Can't Be Late: Optimizing Spot Instance Savings under Deadlines

Yongyi Wang

Overview

Spot Instances: Lower cost but preemptible, making them challenging for deadline-sensitive jobs.

Use on-demand instances as backup to meet deadlines while leveraging spot instances for cost savings.

Key Contribution: Development of the Uniform Progress policy, which is:

- Parameter-free
- Does not rely on assumptions about spot instance availability

Empirical Study:

- Based on real AWS spot availability traces
- Demonstrates significant cost reduction compared to the greedy policy
- Ensures deadlines are met

Implementation: Integrated into SkyPilot, an intercloud broker system

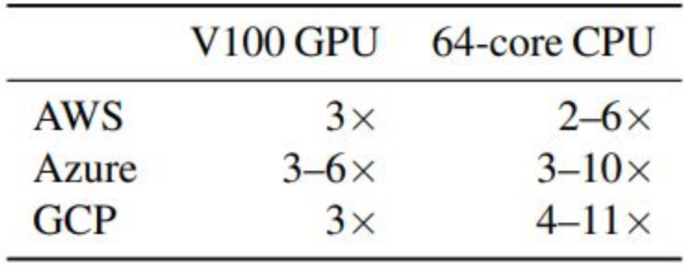
Results: Achieves 27%-84% cost savings across various real-world workloads.

Introduction

Spot instances vs on-demand instances

a. On-demand instances: always available but come at a high price

b. Spot instances: much cheaper but less available and can be preempted unexpectedly



	V100 GPU	64-core CPU
AWS	3×	2–6×
Azure	3–6×	3–10×
GCP	3×	4–11×

Table 1: Cost savings of spot vs. on-demand instances.

Introduction

How to enable an application to leverage spot instances while still meet its deadline?

A simple solution: use a spot instance up to the point at which the remaining computation time equals the remaining time to deadline, and then switch to an on-demand instance until it finishes. (greedy policy)

Introduction

Authors' solution: Uniform Progress

Highlights:

1. Parameter-free
2. Does not rely on assumptions about spot instance availability
3. Achieves 27%-84% cost savings across various real-world workloads.

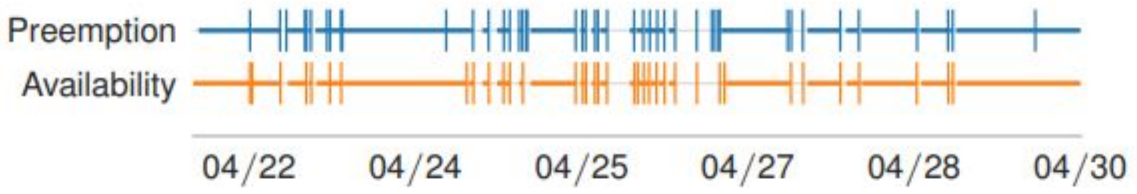
Characterization of Spot Instances

1. **Trace Collection**

Challenge: expensive

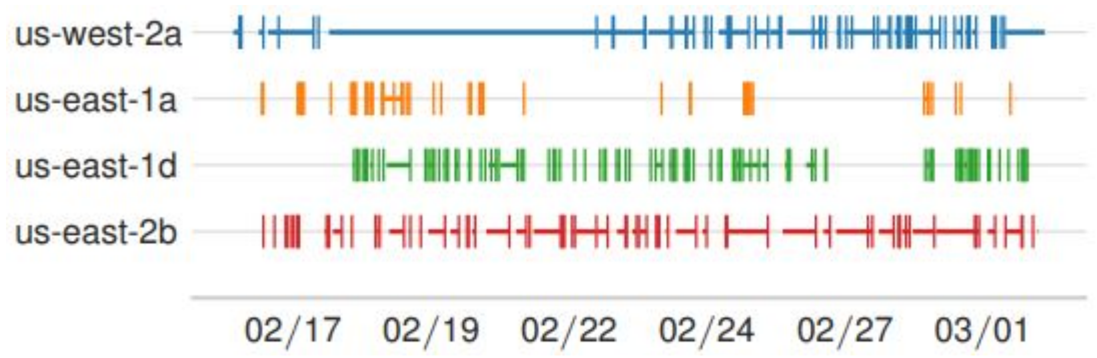
collect a real preemption trace for V100 instances for three-month long traces in 9 zones could cost over $10,000.

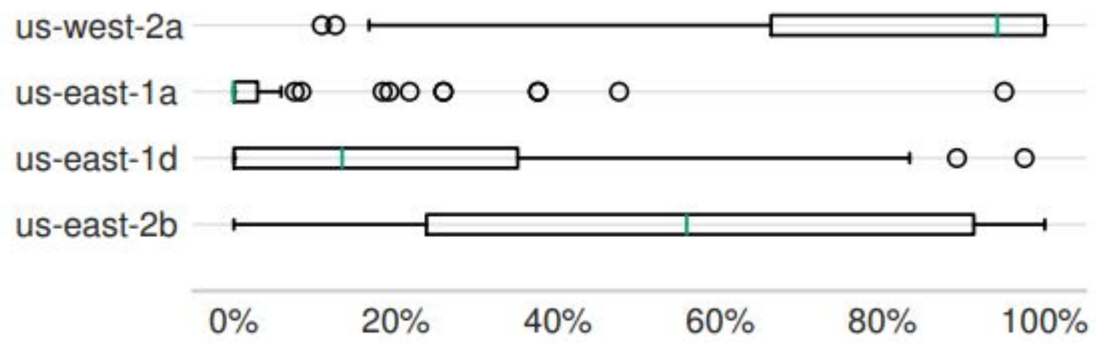
Solution: propose an approximation by collecting availability traces



