ESCHER

Expressive SCHEDuling with Ephemeral Resources

Romil Bhardwaj, Alexey Tumanov, Richard Liaw, Stephanie Wang, Robert Nishihara, Philipp Moritz, Ion Stoica
Typical Distributed Application

**Application** is composed of tasks with resource requirements

**Scheduler** matches task resource requirements to node resource availabilities

**Cluster** is composed of nodes with resource configurations

---

Node 1
- Resources: {CPU: 1, GPU: 1}

Node 2
- Resources: {CPU: 4}

Node 3
- Resources: {CPU: 4, GPU: 4}

Task 1
- ResReq: {CPU: 1, GPU: 1}

Task 2
- ResReq: {CPU: 1, GPU: 4}

Application 1
- Scheduler
- Distributed Execution Framework
Example - Distributed Training

Scheduling Requirements

1. Gang Scheduling
   • Scheduling co-dependent tasks requires all-or-none semantics

2. Co-location
   • Tasks of a job share parameter updates and must be placed on the same node for performance

3. Anti-affinity
   • Avoid interference and resource contention by spreading jobs evenly spread across nodes
Example - Distributed Training

Scheduling Requirements

1. Gang Scheduling
   • Scheduling co-dependent tasks requires all-or-none semantics

2. Co-location
   • Tasks of a job share parameter updates and must be placed on the same node for performance

Supporting custom scheduling constraints requires evolvable schedulers
Evolvability in Monolithic Schedulers

• Applications state resource requirements

• Scheduler provides a fixed set of supported policies
  • E.g., Affinity, anti-affinity

• Challenging to evolve
  • Implementing custom policies requires modifying the core scheduler
  • Can take months to add support
  • Difficult to maintain - must commit to maintaining branch

Kubernetes, YARN
Evolvability in Two-level Schedulers

Mesos, Omega

- Physical resources are exposed to applications
- Applications implement end-to-end scheduling
- Highly flexible, but application must implement a scheduler:
  - Resource state tracking
  - Task queueing
  - Fault tolerance
Summary of solutions today

**Monolithic Schedulers**
Simple, but hard to evolve
- Application Layer
- Cluster Framework

**Two-level Schedulers**
Highly evolvable, but complex
- Application Layer
- Cluster Framework

**ESCHER**
Simple and evolvable
- Application Layer
- Cluster Framework
ESCHER Insights

With the following two scheduling abstractions, frameworks can allow applications to express a wide range of scheduling policies:

1. A simple resource matching scheduler

2. An API for applications to create resources at runtime
Abstraction 1 - Resource Matching Scheduler

**Scheduler** matches tasks resource **requirements** to node resource **availabilities**

- **Task 1**
  - ResReq: {CPU: 1}

- **Node 1**
  - Resources: {CPU: 8}

- **Node 2**
  - Resources: {CPU: 8}

- **Node 3**
  - Resources: {CPU: 0}
Abstraction 2 – Create Resources on-the-fly

Applications should be able to **create resources and get cluster state** at runtime through an API

```python
def set_resource(resource_name, capacity, node_spec=None)
```

```python
def get_cluster_state()  # Returns a map of {node: resources}
```

- Can specify resource availability constraints for resource creation
- If not node_spec not specified, resource created locally
A simple resource matching scheduler can be induced to make targeted placement decisions with short-lived *ephemeral* resources.
Example - Task co-location

Run tasks on the same node

```python
def task1():
    set_resource(label="co_location", capacity=1)
    ...

def task2():
    ...

def main():
    launch(task1, res = {})
    launch(task2, res = {"co_location": 1})
```

ESCHER allows declarative specification of scheduling policies by dynamically creating ephemeral resources.
Example - Load Balancing

Spread tasks across machines

```python
def static_load_balancing(num_tasks, num_nodes):
    resource_capacity = ceiling(num_tasks/num_nodes)
    set_resource(label="load_bal", node_spec={},
                  capacity=resource_capacity)
    for task in tasks:
        task.resources = {"load_bal": 1}
        task.launch()
```

Many more ESCHER policies (gang scheduling, bin-packing, anti-affinity, soft constraints, priorities, hierarchical fair sharing) in the paper!
Policy Composition: Load Balancing & Co-location

Co-locate two tasks and spread out pairs of tasks

```python
def task1(id):
    set_resource(label=id, node=None, capacity=1)

def task2():
    ...

def main():
    # Create load-balancing resources
    set_resource(label="load_bal", capacity=1, node_spec={})

    # Launch tasks
    # Launch task 1
    launch(task1, id=f'task{i}', resources={"load_bal": 1})

    # Co-locate task 1 & 2
    launch(task2, resources={f'task{i}': 1})
```

Compositions of policy can be represented as combinations of ephemeral resource constraints
Policy Composition: Load Balancing & Co-location

Co-locate two tasks and spread out pairs of tasks

```python
def task1(id):
    create_resource(label=id, node=None, capacity=1)
    ...

def task2():
    ...

def main():
    # Create load-balancing resource
    create_resource(label="load_bal", capacity=1, node_spec={})

    # Launch tasks
    for i in range(0, task_count):
        # Load balance task 1
        launch(task1, id=f'task{i}', resources = {'load_bal': 1})
        # Co-locate task 1 & 2
        launch(task2, resources = {f'task{i}': 1})
```

Do I have to maintain this resource-policy mapping in the application?
ESCHER Scheduling Libraries (ESLs)

• An app-level library of scheduling policies which encapsulate all state management for ephemeral resources

• Encourage code-reuse and simplify application code

```python
def colocated_task():
    ...

def main():
    esl = CoLocationESL()
    coloc_res = esl.get_colocation_group("mygroup", res_req={gpu: 8})
    launch(colocated_tasks, res += coloc_res)
```
ESCHER Workflow

Application Space

1. Request Scheduling Policy
2. ESCHER API Calls
3. Resource Specification (R)
4. Launch Task with R

ESCHER Scheduling Library (ESL)

ESCHER Scheduler

Framework

- set_resource()
- get_cluster_state()

Physical Cluster

- Node 1: {CPU = 8, co-loc = 4}
- Node n: {CPU = 4, GPU = 4, data-loc = 1, load-bal = 1}
Implementation

- Modified the Ray Scheduler to support online resource updates
- No changes required in Kubernetes core – we reuse the extended resources API
Evaluation - AlphaZero

• AlphaZero trains an RL agent to play Go

• Training has two key processes:
  • **Board Generation**: CPU intensive generation of possible game states
  • **Board Evaluation**: A GPU agent predicts the “goodness” of the generated states and chooses an action

• These processes require both **co-location** and **load-balancing**
ESCHER is **1.5-2x faster** in exploring board states than a locality-unaware scheduler.

ESCHER performs **comparably** with a static hard-coded policy with just **5 lines of code changes**.
ESCHER is **1.5-2x faster** in exploring board states than a locality-unaware scheduler.

ESCHER performs **comparably** with a static hard-coded policy with just 5 lines of code changes.

More results (MapReduce on 100 node K8s cluster, Hierarchical Fair Sharing, Distributed Training, Microbenchmarks) in the paper!
Using ESCHER adds latency for some policies such as gang scheduling, but significantly reduces the implementation burden.
ESCHER Summary

• Applications need **fine-grained scheduling** control without the complexity of implementing scheduling mechanisms.

• ESCHER presents an evolvable scheduler architecture with two key abstractions – a **resource matching scheduler** and **set_resource API**

• Ephemeral resources are **easily implemented in Ray and Kubernetes** and provide scheduling flexibility for a range of workloads with minimal overhead.