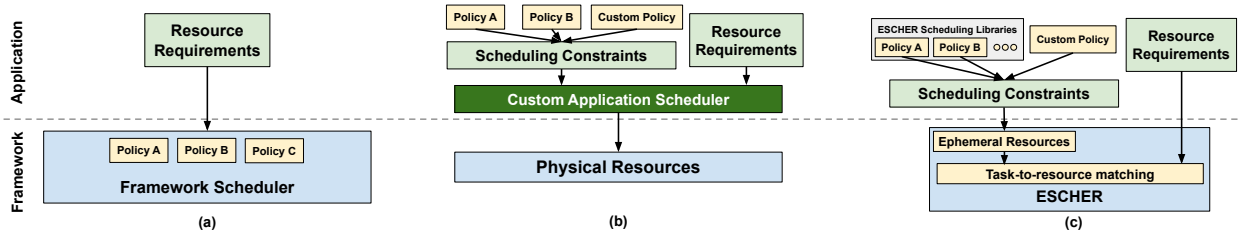


Figure 1: (a) A monolithic scheduler implements both scheduling and resource constraint matching [? ? ? ?]. Some schedulers allow applications to express and compose certain policies [? ? ?], but custom application policies may require modifying the scheduler itself. (b) To maximize flexibility, some frameworks expose physical resources [? ?], but require applications to write custom schedulers that manage both policy and resource coordination [? ? ?]. (c) ESCHER. With ephemeral resources, applications can express custom policies through ephemeral resources, while the cluster scheduler provides just one service - satisfying per-task resource constraints.

At the other end of the spectrum (Figure **??**b), are schedulers like Omega [**?**] and Mesos [**?**] that enable an application to directly allocate resources and implement its own scheduling logic. This makes these cluster managers very flexible, but dramatically increases the complexity of the application. Implementing a scheduling policy in a distributed system can be a daunting task, as it requires not only allocating resources, but tracking the resource availability in the presence of various failures and new nodes joining the system.

In this work, we present another point in the design space that allows application developers to easily implement a range of new scheduling policies. This design point is enabled by a mechanism recently introduced by cluster managers like Kubernetes and Ray which provides an interface for applications to dynamically create, modify, and destroy logical resources. We call these resources *ephemeral resources*. Like regular resources, ephemeral resources are pairs of labels and count values which can be allocated to tasks. The scheduler treats ephemeral resources in the same way as physical resources, subjecting them to admission control to ensure they are not oversubscribed. This frees the applications from performing admission control and tracking availability.

We find that a surprisingly large number of scheduling policies can be expressed by dynamically creating, updating and destroying ephemeral resources. Consider a simple scheduling constraint to colocate two tasks T1 and T2. To express this constraint, an application would submit T1, which creates an ephemeral resource R1 during execution, and then submit T2 with R1 as a resource requirement. The scheduler is then forced to place T2 on the same node as T1, since no other node has resource R1. While this is a very simple example, it illustrates the underutilized power of ephemeral resources for satisfying application-level scheduling constraints. In contrast, a monolithic cluster manager would have to expose a primitive designed specifically for task-task affinity, and a two-level scheduler application would have to implement the entire policy themselves, choosing where *both* T1 and T2 execute.

The key promise of ephemeral resources is that they enable an application to implement new scheduling policies *not* supported by the underlying cluster manager. This increases the velocity of deploying and iterating on new application functionality. However, there are two natural questions that follow. First, how general are the scheduling policies enabled by ephemeral resources? Second, what are the costs in terms of implementation complexity and overhead compared to natively implementing the same policy in the cluster manager? After all, if these overheads dominate, then an application developer is better off building their own scheduler.

To answer these questions, we propose a scheduling architecture for distributed applications called ESCHER [1]. In ESCHER, the application uses ephemeral resources to implement its scheduling policy instead of relying on the cluster manager's baked-in policies. The key insight of ESCHER is that a broad class of heterogeneous scheduling constraints can be cast as *ephemeral resource requirements*. The underlying scheduler simply enforces these requirements. ESCHER enables applications to implement a large number of scheduling policies by (1) dynamically creating new ephemeral resources, and (2) specifying task resource requirements on these ephemeral resources. We find that by using these two simple primitives, we are able to satisfy a large set of scheduling constraints, without requiring any changes to the core scheduler or significantly affecting application performance. For instance, gang-scheduling in ESCHER can be written in 10x fewer lines of code with less than 2x overhead in scheduling latency compared to implementing the policy natively in the core scheduler (**??**).

However, the flexibility of ephemeral resources does not come for free. First, it increases the application complexity compared to monolithic schedulers in which applications just need to select one of the available policies. Implementing

[1] ESCHER stands for Expressive SCHeduling with Ephemeral Resources.

certain policies, such as gang scheduling, requires the application to implement additional mechanisms using ephemeral resources such as ghost tasks, i.e., tasks whose sole purpose is to signal when all required resources have been allocated. Second, because ephemeral resources are created dynamically, an application must handle infeasible requests explicitly. For example, if a task's resource request cannot be satisfied, the task will hang and it must be explicitly terminated.

To alleviate the challenge of application complexity and provide protection against invalid resource specifications, ESCHER supports simple libraries to support common policies. We call these libraries ESCHER Scheduling Libraries (ESLs). ESLs aim to provide the best of both worlds: the simplicity of monolithic schedulers, and the flexibility of adding new scheduling policies at the application level by either extending an existing ESL or creating a new one. ESLs decouple application logic and policy by abstracting ephemeral resource management for common high-level scheduling policies, thus dramatically reducing development cost via code reuse and enabling composition of simple policies into more complex ones.

To evaluate ESCHER's performance, we implement it on both Kubernetes [**?**] and Ray [**?**] by leveraging their existing implementations for label-based scheduling, which were originally intended to represent custom physical resources rather than logical scheduling constraints. We run ESCHER on a range of applications and policies, including WordCount MapReduce with max-min fair sharing on Kubernetes as well as AlphaZero [**?**] and distributed model training on Ray [**?**]. We show that ESCHER does not impact the end-to-end performance of most applications when compared to a system that implements the same policies in the core scheduler. Meanwhile, the application can use ESCHER to express additional policies not supported by the underlying core scheduler, e.g., composing gang scheduling with affinity (**??**).

Thus, ESCHER shows that one can take advantage of the ephemeral resource abstraction, whose implementation is already partially provided by some cluster managers, to express a surprisingly diverse set of scheduling policies at the application level without having to touch the core scheduler. This allows users to quickly implement new policies, as needed, to improve support for their applications.

In summary, we make the following contributions:

- ESCHER, a scheduling architecture that uses ephemeral resources to express scheduling policies without modification to the core scheduler.
- Design and implementation of a wide class of scheduling policies (§**??**) using the ephemeral resources API.
- ESLs: application-level scheduling libraries that enable applications to easily compose and re-use policies.

2 Motivation

?? lists some common scheduling policies required by modern distributed applications and their off-the-shelf support across different frameworks and specialized schedulers. None of the schedulers support all policies, and many were built as a one-off solution to achieve a composition of these policies. New applications which require a new policy must find alternate methods of executing it - either by using some mechanism provided by the scheduler, such as labels, or writing their own scheduler from scratch. We now give a motivating example that is insufficiently served by existing schedulers and describe how ESCHER fills this gap.