Characterization of Spot Instances

High Variance in Spot Availability

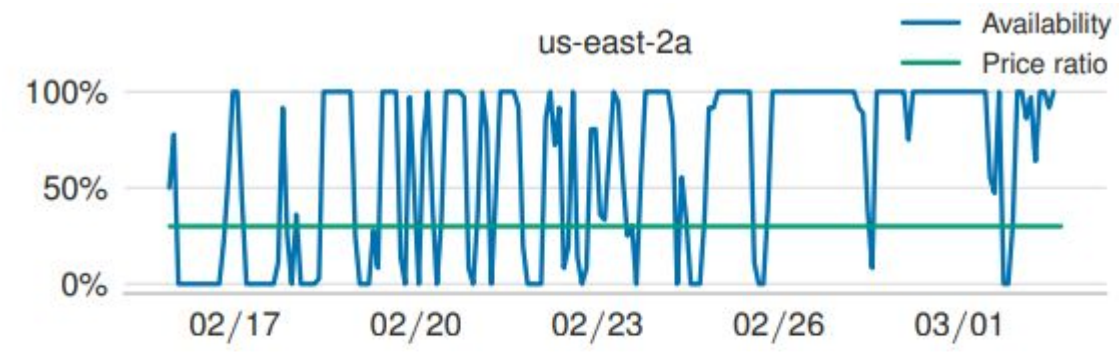




Boxplots of spot availability fraction, i.e., percentage of the time an instance is available in 6-hour windows.

Characterization of Spot Instances

High volatility



a highly volatile pattern: availability can change from 100% to 0% within hours.

However, spot pricing is much more stable.

Using Spot for Deadline-Sensitive Jobs

Problem Setup

$C(t)$: remaining computation time at time t

$R(t)$:remaining time-to-deadline at time t

$C(0)$: total computation time

$R(0)$: deadline

$R(t) = R(0) - t$

$\partial C(t) / \partial t = -1$

We assume that both $C(0)$ and $R(0)$ are given and the job is fault-tolerant to interruptions.

Using Spot for Deadline-Sensitive Jobs

Problem Setup

d: changeover delay, which includes the time required to launch an instance, set up dependencies, and recover any potential progress loss

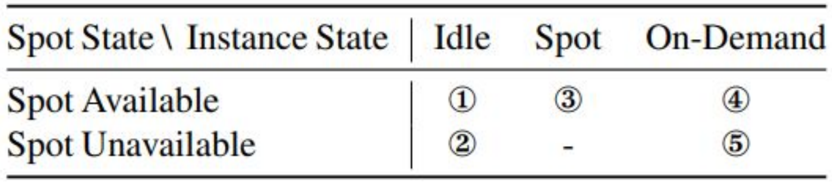
d is charged at the new instance type’s price.

The goal is to minimize the cost for completing job’s computation timeC(0) before deadline R(0). For simplicity, we define the price for an on-demand instance to be k > 1, and a spot instance to be 1 and both prices are fixed throughout the time before deadline R(0).

Using Spot for Deadline-Sensitive Jobs

Scheduling Policy

At any time t, a job can be in one of the following three states: idle, running on a spot instance, or running on an on-demand instance.



Spot State \ Instance State	Idle	Spot	On-Demand
Spot Available	①	③	④
Spot Unavailable	②	-	⑤

ideal case: $d = 0$

transition between state ② and ③, until $C(t) = R(t)$

Then transition between ③ and ⑤

optimal because it utilizes all available spot instance lifetimes before the deadline

Using Spot for Deadline-Sensitive Jobs

Scheduling Policy

Practical case: $d > 0$

Have to decide whether it is worth switching to a different instance at the expense of losing time d without making progress

Greedy policy:

Thrifty Rule: The job should remain idle after $C(t) = 0$.

Safety Net Rule: When a job is idle and $R(t) < C(t) + 2d$, switch to on-demand and stay on it until the end.

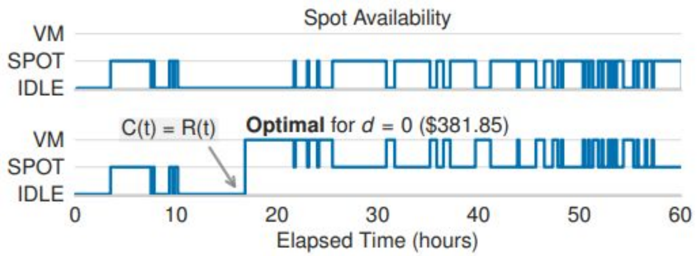
Exploitation Rule: Once start using a spot instance, stay on it until it is preempted.

Using Spot for Deadline-Sensitive Jobs

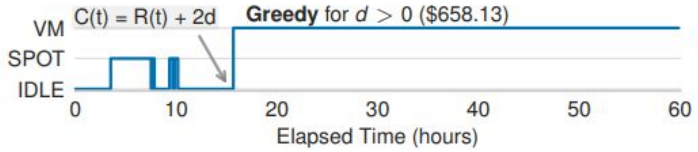
Scheduling Policy

The greedy policy behaves as follows:

1. Stay on any available spot instance until it is preempted (Exploitation Rule), and keep waiting if no spot instance is available
2. (Safety Net Rule) When $R(t)<C(t)+2d$ holds and the job is idle, move to on-demand and stay there until the end.



(a) Without changeover delay.



Can we do better than greedy while not assuming future knowledge?

Theoretical Analysis

Worst Case with Competitive Analysis

c: competitive ratio, which is the ratio of the cost of the policy to the best omniscient policy with full knowledge of future spot availability.

To simplify the presentation, we assume changeover delay d is small and ignore the term O(d). We use R(t)=C(t) + d as Safety Net Rule's condition, instead of R(t) = C(t) + 2d, which will not affect the conclusion, due to negligible O(d).

A natural bound for c is 1≤c≤k, where k can be reached when the oblivious adversary choose a case that a given policy have to use all on-demand, and the omniscient policy could use all spot instances.

Theoretical Analysis

Worst Case with Competitive Analysis

Theorem 1. For any deterministic policy P, $c \geq k - O(d)$.

As an adversary can simply make spot available from t’ , where $R(t') = C(t') + d$.

A policy has to be randomized to beat greedy, whose competitive ratio $c = k$.

Theoretical Analysis

Worst Case with Competitive Analysis

A better policy: randomized shifted greedy (RSF) policy

we divide the time into n even slices with length R(0)/n and apply greedy in each of these slices with C(0)/n progress to make.

