东南大学
SOUTHEAST UNIVERSITY

计算机科学与工程学院
School of computer science and engineering

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

NSDI'24 outstanding paper

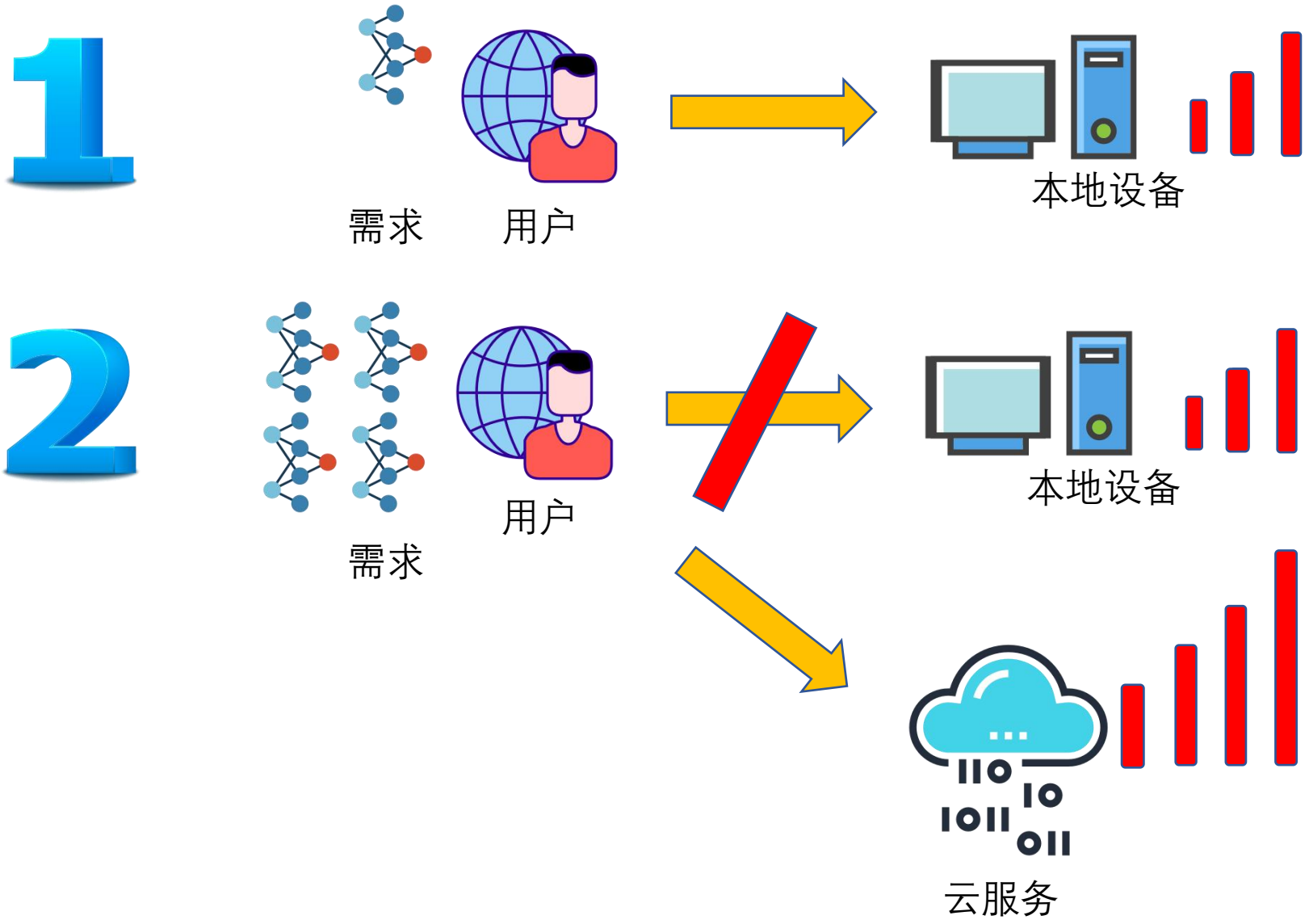
Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, and Zongheng Yang, *University of California, Berkeley;* Eric Friedman and Scott Shenker, *University of California, Berkeley, and ICSI;* Ion Stoica, *University of California, Berkeley*