2.1 Existing systems are hard to evolve

As applications become increasingly diverse, the cluster schedulers must evolve to support novel scheduling policies required by these applications. Consider distributed training, which has emerged as a dominant ML workload. Multiple training jobs are often scheduled simultaneously due to multi-tenancy and individual users submitting multiple training runs in parallel to evaluate different model architectures. This requires several distinct scheduling policies:

- *Anti-affinity:* Evenly spread training jobs across the cluster to ensure high throughput.
- *Affinity:* Co-locate all tasks of the same training job on the same machine to avoid unnecessary data transfers.
- *Gang scheduling:* For multi-node jobs, schedule all tasks of the job simultaneously to avoid starvation.
- *Bin packing (dynamic):* Monitor job utilization and consolidate jobs to reduce resource fragmentation.

Many popular schedulers implement at least one of these policies, but it is rare for them to support all four (**??**), never mind a composition of the policies. There are two fundamental challenges that make it difficult or infeasible to extend monolithic schedulers (**??**a) in this way. We use Kubernetes [**?**] to illustrate these challenges.

First, the application must *express* its policy using the API chosen by the framework scheduler. While some schedulers support composition, it is difficult in general to design a scheduler API that can capture all possible use cases. For example, in Kubernetes, applications specify scheduling policies with *static weights* to resolve conflicts. This can be used to express a composition of two policies that, for instance, weights affinity over anti-affinity. However, the complexity of composition is not linear in the number of policies. E.g., to add bin packing and prioritize it, the application would have to ensure that the weight of bin packing is always greater than the sum of weights for affinity and anti-affinity.

Second, the scheduler *implementation* must extend to new policies. This is difficult because the scheduler must ensure

Policy	Framework			Scheduler		
	YARN CS [?]	Kubernetes [?] Core / Labels	Spark [?]	Sparrow [?]	Gandiva [?]	Gorila [?]
Task co-location Place *n* tasks on the same physical machine.	✓	✓/✓	✗	✓	✓	✓
Data locality Place tasks with operands.	✓	✓/✓	✓	✓	✗	✗
Elastic Load Balancing Given an unknown number of tasks, evenly spread them out across *m* workers.	✓	✓/✗	✗	✓	✓	✓
Bin-packing Given an unknown number of incoming tasks, minimize the number of workers used to complete the tasks.	✓	✓/✗	✗	✗	✓	✗
Anti-affinity Given two tasks, place them on distinct nodes.	✓	✓/✓	✗	✗	✓	✗
Gang scheduling Given a set of tasks, enforce all-or-none run semantics.	✗	✗/✗	✓	✗	✗	✓
Weighted Fair Queuing[?] Given a set of tasks, enforce priority ordering.	✓	✓/✗	✓	✓	✗	✗
Soft-constraints For a priority ordering of resource constraints, schedule a task with the highest possible resource satisfiability.	✗	✓/✗	✗	✗	✗	✗

Table 1: Common scheduling policies and off-the-shelf support from existing schedulers. Kubernetes comparision includes both modes of operation, using just the core scheduling functionality and using labels. In addition to these policies, ephemeral resources allow applications to specify and compose custom policies.

that each new policy interfaces correctly with all other existing policies. Adding another policy requires modifying Kubernetes itself, which takes significant time and effort. Dynamic policies are even more difficult to support if the scheduler was not initially designed for it. For instance, adding gang scheduling support in Kubernetes took months of discussions and the eventual feature was not mainlined and instead implemented in an add-on scheduler [**? ? ?**]. Similarly, Machine Learning pipelines involve multiple interdependent tasks (e.g., data pre-processing, training, serving) defined in a DAG. Scheduling DAGs is not natively supported in Kubernetes, leading to the emergence of specialized plugins such as Kubeflow [**?**].

Due to these limitations, applications must write custom schedulers to maximize performance, as Gandiva [**?**] did for distributed training. Unfortunately, this design requires the application to implement both the policy and the scheduler *mechanism*, maintaining resource state and handlers for task and resource management, coordination, node addition and deletion, etc. (**??**b). This is both difficult to build and extend. E.g., Gandiva (built on Kubernetes) supports affinity, anti-affinity, and dynamic bin-packing, but the addition of gang scheduling would greatly increase the complexity of the scheduler code. Thus, Gandiva remains limited to distributed training jobs that fit on a single multi-GPU node.

2.2 Static labels are insufficient

Some frameworks [**? ? ? ?**] already support *static* label creation as string key-value pairs (e.g., "v100 GPU": 1) associated with cluster nodes. This allows cluster operators to tag nodes with physical resource attributes (e.g., CPU/GPU architecture, rack affinity) at cluster launch time, which can be requested by applications at execution time.

In ESCHER, we propose repurposing this API to express *custom application scheduling policies*, in addition to physical resource requirements. Unlike physical resources, which can be statically determined at a node's launch time, logical scheduling constraints may depend on run-time information. Therefore, it is natural to extend existing static label creation APIs to *ephemeral resources* that are dynamically created.

For example, to express task-task affinity between tasks T_1 and T_2, we must first learn where T_1 was placed before deciding the placement constraint for T_2. This can be easily done through ephemeral resources: T_1 dynamically creates a logical resource that is required by T_2. With static labels, the only option is for the application to pin T_1 and T_2 to a predetermined node.

For the same reason, there are some inter-task constraints that are fundamentally impossible to implement with static labels, such as scheduling policies that depend on *time*. One example is a DAG scheduling policy. At its core, this requires a primitive that guarantees that some task T_2 will not run until another task T_1 finishes. This is impossible to express using static labels alone, which cannot reason about the temporal ordering between two tasks.

3 ESCHER Design and Workflow

A scheduling policy is defined by a set of temporal and spatial dependencies between tasks and nodes. We call these dependencies *scheduling constraints*. The key idea in ESCHER is to map these scheduling constraints to resource requirements by introducing a new resource type, *ephemeral resources*.

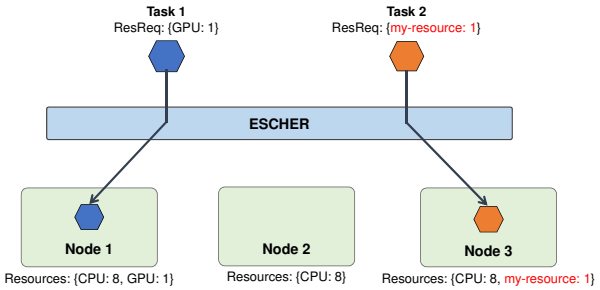


Figure 2: Example using ephemeral resources for task placement. Applications create ephemeral resources (*my-resource*) on the nodes where they wish to place a task and then launch a task requesting *my-resource*. The resource-matching scheduler ensures the task is placed on the desired node.

An ephemeral resource is a logical (i.e., non-physical) resource attribute that the application can dynamically associate with a node. Like physical resources, ephemeral resources have an associated capacity and can be acquired and released by tasks. We call these resources *ephemeral* because the application can create, modify, and destroy them at runtime.

ESCHER uses ephemeral resources for implementing scheduling constraints by leveraging a common functionality provided by cluster schedulers: matching the application-specified resource requirements with the cluster's resource availability. For instance, if an application task requires two GPUs, the scheduler should schedule that task on a node that has at least two available GPUs. With this resource-matching capability, the scheduler treats an ephemeral resource like a physical resource and aims to satisfy its capacity constraints.

The implementation of scheduling policies in ESCHER follows a two-step pattern. First, the application creates ephemeral resources or updates capacities of existing ephemeral resources. This is done programmatically at runtime through the ephemeral resource API. Second, the application associates ephemeral resource requirements with tasks. Note that these steps can happen in any order. This allows the application to make targeted placement decisions (Figure **??**) to satisfy the policy's scheduling constraints.