In each slice, the policy enforces the job to make ≥$\frac{C(0)}{n}$ units of progress within $\frac{R(0)}{n}$

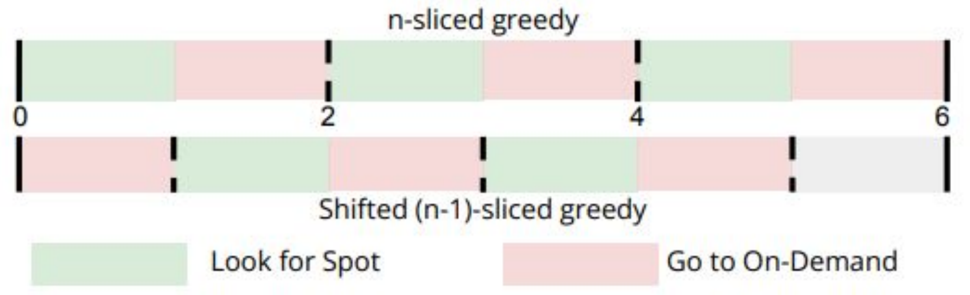
We then shift the n-sliced greedy policy by $\frac{C(0)}{n}$, to get **shifted (n−1)-sliced greedy policy**

It uses on-demand for time $\frac{C(0)}{n}$ from start, then applies (n−1)-sliced greedy from t = $\frac{C(0)}{n}$ until $t = R(0) - \frac{R(0) - C(0)}{n}$

Theoretical Analysis

Worst Case with Competitive Analysis

A better policy: randomized shifted greedy (RSF) policy



We can define a randomized shifted greedy (RSF) policy by using either the n-sliced or the shifted (n − 1)-sliced greedy with equal probability at any time t. We can prove that the competitive ratio for RSF is bounded and lower than greedy.

$$Theorem\,2.\,If\,R(0) \geq 2C(0),\,then\,for\,RSF\,policy\,has\,c \leq \frac{k+1}{2} + \frac{k-1}{2n} + O(d) < k.$$

Theoretical Analysis

Worst Case with Competitive Analysis

A better policy: randomized shifted greedy (RSF) policy

For R(0) ≤ 2C(0), we can simply use on-demand until R(t) = 2C(t) then start using RSF policy. We denote this modified RSF (MRSF) policy.

$$Corollary\,1.\,Let\,a = \frac{R(0)}{c(0)} - 1\,for\,0 \leq a \leq 1.\,MRSF\,policy\,has:$$

$$c \leq k - ak + a\left(\frac{k+1}{2} + \frac{k-1}{2n}\right) + O(d) = k - \frac{a(k-1)(n-1)}{2n} + O(d) < k$$

With MRSF policy, we shown that there exists a policy that performs better than greedy for any R(0), C(0) in worst cases by randomization and distributing job progress.

Theoretical Analysis

Average Case with Stochastic Model

In order to model the spot process, we consider a smoothed version where we assume that a fractional spot is always available, with a ratio $r < 1$, i.e., a job running on the fractional spot makes r amount of progress per unit of time.

Similarly, for simplicity, we assume that d is relatively small and ignore terms of $O(d)$.

Theoretical Analysis

Average Case with Stochastic Model

For greedy policy, It will use the fractional spot until R(t′) = C(t′) + O(d) and then switch to on-demand.

At time t′ , the job progress on the fractional spot would beC(0) − C(t′) = rt′ − O(d)

C(t′) = C(0) − rt′ +O(d), and the remaining time would be R(t′) = R(0) − t′. We can derive t′ and expected payment (total cost) p:

$$R(t') = C(t') \implies R(0) - t' = C(0) - rt' + O(d) \quad (1)$$

$$t' = \frac{R(0) - C(0) + O(d)}{1 - r} \quad (2)$$

$$p = rt' + (R(0) - t')k + O(d) = (r - k)t' + kR(0) + O(d) \quad (3)$$

Theoretical Analysis

Average Case with Stochastic Model

For greedy policy, It will use the fractional spot until R(t′) = C(t′) + O(d) and then switch to on-demand.

At time t′ , the job progress on the fractional spot would beC(0) − C(t′) = rt′ − O(d)

C(t′) = C(0) − rt′ +O(d), and the remaining time would be R(t′) = R(0) − t′. We can derive t′ and expected payment (total cost) p:

$$R(t')=C(t') \Longrightarrow R(0)-t'=C(0)-rt'+O(d) \qquad (1)$$

$$t'=\frac{R(0)-C(0)+O(d)}{1-r} \qquad (2)$$

$$p=rt'+(R(0)-t')k+O(d)=(r-k)t'+kR(0)+O(d) \qquad (3)$$

We can observe that the payment depends on the fractional spot ratio r. Since r−k < 0, payment p reduces when the time t′ spent on the fractional spot increases.

Theoretical Analysis

Average Case with Stochastic Model

For n-sliced greedy policy, when we started considering the expected payment across difference traces, variance for fractional spot involves.

Consider spot fraction R as a random variable with mean r and variance v. We can prove that the expected time on the fractional spot E[t′] increases with the variance v.

Difference of n-sliced (with variance $\hat{v}$) to original greedy (with variance v):

$$\Delta = \frac{R(0) - C(0)}{(1-r)^3}(\hat{v} - v)$$

Since Δ > 0, We can conclude that n-sliced greedy has larger E[t′], leading to a lower expected cost p than original greedy in average case.

Methodology

Time Sliced

Based on the n-sliced greedy policy, we propose the Time Sliced policy.

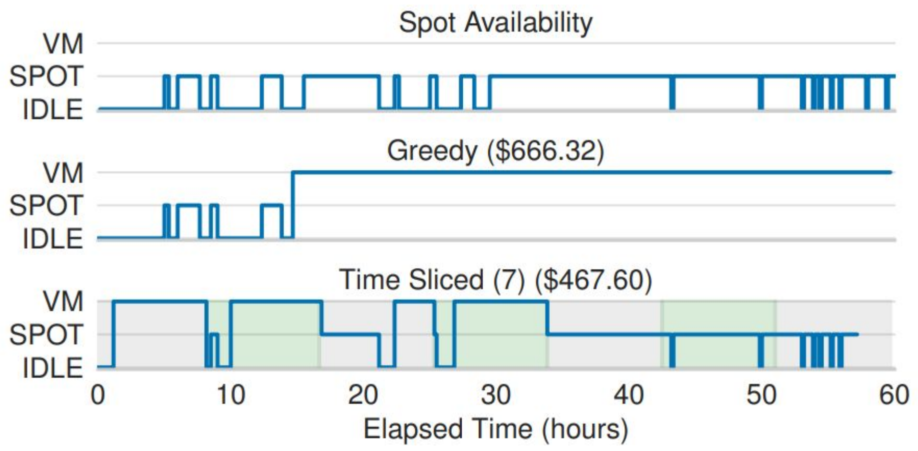
We divide the time before deadline, R(0), into slices, and assign each slice a computation time C(0)/n and deadline R(0)/n, denoted as Ci and Ri for slice i. In each time slice, we apply greedy policy.

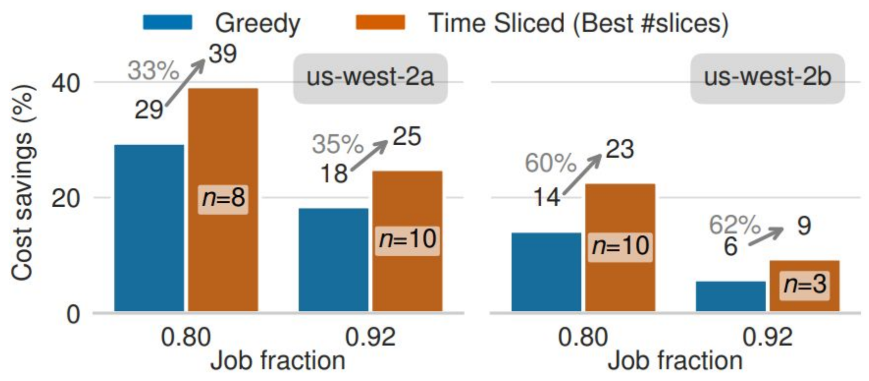
Two changes compared to the n-sliced greedy policy:

1. jobs can continue on spot instances whenever available after Ci(t)≤0,
2. if a slice makes more progress than required, we reduce the required computation in the succeeding slice,Ci+1.

Methodology

Time Sliced





Methodology

Uniform Progress

Although Time Sliced policy with the best slice number n outperforms greedy, selecting the optimal n for different cases is not practical.

At the end of a slice i, ti = $i\frac{R(0)}{n}$, i.e., i = $t_i\frac{n}{R(0)}$. The current progress, cp(ti) = C(0) − C(ti), is guaranteed to meet the expected progress, ep(ti):

$$cp(t_i)\geq ep(t_i)=i\frac{C(0)}{n}=t_i\frac{C(0)}{R(0)} \qquad (4)$$

When the slice number n=1, Time Sliced becomes greedy policy. When more slices involve, with larger n, (4) applies to more time steps. We adapt this idea into Time Sliced by pushing n □ ∞, making each slice infinitesimal. That enforces (4) at any t ≤R(0), i.e., fully distributing progress within the deadline:

$$cp(t)\geq ep(t)=t\frac{C(0)}{R(0)},\forall t\leq R(0) \qquad (5)$$

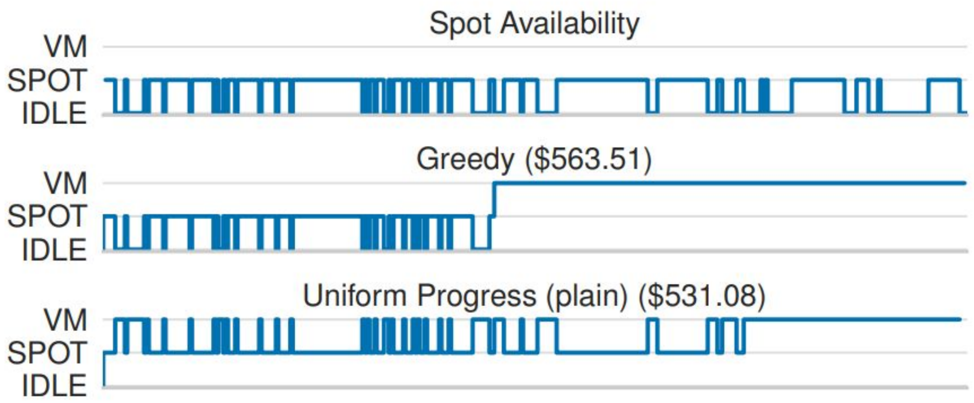
COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Methodology

Uniform Progress Policy

Uniform Progress (plain), that has the following rules:

1. Uniform Progress: When the job is idle and cp(t)<ep(t), switch to on-demand and stay on it to catch up progress.
2. Taking Risks: Switch to spot whenever it is available (even when cp(t) < ep(t)). Stay on the spot until it is preempted (Exploitation Rule).