3.1 ESCHER Workflow

Figure **??** describes the workings of an ESCHER scheduler. ESCHER functionality, by design, is split between the application and the resource management framework. The application can specify scheduling policies through the ephemeral resource API (**??**), while the framework performs resource matching and accounting over a set of underlying physical resources. We envision that most applications would specify and compose policies through the higher-level ESCHER Scheduling Library (ESL) interface, which uses the ephemeral resource API to encapsulate common policies.

When using ESLs, an interaction with the system typically starts with an application requesting a scheduling policy from the ESL (**??**). The ESL may interact with the resource manager, e.g., by reading cluster state, and implements the policy by creating the appropriate ephemeral resources. The application then receives a resource specification R from the ESL. The application attaches R to a task and submits it to the resource manager for placement.

3.2 Ephemeral Resource API

In ESCHER, the resource management framework exposes two simple API calls to manage ephemeral resources: `set_resource` and `get_cluster_status` (Listing **??**). Once created, an ephemeral resource behaves as any regular physical resource and can be acquired and released by tasks.

In addition to the required parameters resource label and capacity, the `set_resource` call also allows the specification of constraints `node_spec` where the resource must be created. If `node_spec` is a resource vector, the resource is updated on all nodes where the constraint resource vector is a subset of the node's available resource vector. Optionally, a `num_nodes` field in the `node_spec` can specify how many nodes to execute `set_resource` on if multiple nodes satisfy the `node_spec` constraints. To make targeted `set_resource` calls, the `node_spec` can contain a unique node identifier (e.g., IP address).

The `get_cluster_status` call returns a mapping of node to local resource capacity and availability. These are not required for all policies, but can be useful to handle node additions and removals (**??**).

The ESCHER scheduler's responsibility is to provide the minimal guarantees provided by any resource-matching scheduler: (1) A task whose resource requirements can be met by a node in the cluster will eventually be scheduled, and (2) A node is never allocated past its capacity. Together, these imply that the scheduler implements: (1) task queuing and dispatch, (2) node selection for each task, and (3) resource allocation for each task. Note that the scheduler does not need to satisfy any other constraints, such as a promise regarding the node where a task is actually scheduled.

3.3 ESLs - ESCHER Scheduling Libraries

Controlling task placement through direct manipulation of ephemeral resources can be burdensome for applications with conventional scheduling requirements. To reduce the application complexity and delineate scheduling policies from the mechanisms to implement them, we propose ESCHER Scheduling Libraries (ESLs).

The role of an ESL is simple - given a set of tasks, generate and apply a set of ephemeral resource requirements on the tasks which satisfy the desired scheduling policy. ESLs achieve this by encapsulating the state management for

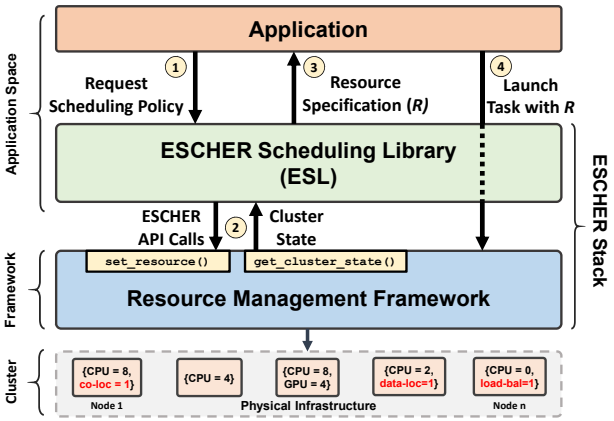


Figure 3: ESCHER task submission workflow with an ESL mediating the implementation. A task requests a supported scheduling policy from the ESL, which invokes the ESCHER API if necessary and returns the resource specification which would satisfy the policy. The task is launched with the returned resource specification.

Listing 1: Ephemeral resource API

ephemeral resources and providing a unified API for implementing domain-specific scheduling policies. An application requests a scheduling policy supported by an ESL, which then makes the appropriate ESCHER API calls and returns the resource specification the application must use to realize its desired policy. In our implementations, ESLs are designed as a daemon that can service scheduling requests made from a single or multiple applications.

ESLs are similar in spirit to Library Operating Systems (LibOS [?]) in the Exokernel [?] design. In the same way that a LibOS encapsulates the complexity of direct resource management exposed by an Exokernel, an ESL abstracts the policy implementation and enables sharing across different applications. Moreover, this design makes applications portable across different cluster frameworks, e.g., Ray vs. Kubernetes (??), since ESLs separate the application policy from the application code. Finally, ESLs can protect applications from invalid specification of ephemeral resources by validating resource requests before launching tasks.

Since distributed applications can have widely varying scheduling requirements, we anticipate the development of domain-specific ESLs which can strike a balance between generality and preserving domain-specific optimizations. As an example, we describe and implement an ESL for hierarchical max-min fair sharing in ?? and ??.

3.4 Fault tolerance

In the event of a node failure, ESCHER works in tandem with the fault-tolerance scheme of the underlying framework. Most frameworks [? ?] simply re-execute the tasks with the same resource requirement. However, since these resource requests include ephemeral resources which no longer exist, these re-executed tasks cannot be scheduled.

At the bare minimum, an ESL must ensure that ephemeral resources are restored at some other eligible node. To do this, ESCHER relies on the cluster framework to report failed nodes through the `get_cluster_status` API. Once failed nodes are detected, an ESL can recreate the ephemeral resources on suitable nodes by replaying the `set_resource` calls for the failed resources. If a candidate node is found, the resource is recreated, else the tasks must wait for the failed node to recover before they can be rescheduled. When a node is restored, its resources are also reinitialized, allowing any waiting tasks to get rescheduled.

3.5 Evolvability and complexity in ESCHER

The ability to create ephemeral resources on targeted nodes makes scheduling with ESCHER as flexible as letting the application directly control task placement on cluster nodes. Indeed, scheduling a task T on node N is equivalent to assigning a uniquely-named ephemeral resource R_N with capacity 1 to node N, and having T request one unit of resource R_N.

While this targeted placement makes ESCHER highly evolvable, what are its benefits over simply yielding placement control over resources to the application directly? After all, if the goal is to schedule a task on a particular node, ESCHER makes this operation arguably more complex as it requires creating an ephemeral resource on the node.

The primary benefit of ESCHER is that policy and ESL implementations do not need to reason about *tasks*. In a framework with fully application-level scheduling, such as Mesos or Omega [? ?], the application scheduler has to maintain possibly distributed state about the current set of tasks. When a task is submitted and can't yet be scheduled, the scheduler queues the task. When a task starts, the scheduler must update the current resource availability to ensure that resources do not become oversubscribed. On task completion, the scheduler must again update the current resource availability and select a new task to run from the queue.

Of course, none of these functionalities are unique to ESCHER. Task to resource matching is a necessity to every scheduler system, which is why it is the core responsibility that we assign to the ESCHER scheduler. Thus, in ESCHER, *an application that has no specific policy requirements can use an ESCHER scheduler directly without implementing any scheduling code.* This is not possible in systems that expose resources directly to the application, such as Mesos [?] or Omega [?], as it is expected that the application will also implement all mechanisms related to task scheduling.