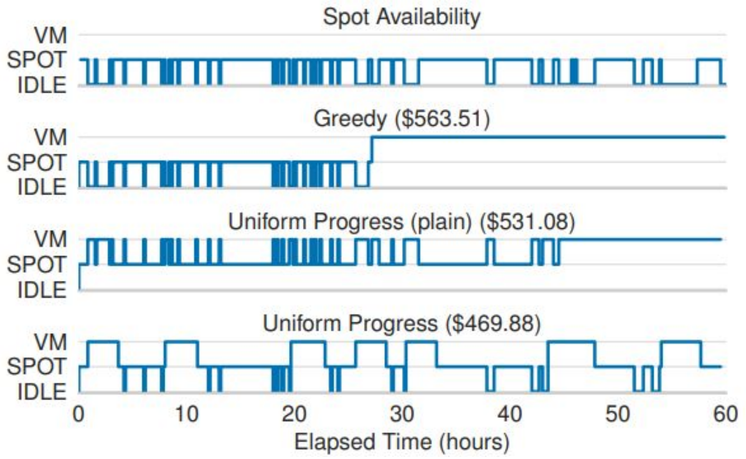


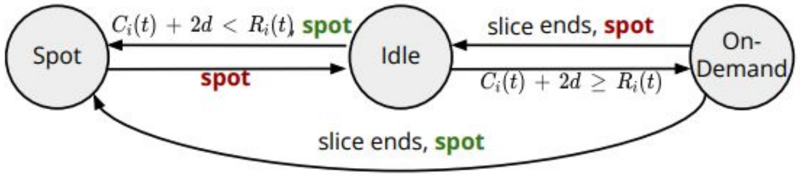
Methodology

Uniform Progress Policy

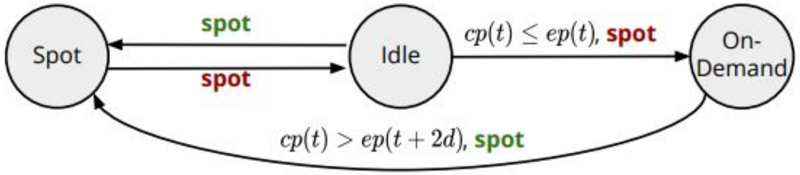
Uniform Progress policy:

Hysteresis: When the job is on on-demand, stay on it until cp(t)≥ep(t+2d).





(a) Time Sliced



(b) Uniform Progress

Methodology

Some other policies:

Omniscient Policy:

We define some binary variables:

- a(t) whether a spot instance is available at time t.
- s(t), v(t) indicate the policy choose to use a spot/ondemand instance at time t.
- x(t), y(t) represent changeover delays happen to a spot/on-demand instance at time t.

$$\min_{s(t),v(t)} \sum_{t=0}^{R(0)} [s(t)+v(t)k] \tag{6}$$

$$\forall t,\ s(t)+v(t) \leq 1,\ s(t) \leq a(t) \tag{7}$$

$$\sum_{t=0}^{R(0)} [s(t)+v(t)] \geq d \sum_{t=1}^{R(0)} (x(t)+y(t)) + C(0) \tag{8}$$

$$\forall t,\ x(t) \leq s(t),\ x(t) \leq 1-s(t-1),\ x(t) \geq s(t)-s(t-1) \tag{9}$$

$$\forall t,\ y(t) \leq v(t),\ y(t) \leq 1-v(t-1),\ y(t) \geq v(t)-v(t-1) \tag{10}$$

Methodology

Some other policies:

Partial Lookahead Omniscient Policy:

It has limited foresight into future spot availability. By partitioning the deadline into n slices, it can only see complete availability within each slice.

Next Spot Lifetime Oracle:

Cloud providers offer an oracle o(t) that returns the lifetime of the next spot instance a job can acquire at the current time t.

Methodology

Extending to Multiple Instances:

Polarization Rule: For a job requiring $N > 1$ instances, a policy should either use no instance or N instances at any time.

We now extend previous policies to multiple instances.

The action space for a policy is simplified to either: N spot, N on-demand, or no instances at any time t. The problem for multiple instances is now equivalent to the single instance, with the one-to-one mapping of states.

Methodology

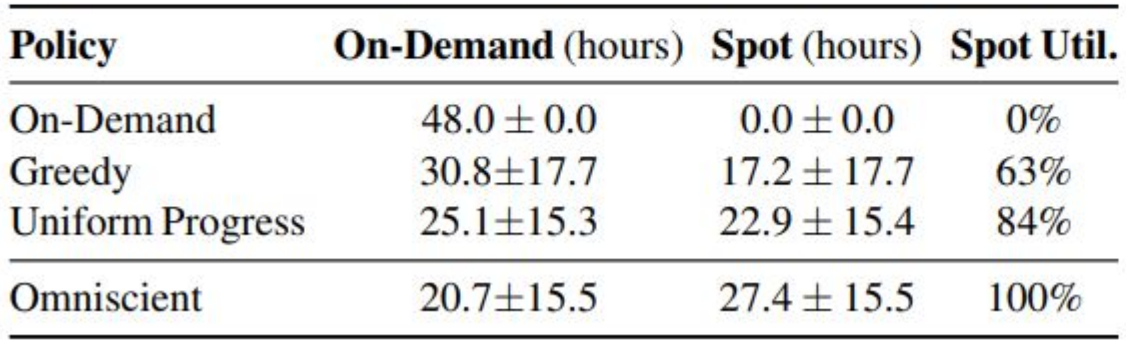
Relaxing Computation Time and Changeover Delay

Computation time: Given that no policy can predict C(0) precisely beforehand, we adjust the deadline guarantee of the policies to be best effort, ensuring a finish time within the original deadline plus the difference.

Changeover delay: We assume that no policies can foresee the exact changeover delay until its occurrence, though the average changeover delay is given. If a user would like to ensure the original deadline with a given maximum changeover delay, they can specify a new deadline.

Evaluation

Time Spent on On-demand and Spot Instances



Policy	On-Demand (hours)	Spot (hours)	Spot Util.
On-Demand	48.0 ± 0.0	0.0 ± 0.0	0%
Greedy	30.8 ± 17.7	17.2 ± 17.7	63%
Uniform Progress	25.1 ± 15.3	22.9 ± 15.4	84%
Omniscient	20.7 ± 15.5	27.4 ± 15.5	100%

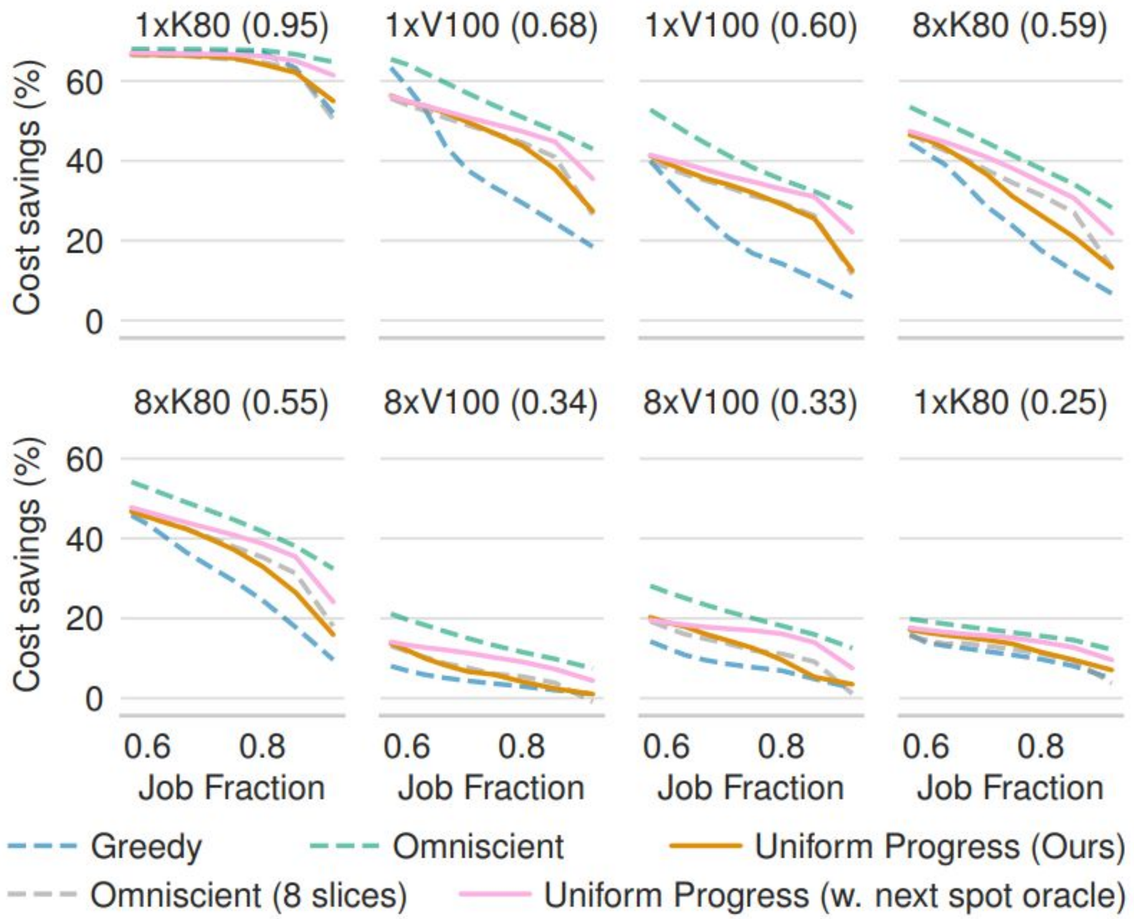
The results with a fixed job fraction C(0) R(0) = 0.8 on the 2-week traces, averaging across eight (instance type, availability zone) pairs, each with 300 randomly sampled traces.

Evaluation

Various Deadlines

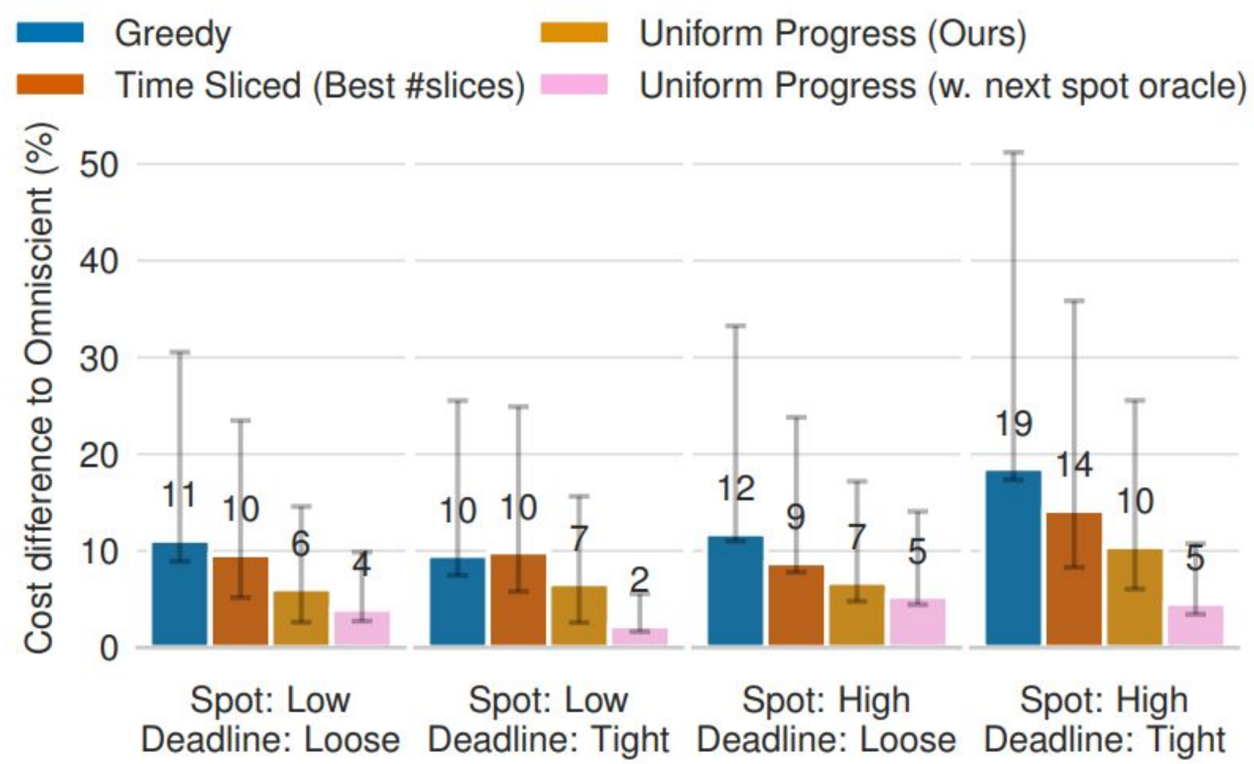
Uniform Progress achieves similar performance in most cases, despite lacking future knowledge.

This suggests any other policy without future knowledge may not yield much higher savings.