For applications that do require a custom policy or an ESL, this division of responsibilities still reduces the effort

required from the application developer, compared to implementing a complete scheduler. For example, most of the policies that we present in **??** do not require examining the current cluster state; it is enough for a task to create a resource on its local node or create resources on all nodes that match a given resource spec. The exceptions are gang scheduling, which requires reading the cluster state to roll back a group of tasks in case of a failure, and load-balancing, which computes the current load from the cluster state. In contrast, a fully application-level scheduler must continually update and reason about the current cluster state in order to find a suitable node for each task. In the ESCHER design, this responsibility is given to the system rather than the application.

4 Scheduling with ESCHER

A scheduling policy is a mapping of tasks to resources which satisfies any spatial ("where") and temporal ("when") constraints. We now describe how these constraints can be cast as resource requirements with ESCHER.

4.1 Scheduling primitives in ESCHER

We present four scheduling primitives implemented using ephemeral resources which can be used to express both spatial and temporal constraints. We note that this is not an exhaustive set of primitives possible with ephemeral resources. Applications have the flexibility to define their own primitives through ephemeral resource manipulation.

[P1] Locality. Tasks must often be co-scheduled on the same physical node as another task or must be co-located with data. These spatial constraints can be easily expressed in ESCHER. The target task for co-location creates a local ephemeral resource E_r with unbounded capacity when the task starts or the data to be co-located with is created. The constrained task then requests 1 unit of E_r and, thus, automatically gets scheduled on the same node.

[P2] Task Signaling. Distributed applications rely on expensive RPCs to coordinate the execution of interdependent tasks. This is prevalent in directed acyclic graph (DAG) task schedulers, where the ordering of tasks is critical for correctness. These temporal constraints can be expressed with ESCHER by creating ephemeral resources dynamically, effectively using them as signals. E.g., if task T2 has *any* "happens-before" dependency on task T1, T1 can create a resource E_{T2} when it completes. T2 *a priori* requests E_{T2} as a part of its resource requirements when launched, and thus is scheduled as soon as E_{T2} is created by T1. Note that signals in ESCHER are single-shot—all task requests are declaratively placed at the start, and tasks begin execution only when their ephemeral resource demands are met by newly created resources. More generally, barrier synchronization is naturally supported. Given $\left\{T_j^{i-1}\right\} \rightarrow T^i$ for $j > 1$, T^i could simply request a single unit of resource created by each of T_j^{i-1} upon their respective completion. Thus, semaphores (and therefore, mutual exclusion) can also be implemented.

[P3] Queues. Many policies [**? ?**] use one or more task queues as a fundamental construct in their implementation. The core ESCHER scheduler queues tasks until their resource requirements (ephemeral and physical) can be satisfied. ESCHER allows the application to decide when to dequeue tasks by increasing the capacity of an ephemeral resource. Creating a queue is simply creating a unique ephemeral resource E_q with initial capacity 0 on any node. A task is enqueued by launching it in a wrapper task requesting 1 unit of E_q resource. The queue drain rate can be set by changing the capacity of E_q. On acquiring the E_q resource, the wrapper task submits the contained task to the scheduler with its physical resource requirement (e.g., 2 CPUs) and exits. Note that it's possible to implement batched scheduling by incrementing the capacity of E_q by the desired batch size.

[P4] Resource Locking. ESCHER enables a new scheduling construct where an ephemeral resource can be used to acquire and lock one or more physical resources ("bundle"). This reservation of resources is achieved with *ghost tasks* - long-running tasks which acquire the bundle like a regular task and create a local ephemeral resource to accommodate new tasks. Ghost tasks create a pattern of indirection where tasks request ephemeral resources instead of physical resources to get scheduled. We note that ghost tasks achieve the same outcome as incremental locking presented in Omega [**?**]. This is useful when applications require atomic transactions on a *group* of resources, such as in gang scheduling.

4.2 Scheduling policies with ESCHER

To illustrate the use of these scheduling primitive constructs, we now describe the implementation of an example application-level policy and a cluster-level policy with ESCHER. **??** lists more policies and their implementation with ESCHER.

4.2.1 Application Policy: Gang Scheduling Distributed training [**?**] and reinforcement learning workloads [**?**] require gang scheduling, where all tasks should start and run concurrently. This implies all-or-none scheduling semantics, where either all resources requested by all tasks are granted simultaneously, or no resources are granted.

In implementing all-or-none constraints, a common requirement is to check whether sufficient resources are available to satisfy the policy and reserving them, if necessary. To achieve this, ESCHER uses *ghost tasks* from the resource locking primitive. For instance, gang-scheduling a pool of 8 tasks (each of which requires 1 CPU) can be done by launching a ghost task which requires 8 CPUs and creates 8 units

Policy Example	Primitive used	Implementation with ESCHER
Sequential: Run T_2 after T_1.	Signaling	T_2 requests ephemeral resource E created by T_1 on completion.
Gang Scheduling: Run T_1 and T_2 simultaneously.	Locking, Signaling	Two ghost tasks T_1^g and T_2^g request 1 CPU each, and each creates an ephemeral resource E_{CPU} of capacity one. When both T_1^g and T_2^g run, schedule T_1 and T_2, each requesting one unit of E_{CPU}.
Affinity: Run T_1 and T_2 on the same node.	Locality	A ghost task T^g requests 2 CPUs, and creates an ephemeral resource E with capacity 2. T_1 and T_2 request one unit of E each.
Anti-affinity: Run T_1 and T_2 on different nodes.	Queues	Every node in the cluster creates anti-affinity resource E with capacity 1. T_1 and T_2 request one unit of E each.[1]
Load-balancing: Evenly spread tasks across nodes.	Queues	Create load-balancing resource L with capacity 1 on each node. Each task requests one unit of L each. When all L resources are exhausted, increase capacity by 1.
Data-locality: Run T where its input D is stored.	Locality	When storing D, create ephemeral resource E_D on the same node. T requests E_D.

[1] If T_1 and T_2 are long-running, the application cannot use the nodes they are running on for other anti-affinity placements. To avoid this, we have two short-lived ghost tasks T_1^g and T_2^g request 1 unit of E each, create ephemeral resources E_1 and E_2, and then terminate. T_1 requests E_1 and T_2 requests E_2.

Table 2: Expressing scheduling constraints with ephemeral resources

(a) **(b)**

Figure 4: Scheduling with ESCHER. (a) Soft constraints with ESCHER. (b) Composition of load-balancing and co-location policies with ephemeral resources in ESCHER.

of `gang-sched` resource. If all ghost tasks are successful, each task in the pool can then request 1 `gang-sched` resource and 0 CPUs to get scheduled. If any ghost task is unsuccessful, a timeout in other ghost tasks executes a rollback and removes the `gang-sched` resource. To avoid live-locks, either the applications can execute an exponential back-off [**?**] before retrying, or an ESL can serialize all gang scheduling requests through a common shared library. We discuss this design space in **??**.

4.2.2 Cluster Policy: Hierarchical Fair Sharing From a cluster operator's perspective, using ESCHER allows enforcement of cluster-level scheduling goals, such as multi-tenancy, while still supporting application-level scheduling policies described above. For example, consider large organizations, which typically have a cluster of resources shared among teams. This sharing has three requirements. First, the scheduler must allow assigning resource sharing weights to users. Second, to maximize resource utilization, the scheduler must implement max-min fairness [**?**], i.e., temporarily re-allocate idle resources to oversubscribed users. Finally, teams need to further partition their share of resources among sub-teams.

Hierarchical max-min fair sharing (HFS) can be implemented as an ESL using a variant of the Queue primitive. The HFS ESL is instantiated to operate on a specified domain of nodes and provides a single call - `create_user(id, weight )` - which returns a resource name unique to the user id. On invoking this routine, the ESL executes a `set_resource` call to create a unique resource (e.g., `res_user1`) with infinite capacity for the user on each allocated node. When a user submits a task, they must request a capacity of 1 their unique resource label (e.g., `res_user1`) which ensures their task is run only on the resources provisioned for them. Since ephemeral resources can be updated at runtime, the ESL dynamically resizes user allocations by adding and removing their ephemeral resources. Hierarchies in this setup can be created by launching multiple instances of the ESL and restricting their operating domain to the nodes granted by the parent ESL.

4.2.3 Soft constraints with ESCHER The core scheduler enforces task resource requirements as a hard constraint, keeping the core scheduling logic simple. However, some applications may demand relaxed scheduling semantics, where some resource requirements can be specified as *soft constraints*. Ephemeral resources can be used to implement soft constraints even when the scheduler only supports strict matching of resource requirements. First, the application specifies the soft constraints as an ordinal set of resource set preferences $R = [r_1, r_2, ..., r_n]$, where r_i is the i^{th} preferred resource requirement set. For instance, an application which prefers a GPU but will work without one would specify its resource requirements as $R = [\{gpu : 1\}, \{\}]$.

The soft-constraints ESL then instruments the application's tasks with a lightweight heartbeat sent to the ESL to notify it of successful scheduling when the task launches (Listing **??**). The ESL then sequentially attempts to launch a task, starting with resource requirement r_1. If the ESL does not receive the callback from the task within a certain timeout t, it implies the resource requirement was not matched. The ESL then cancels and resubmits the task, now with a resource requirement r_2. This best-effort scheduling is attempted for all resource preferences $r \in R$ until the scheduling succeeds or all preferences have been evaluated.

Objective functions. Soft constraints can be used to approximate policies that optimize a combined objective function. For instance, consider a policy which aims to balance both CPU and memory utilization in a cluster. Formally, given weights for memory and CPU utilization α and β, the objective is to maximize the minimum of $\gamma_n = \alpha M_n + \beta C_n$ across all nodes, where M_n and C_n are the memory and CPU utilization on node n (ranging between 0 and 1).

To express this policy in ESCHER, the scheduler first creates the resource *obj* on each node with a capacity of $\alpha + \beta$. A task with memory and CPU requirements m and n then specifies the soft constraint $R = [\{obj : \gamma\}, \{obj : \frac{\gamma}{2}\}, \{obj : \frac{\gamma}{4}\}, ...\{obj : \frac{\gamma}{2^k}\}]$, where $\gamma = \alpha m + \beta n$. The task also includes the hard constraints $\{MEM : m, CPU : n\}$. This preference places each task on the least utilized node, correct up to a factor of 2, while guaranteeing that no node is overallocated.

4.2.4 Policy Composition Applications like hyperparameter search require a hierarchical composition of other policies (**??**). Scheduling policy compositions can be logically expressed either as an AND conjunction or an OR conjunction. AND constraints are expressed by concatenating the ephemeral resource vectors for two policies. OR constraints are supported using soft constraints. For instance, if an application wants to co-locate $task1$ and $task2$ while load balancing their groups across the cluster, it can simply apply co-location on $task1$ and $task2$ and load balancing only on $task1$ as shown in **??**. Similarly, cluster-level policies can be composed with application-level policies (Section **??**).

Conflicting compositions of policies may render a task impossible to schedule. E.g., composing a cluster-level fair sharing policy and an anti-affinity policy may result in a infeasible task if the fair share of resources is insufficient. In such situations, one can specify conflict resolutions by using soft-constraints to relax scheduling policies. For instance, a soft constraint vector [{fair_a: 1, anti_aff: 1}, {fair_a: 1}] would try scheduling a task with fairness and anti-affinity, and relax the anti-affinity constraint if a conflict arises.

5 Implementation

ESCHER is a design that can be applied to both centralized and decentralized schedulers. We built two prototypes of ESCHER, on Kubernetes [**?**], a container orchestration framework with a centralized scheduler, and Ray [**?**], a distributed execution framework with a decentralized scheduler.

ESCHER inherits the scheduling properties of the parent cluster framework. For instance, it can utilize fractional and heterogeneous resources on Ray and Kubernetes. Since Ray and Kubernetes have a task-by-task scheduler, our current implementation also schedules in a greedy, task-by-task manner. However, ESCHER's queuing primitive can be used to extend a task-by-task scheduler to do batch scheduling.

Each framework handles isolation differently, which affects how ghost tasks are implemented. Ray does not enforce CPU affinity, so a ghost task can block the logical resource for the actual task without blocking the physical CPU. Kubernetes enforces isolation through cgroups. In this case, the ghost task reserves the CPU for its cgroup and when the actual task is scheduled, it is added to the same cgroup by running cgclassify in the task's preamble. The source code is available at https://github.com/romilbhardwaj/escher.

Kubernetes Implementation. A Kubernetes task (pod) specifies its constraints in the form of two sets: a set of filtering policies, such as resource demands, to enforce hard constraints and a set of weights for these built-in policies to add soft constraints. The scheduler first finds a set of candidate nodes, then computes a policy-weighted score for each node to find the best fit.

Kubernetes also allows the definition of arbitrary string and integer pairs associated with nodes known as *extended resources*. Extended resources are identical to regular resources in that they can be acquired and released by Pods, except they can be defined as arbitrary key value pairs. The extended resources API has conventionally been used by cluster operators for marking specialized hardware as a one-time operation when the cluster is initialized. In fact, application pods are not granted access to this API by default.

To implement the ESCHER *set_resource* API, we change the Kubernetes role-based access control to allow applications to directly invoke the extended resources API in Kubernetes. This grants applications the ability to create and update *extended resources* directly using the *patch_node_status* call. From a security perspective, ESCHER applications require access only to create and remove extended resources. The Kubernetes API should deny write access to physical resources. Additionally, we employ namespacing of resources to enforce isolation and prevent malicious applications from overriding other applications.

To force the Kubernetes scheduler to act as a simple resource-matching scheduler, the scheduler is invoked with only hard resource constraints set in the filtering policy. Weights for all other policies are set to zero. Thus, the combination of *extended resources* API with hard resource constraints makes it possible to implement ESCHER policies on Kubernetes *without making any changes to the core Kubernetes scheduler.*

Ray Implementation. Ray runs a scheduler per node, which collectively implement a bottom-up decentralized resource matching mechanism [**?**]. Each node in the cluster has a set of key-value pairs signifying resource labels and their quantity, e.g., `{"cpu": 8, "gpu": 1}`. Each task specifies its hard resource requirements with the same data structure.

Ray nodes share a centralized log of the total resource capacity at each node, where each entry represents the capacity at a node. The Ray scheduler matches a task to resources

by storing resource availabilities from other nodes in a map and allocating the first node which can satisfy the task's resource request. Since a scheduler's view of the global state is only eventually consistent, it can correct previous decisions by running the same logic to find another eligible node. To support ESCHER, we extend the Ray core to permit updates to each node's resource capacity at runtime. We restructure Ray's centralized log so that each entry stores only a *delta*, instead of the absolute capacity at that node. This guarantees no race conditions occur between resource updates, while avoiding expensive coordination, e.g., via a distributed lock.

6 Evaluation

In this section, we evaluate the following questions:

- Can existing distributed applications be ported to use ESCHER and what are its implications?
- What are the tradeoffs with implementing scheduling policies in the application space vs in the framework?
- What are the overheads of scheduling with ephemeral resources?