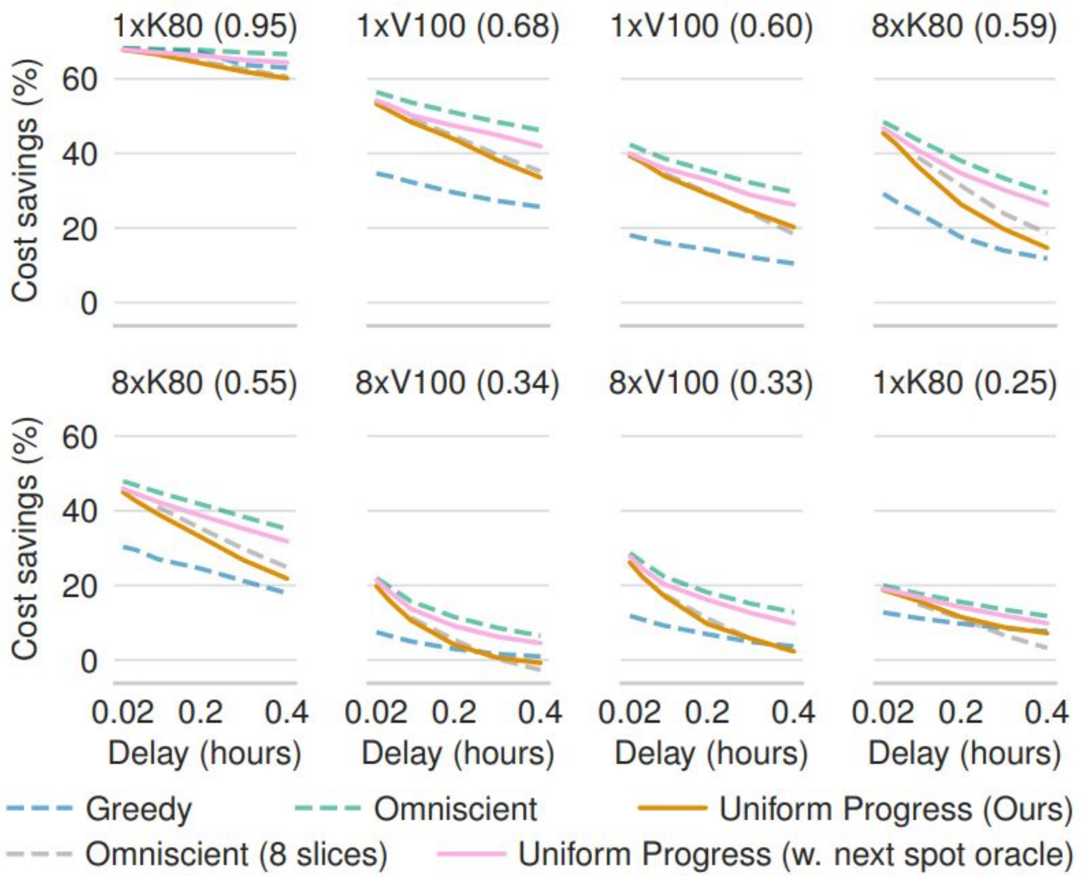
Evaluation

Impact of Spot Fraction and Deadline



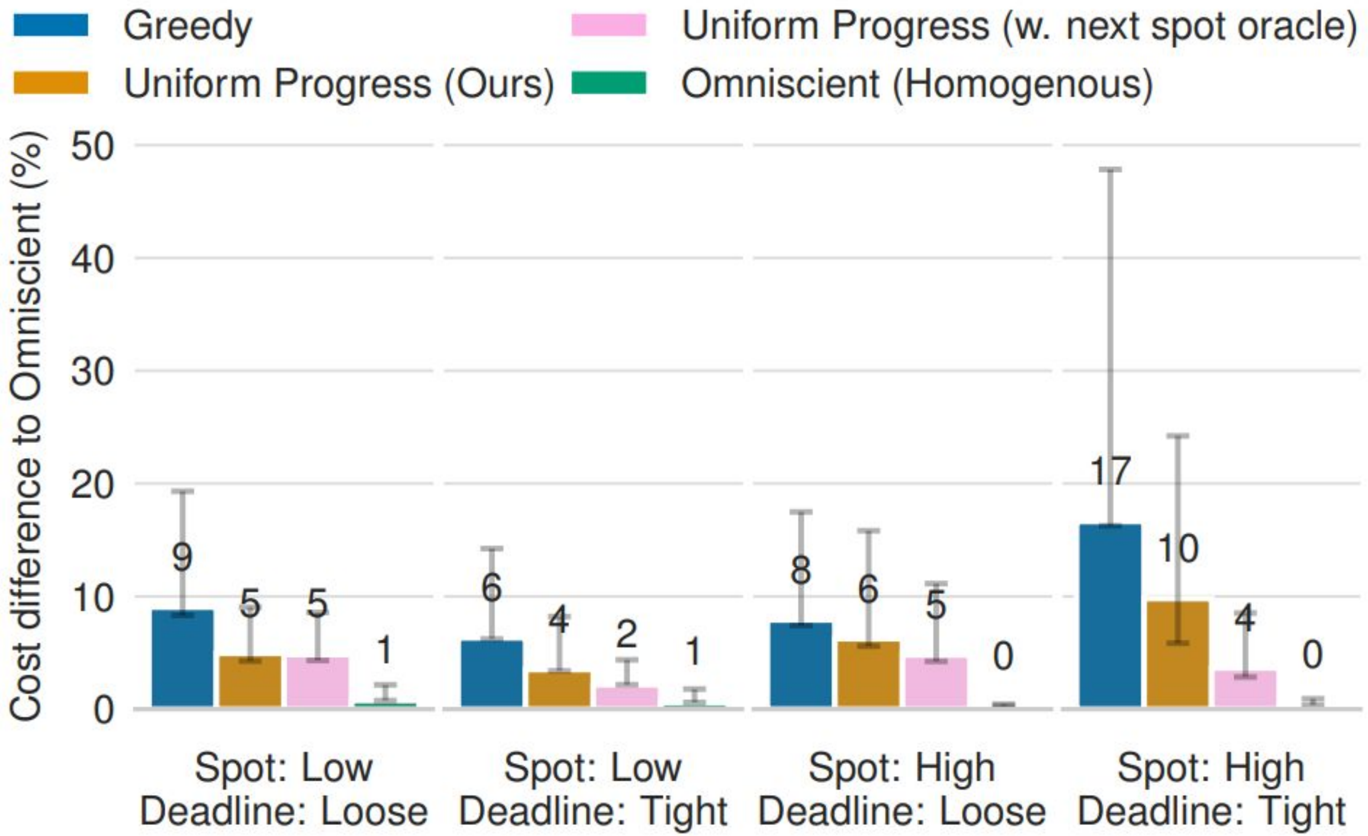
Evaluation

Different Changeover Delays



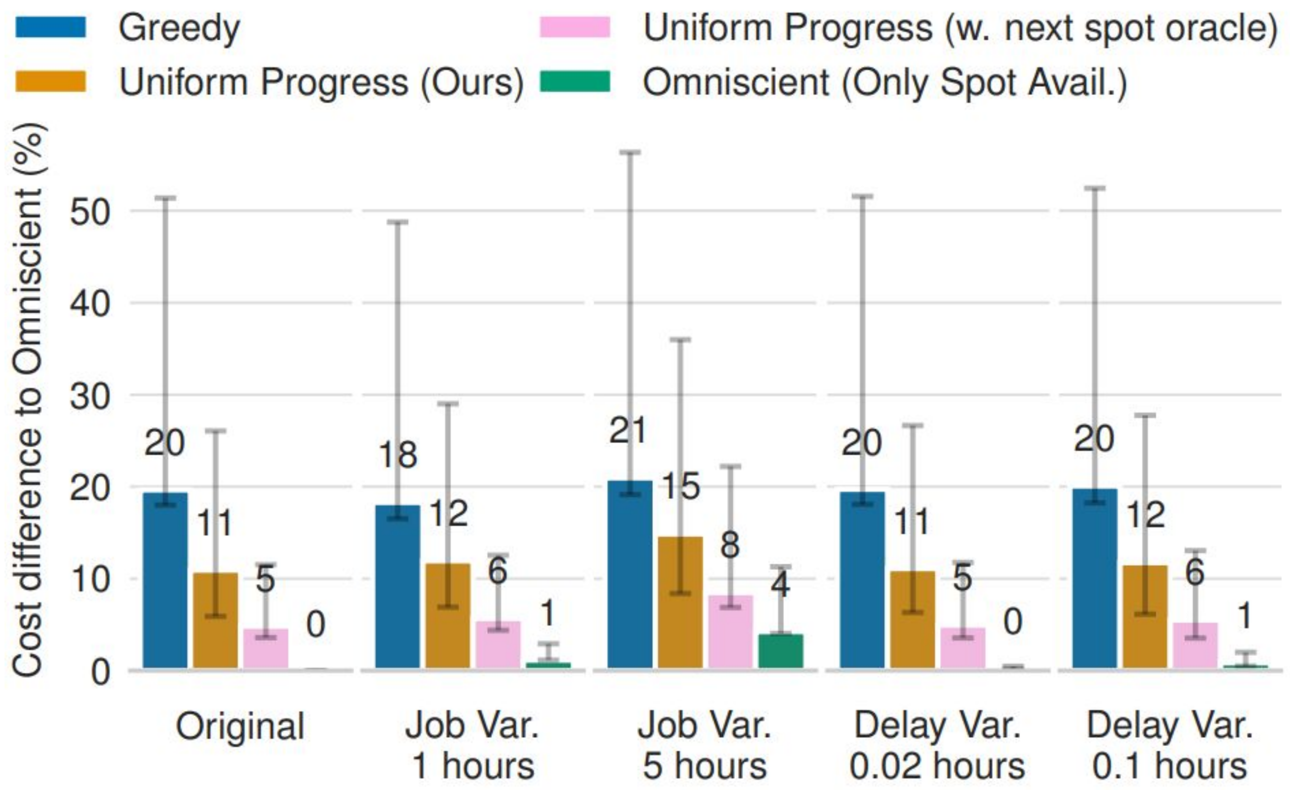
Evaluation

Multiple Instances



Evaluation

Relaxed Computation Time and Changeover Delay



Practical Usage

Workload	Location	Instance Type	Spot Price (Discount)	Computation	Deadlines	Changeover Delay
ML Training	AWS (us-west-2b)	p3.2xlarge	\$0.92/hr (-67%)	72 hrs	84/100 hrs	4+5+9 mins ≈0.3 hrs
Bioinformatics	GCP (us-east1-b)	c3-highcpu-88	\$0.34/hr (-91%)	22.5 hrs	24/28 hrs	2+1+8 mins ≈0.2 hrs
Data Analytics	AWS (us-east-1c)	r5.16xlarge	\$1.85/hr (-55%)	27 hrs	30/36 hrs	4+1+7 mins ≈0.2 hrs