All evaluations use Amazon EC2 m5.12xlarge, m5.4xlarge or p3.8xlarge instances. Kubernetes clusters are provisioned using Amazon EKS running version 1.19.

6.1 End-to-end Evaluation

6.1.1 WordCount with MapReduce WordCount counts the number of words in large text datasets and is often implemented with MapReduce [**?**]. To avoid expensive data transfers, data locality is essential. We implement the map and reduce tasks as independent operators running in containers. The input files are chunks of a file with random words, each hosted by one of 100 nodes. The total input size is varied from 50 GB to 500 GB. We implement ESCHER on Kubernetes, using an ESL for data locality (**??**), and compare against Kubernetes's built-in data-locality policy [**?**] and a locality-unaware random policy.

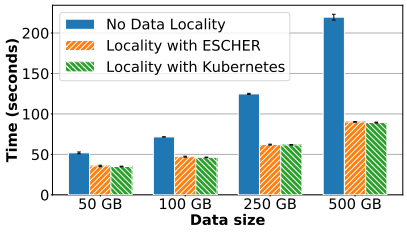
Unsurprisingly, **??** shows that as the input size increases, the overhead of transferring chunks over the network dominates the mapper computation time for the no-locality policy, taking up to 58.3% of the total job time when the input size is 500 GB. Meanwhile, ESCHER on Kubernetes provides the same performance as Kubernetes itself, but without modifying the core scheduler framework. Furthermore, **??** shows that ESCHER can also scale with the cluster size. Throughout different scales, ESCHER performs comparably with the core Kubernetes scheduler, with its makespan staying within 1.9% of the baseline Kubernetes scheduler.. Implementing data locality with ESCHER required adding only two lines: a `set_resource` call during data generation to create a local `data-<id>` resource and a line to specify a `data-<id>` resource requirement for the mapper tasks.

6.1.2 Hierarchical max-min fair sharing Hierarchical Max-Min Fair Sharing (**??**) allocates resources proportionate to a user's weight in a hierarchical organization. Users submit jobs at different times, so their ideal absolute resource share is dynamic, making it impossible to maximize overall resource utilization with static labels. For example, consider a two-team organization: Sub-Org1 with users A and B of weights 2:3, and Sub-Org2 with user C. To ensure fairness with static labels, the only option is to allocate each user a fixed proportionate share, leading to under-utilization when only one user is submitting work.

Because ephemeral resources can be *dynamically* created and destroyed, an HFS policy ensures fairness while also maximizing overall utilization as users enter and leave the system (**??**). We deploy a HFS policy on a 100 node cluster running WordCount. We use a parent ESL for the teams and two children ESLs for Sub-Orgs 1 and 2 to create a hierarchy of ESLs. An HFS ESL tracks idle resources and reallocates resources between teams or users. The workload in **??** starts with only user A submitting tasks to the scheduler. Since other users' resources are idle, the HFS ESLs re-allocate all idle resources to A to achieve max-min fairness. At time t=60, user B starts submitting tasks. This causes the Sub-Org 1 ESL to reclaim resources from A to re-allocate to B, in proportion to their weights. B's warmup time causes a small dip in net throughput at t=60. Finally at time t=120, user C starts submitting tasks and the parent ESL reallocates resources to Sub-Org2. Since Sub-Org 2 and Sub-Org 1 have equal weights, C's resource allocation is equal to the sum of A and B's allocation. Ephemeral resources also enable composition: the application composes its custom policy (in this case, data locality for WordCount) with the two HFS ESLs by concatenating all the resource requirements.

6.1.3 AlphaZero AlphaZero [**?**] is a reinforcement learning application for the board game Go. We demonstrate ESCHER's flexibility by porting an implementation [**?**] onto Ray without compromising performance relative to the optimal hard-coded (but inflexible) placement.

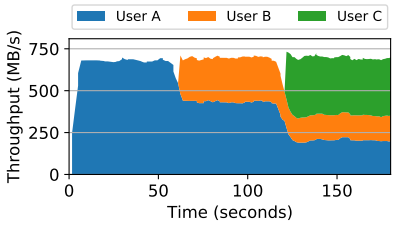
AlphaZero executes a Monte Carlo Tree Search on the game state space in a CPU-intensive *BoardAggregator* process. The search is guided by a *PredictorAgent* running a neural network on a GPU which evaluates a board and predicts the associated reward. Co-locating *BoardAggregator*s and their corresponding *PredictorAgent*s on the same physical node is thus desirable to avoid network overheads from transferring board states. These pairings also require anti-affinity for load balancing and to avoid interference [**?**]. With ephemeral resources, this composed policy can be specified in 5 lines of code (**??**): we apply a load-balancing policy (**??**) to the *PredictorAgent* and a co-location policy to the *BoardAggregator* and *PredictorAgent*.



(a)

Nodes	Scheduler: Generic	Kubernetes	ESCHER
10	183.32 ± 0.51	54.69 ± 0.46	55.24 ± 0.39
50	113.71 ± 0.49	44.02 ± 0.27	44.71 ± 0.44
100	51.90 ± 0.31	35.08 ± 0.31	35.76 ± 0.49

(b)



(c)

Figure 5: Data locality and hierarchical max-min fair sharing for WordCount. (a) Makespan of WordCount running on a 100-node Kubernetes cluster, comparing a random placement policy, ESCHER on Kubernetes with data locality, and Kubernetes' native data locality. (b) Makespan of WordCount MapReduce jobs in seconds across varying cluster sizes. (c) Hierarchical max-min fair sharing with ESCHER. A and B are in Sub-Org1 with weights 2:3; C is in Sub-Org2. A, B, and C begin submitting tasks at t =0, 60, and 120, respectively.

We ran 10k iterations of AlphaZero on a 32-node cluster (128 GPUs total). We compare three setups: (a) co-location with hard-coded placement, (b) co-location with ephemeral resources, and (c) a baseline policy with no co-location. **??** plots the CDF for board exploration time. Co-location is important for performance, outperforming no-colocation by 15.4% in median latency and 20% in P95 latency. Additionally, co-location with ephemeral resources adds insignificant overheads of <1%, while requiring less developer effort: the application code (**??**) does not need to match *PredictorAgent-BoardAggregator* pairs to specific nodes.

6.1.4 Distributed Training For a distributed training job, worker placement is critical to performance, as co-locating workers reduces the cost of model synchronization at each step. Gandiva [**?**] is a scheduler for deep learning jobs that aims to optimize training job performance. It composes a higher level *load-balancing* policy and a lower-level *co-location* policy to evenly spread jobs across machines while reducing intra-job communication overhead. To demonstrate ESCHER's flexibility, we augment Gandiva's [**?**] worker co-location and migration policy with Gang Scheduling to support distributed training jobs, and integrate the policy into Tune [**?**], an open source distributed training library built on Ray [**?**], which we will refer to as *EscherTune*. We modified the Trial abstraction in Tune to be wrapped in a ghost task that ensures gang scheduling and applied co-location on tasks belonging to the same Trial. EscherTune triggers a migration whenever it detects sufficient available resources to place all workers of a job on the same node. To execute a worker migration, EscherTune checkpoints the current job using application-specific checkpoint functionality and destroys all current workers. Then, EscherTune assigns ephemeral resources to the new target node, and relaunches all worker tasks of the training job without modifying their ephemeral resource requests.

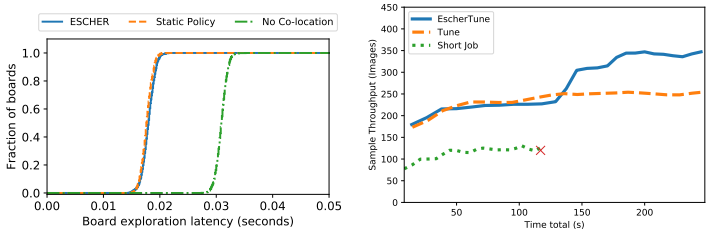
We compare EscherTune with Tune's open-source policy on a cluster of 12 GPUs. We launch 5 short-running training jobs (*short-jobs*), each requiring 1 GPU, followed by 1 long-running training job requiring 4 GPUs (*long-job*). Each training job is training a ResNet-101 model on CIFAR-10 with a batch-size of 64 images per device.

Initially, the *short-jobs* are load-balanced across the cluster, while the 4 workers of the *long-job* are spread across the cluster depending on GPU availability. This is a sub-optimal placement, so EscherTune migrates the *long-job* to colocate its tasks as soon as resources become available from a *short-job* completion, resulting in 36.3% higher throughput (**??**). Meanwhile, Tune uses a static placement, so the *long-job*'s throughput remains the same. Furthermore, EscherTune's implementation consists of only 50 lines of Python, with no changes to Tune or the Ray scheduler.

6.2 Microbenchmarks

6.2.1 Overhead of application-level policies ESCHER scheduling policies can be implemented either in the application space for evolvability or in the framework for performance. We evaluate the trade-offs involved in this choice by comparing three distinct designs of gang scheduling, all with ephemeral resources on Ray.

AppSpace uses ghost tasks to atomically reserve resources (**??**). While this policy is simple to integrate, the lack of coordination between applications can lead to deadlock, which must be resolved through timeouts. *LibSpace* avoids this by using a *shared library*: a shared service in the cluster that serializes gang scheduling requests across applications. *LibSpace* thus avoids live lock entirely but requires deploying a separate shared service. Finally, *FrameSpace* modifies the Ray scheduler to expose a gang scheduling API. Internally, a centralized service within Ray directly reserves and creates ephemeral resources. Since it has direct access to the resource table, *FrameSpace* avoids using ghost tasks, reducing overheads from worker allocation and task dispatch.



(a) (b) (c)

Figure 6: AlphaZero and distributed training on ESCHER. (a) Implementing AlphaZero policy with ESCHER, composing co-location with load-balancing. (b) A CDF of AlphaZero board exploration latency, and (c) Throughput comparison of a distributed training workload with a mix of short-running and long-running jobs. EscherTune is an augmentation of the hyperparameter search framework Tune [?], using ESCHER to dynamically re-schedule jobs as others complete. The red X indicates the completion of a short job.

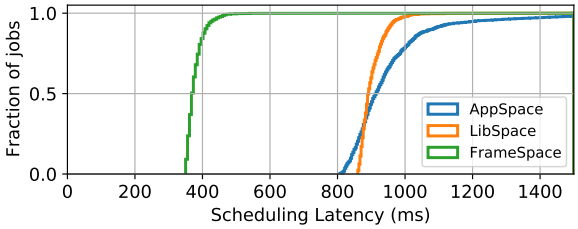


Figure 7: Request latency for gang scheduling implemented in the application space, with (*LibSpace*) and without (*AppSpace*) coordination, versus the framework space (*FrameSpace*). *FrameSpace* is 1624 lines of code (LoC), *LibSpace* with 261 LoC and *AppSpace* with 78 LoC.

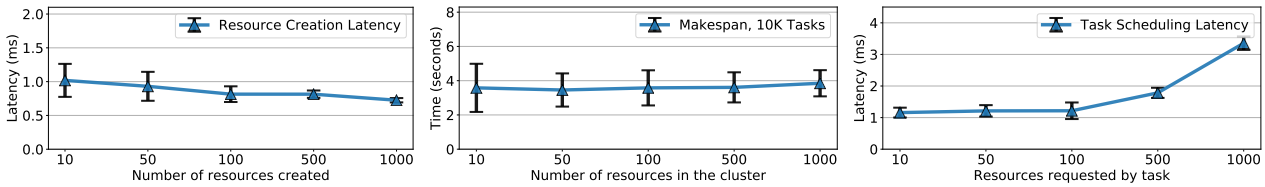
?? compares the request latency of these designs on a 32-node cluster with 256 CPUs. While the mean latency of *AppSpace* and *LibSpace* is similar, *AppSpace* has higher variance and a longer tail because it uses timeouts to break deadlocks. *LibSpace* incurs overhead from serializing requests at a separate service, resulting in a higher minimum latency. On average, *FrameSpace* is nearly 2× faster than *AppSpace* and *LibSpace* because it directly reserves resources instead of using ghost tasks. However, we note that for long-running tasks such as model training and batch processing workloads, the absolute scheduling latency is still a tiny fraction (<1s) compared to the runtime of the workloads (multiple hours). Moreover, implementing *FrameSpace* is a significant effort, requiring a deep understanding of the Ray scheduler and modifying 1624 lines of Ray code. To compare, *LibSpace* and *AppSpace* are implemented in 261 and 78 lines of *application-level* code, respectively.

6.2.2 Overheads of Ephemeral Resources We evaluate the time to create resources and propagate their availability throughout the cluster. Since the `set_resource` call is asynchronous, we verify that the resources have been created and are available for use by launching no-op tasks that request these newly created resources. Figure **??** compares the mean latency of creating an equal number of resources on each node in a 50-node Ray cluster. We show that even when creating 1000 ephemeral resources, we can maintain 1ms latency per request. As more resources are created, the cost of resource creation is amortized and the per-resource creation cost decreases to 0.72ms. In general, the overhead of creating or deleting an ephemeral resource should be roughly equivalent to that of a key-value store request.

Ephemeral resources and scheduling latency. The creation of ephemeral resources may add burden to the scheduler, as it must consider a greater number of attributes during resource matching. Therefore, we analyze the effect of resource creation on task scheduling latency. We create an equal number of resources across 50 Ray nodes in a cluster using the `set_resource` API. We then evaluate two cases based on the resource requirements of the tasks involved.

First, in Figure **??**, we launch 10,000 tasks, none of which require any ephemeral resources to be scheduled. As the tasks do not have any specific resource requirements, the scheduler execution time and workload makespan are not affected by the number of ephemeral resources present.

Second, when tasks do request ephemeral resources, the core scheduler must match the task's requirements to a set of candidate nodes. To evaluate the overheads introduced by this matching, we setup a 50 node Ray cluster and create 1000 unique ephemeral resources evenly spread across nodes. Figure 8c highlights the scalability of the scheduler as the number of ephemeral resources requested by a task grows. The the task scheduling latency grows only from 1.1ms to 1.2ms when requesting 1 vs. 100 ephemeral resources, respectively. We note that all policies described in this work require only a few ephemeral resources to express.



(a) (b) (c)

Figure 8: ESCHER microbenchmarks. (a) Mean per-resource creation latency in Ray. Creating ephemeral resources in ESCHER is a low-cost operation that scales linearly with the number of resources created. (b) Scheduling latency overheads from presence of ephemeral resources. Makespan of a 10000 task workload remains unaffected by the count of ephemeral resources in the cluster. (c) Effect of task resource requirements on scheduling latency in an environment with 10000 resources.