Table 4: Detailed characteristics of real workloads. Deadlines are derived from job fractions 90% and 75%, and changeover delays are the sum of VM provisioning, environment setup, and job recovery progress loss time.

		Uniform Progress	
Workload	**On-demand**	Tight DDL (0.9)	Loose DDL (0.75)
ML	\$233.5	\$138.2 (**-41%**)	\$122.0 (**-48%**)
Bioinfo	\$140.5	\$51.9 (**-63%**)	\$22.8 (**-84%**)
Analytics	\$109.6	\$80.0 (**-27%**)	\$74.1 (**-32%**)

Table 5: Cost savings for real workloads. Results of two deadlines are shown (job fractions 0.9 and 0.75).

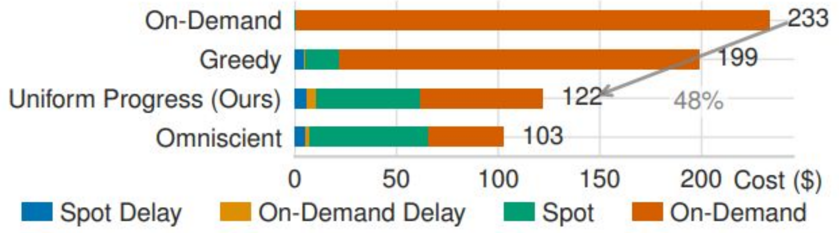


Figure 16: Cost breakdown of each policy for ML workload.

Thanks for listening!