7 Discussion

Why use ESCHER? For common scheduling policies, ESCHER's primary benefit compared to a monolithic cluster manager is not performance. Rather, it's the ease to specify and implement new policies without requiring any changes to the cluster scheduler. This unlocks developing new applications with sophisticated scheduling constraints that are not yet supported by the underlying scheduler. One example is the *composition* of affinity and gang scheduling policies used in the distributed training example in **??**, which is not supported by Ray's native scheduling primitives. Another example is DAG-based scheduling, which is offered natively by Ray but not Kubernetes. DAG-based scheduling can be implemented by leveraging the task signaling primitive (**??**). Effectively, ESCHER expands the set of policies a monolithic cluster manager can support.

Two-level schedulers [**? ?**] achieve the same goal by exporting scheduling control directly to applications, but in doing so they also require applications to implement the entire scheduler themselves (Section **??**). ESCHER on the other hand requires minimal changes to application code: it required adding only two lines of code to MapReduce, five lines to AlphaZero and fifty lines to EscherTune, each of which had widely varying policy requirements. For policy composition especially, this ease of development is due in part to the use of ESLs.

Limitations. A key goal of ephemeral resources is to provide a simple and narrow API that is easy to implement by most cluster managers. As a result, ESCHER eschews abstractions that would require complex implementations such as transaction support [**?**] (which would allow to trivially implement gang scheduling) or utilization-based scheduling, such as load balancing (**??**). While the application can still implement these policies using ghost tasks, these implementations have inherently a higher overhead. Of course, if applications require higher scheduling performance, we can eventually implement these policies in the cluster manager. Even in this case, ESCHER remains valuable as it can bridge the gap by enabling the applications to implement these policies before the cluster manager does.

Another limitation of ESCHER is that it doesn't expose the resource availability to the application, which means that the only way for an application to learn that there are not enough resources available is by submitting a task which hangs. As a result, the application might have to explicitly kill the tasks that hang, adding to the complexity. We do not expose resource availability is because it would not fully solve the problem: there is still a race condition when two tasks on different machines simultaneously request more than the available resources (e.g., a single GPU is available and two tasks simultaneously request one GPU each). This problem is exacerbated by the fact that ephemeral resources can be dynamically created, modified, or destroyed. One solution to this problem is providing transaction semantics, which, as mentioned above, ESCHER eschews due to its complex implementation. An avenue for future work would be to alleviate the challenges of handling such a dynamic environment, e.g., by using lazy execution or extending the API to allow constraints on ephemeral resources.

8 Related Work

Monolithic schedulers. Monolithic cluster schedulers aim to implement both the scheduling policy and mechanism for a distributed application. Some provide a single generic scheduling discipline, such as fair sharing [**? ?**], gang scheduling [**?**], or delay scheduling [**?**]. These schedulers provide little control to the application beyond the ability to set some configuration knobs, such as the time to wait before scheduling a task on a node that doesn't store its inputs [**?**].

Other monolithic schedulers aim for generality and provide APIs to allow applications to express scheduling constraints. Examples are YARN [**?**], Condor's ClassAds [**?**], and Kubernetes [**?**]. Their monolithic design makes it hard to add support for new policies and their compositions. For instance, adding support for gang scheduling to Kubernetes requires deep structural and API changes, and was eventually implemented as a standalone system [**?**]. Often the API they provide is complex as well, since it must be expressive enough to capture complex constraints such as composition. For instance, the specification of ClassAds [**?**] is 35 pages [**?**].

ESCHER achieves both *simplicity* and *evolvability* by decoupling policy specification and the resource-matching mechanism.

Like ESCHER, DCM [**?**] also aims to maximize extensibility of framework schedulers by using a declarative model for applications to specify their desired policy behavior as SQL queries. In doing so, DCM deploys a custom scheduler and optimizer running with Kubernetes. While this is well suited for policies which optimize for global objectives, expressing application-level constraints requires users to create and maintain a table in cluster state database, which can be challenging in a distributed environment. ESCHER's emphasis lies on supporting application-level scheduling goals, allowing it to easily handle task-task dependencies (Primitive P2 in **??**). Moreover, ESCHER reuses existing virtual resource implementations, thus requiring no additional services or schedulers to be deployed in the cluster framework.

Rayon [**?**] is a space-time reservation admission system, allowing applications to reserve a *skyline* of resource capacity, $c(t)$, as a function of time. ESCHER can implement a discrete version of this by having a ghost task evaluate $c(t)$ and update ephemeral resources to match $c(t)$ at any instant.

Two-level schedulers. Rather than trying to implement application level policies, some cluster management frameworks are designed explicitly to give all resource management and scheduling control to the application. Many of these frameworks employ a two-level hierarchy [**? ?**], where the first level manages only resource isolation between applications, while the second level exposes physical resources to applications. These applications are then responsible for building their own scheduler. Omega [**?**] follows a similar separation by providing transaction semantics on a shared cluster state for distributed schedulers. While this approach grants maximum flexibility to applications, it adds significant complexity to application code since the application must now handle both scheduling policy and the mechanisms to ensure resource coordination between tasks. Some popular frameworks, such as Spark [**?**] and Flink [**?**] obviate the need for distributed coordination by designating a special node (e.g., master) to spawn all tasks. However, they too have monolithic designs that are not evolvable. In contrast, ESCHER focuses on providing a generic scheduling framework where the application only focuses on the scheduling policy. Indeed, ESCHER can be used in tandem with two-level schedulers by launching an ESCHER scheduler to manage resources allocated by the top-level scheduler.

Label-based and declarative scheduling. [**? ? ? ? ?**] provide mechanisms to annotate nodes with resource types and use these labels (e.g., "GPU:Nvidia:V100") for placement constraints. In some cases, these labels do not have an associated capacity (e.g., string key-value pairs), rendering infeasible implementation of policies with *quantitative* conditions. In other cases, quantitative labels are static. TetriSched [**?**] operates on labelled resources by allowing declarative resource constraint specification and composition. Wrasse [**?**] uses the bins and balls abstraction along with user-defined utilization functions to come up with a specification language. However, neither of these provide support for *dynamic* scheduling policies, e.g., making inter-task constraints hard to implement in a single shot. The expressivity of declarative schedulers is restricted to information known *a priori* (i.e., static label information). Circular inter-task dependencies ([**?**]) are fundamentally impossible to implement without a dynamic mechanism to unroll the dependency (**??**). We note, however, that declarative schedulers ([**? ?**]) are synergistic with ESCHER. Ephemeral resources can be used as an intermediate representation (IR) for their frontend API (e.g., SQL [**?**] and STRL [**?**]).

Some existing schedulers provide the ability to configure non-physical resources, e.g., the *extended resources* API in Kubernetes [**?**]. The original purpose of this mechanism is for the cluster operator to add accounting for custom resources (e.g., accelerators). Meanwhile, generic application-level scheduling policies like affinity and load balancing are still implemented in the Kubernetes core. In contrast, ESCHER obviates the need to implement these policies in the core scheduler, deferring it to the application level via the mechanism of ephemeral resources. In fact, the ESCHER implementation on Kubernetes repurposes extended resources to implement *all* scheduling policies and simplifies the Kubernetes scheduler to only resource matching. Finally, the ability to dynamically update ephemeral resources at runtime enables expressing previously inexpressible, e.g., inter-task "happens-before" relationships and iterative task graphs.

9 Acknowledgements

We thank the SoCC reviewers and our shepherd, Lin Wang, for their invaluable feedback. This research is partly supported by NSF (CCF-1730628) and gifts from Amazon, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.