汇报人：王维龙　2024 年 12 月 20 日

提纲

提纲

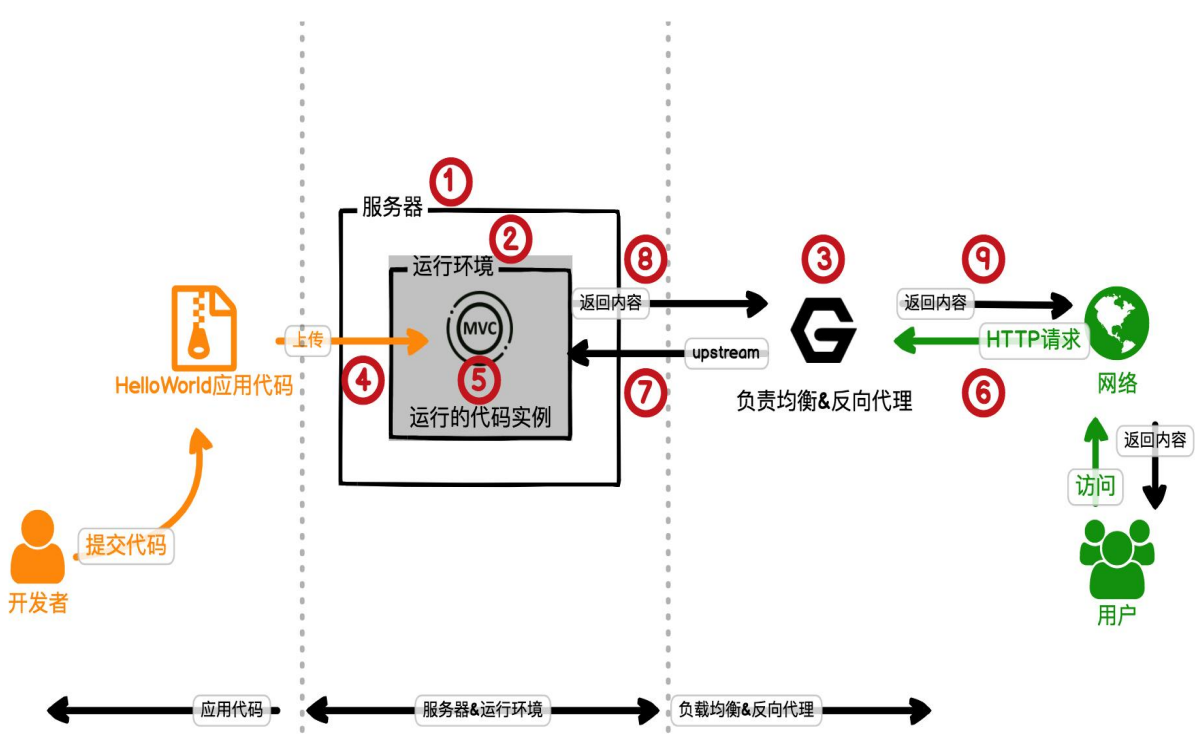
背景概况



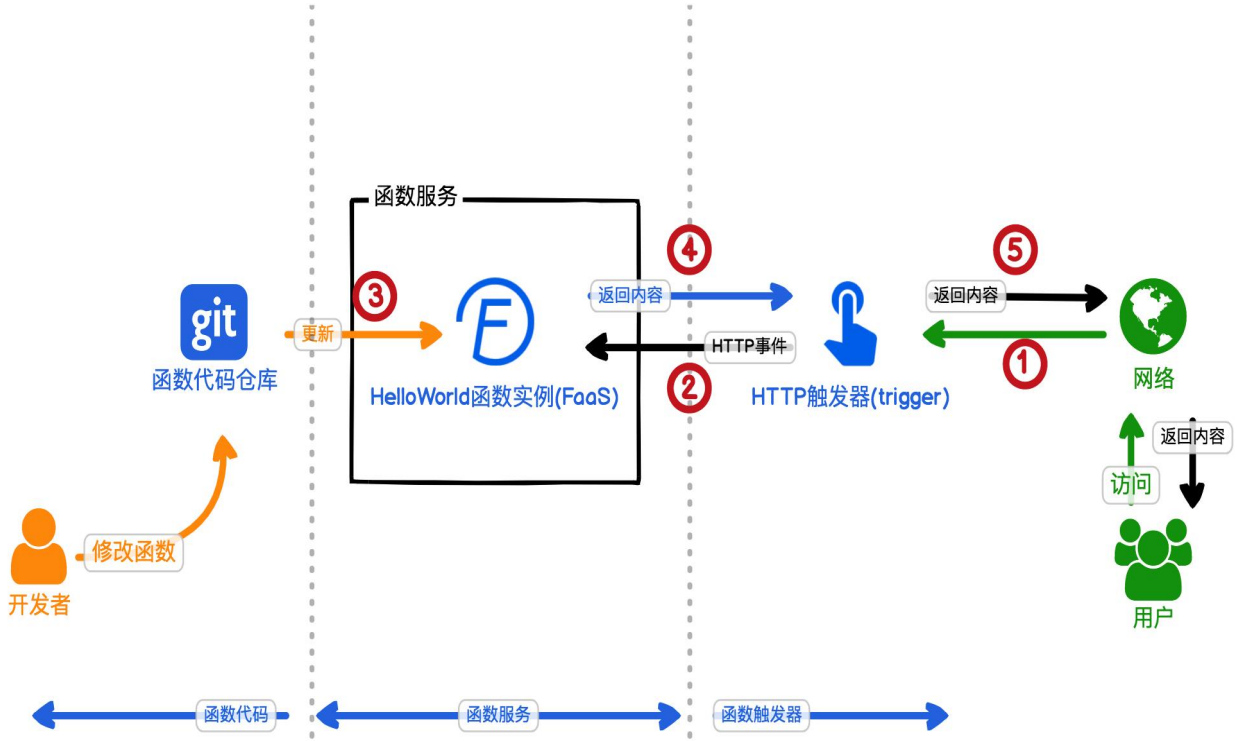
背景概况

云服务中存在许多架构，其中有一种特殊的服务框架叫Serverless或叫服务器无感知框架

Serverless具体是什么呢?

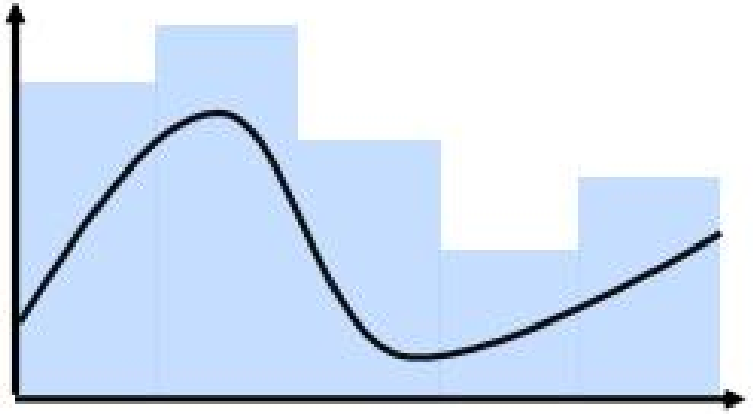


传统云部署(虚拟机, VM)

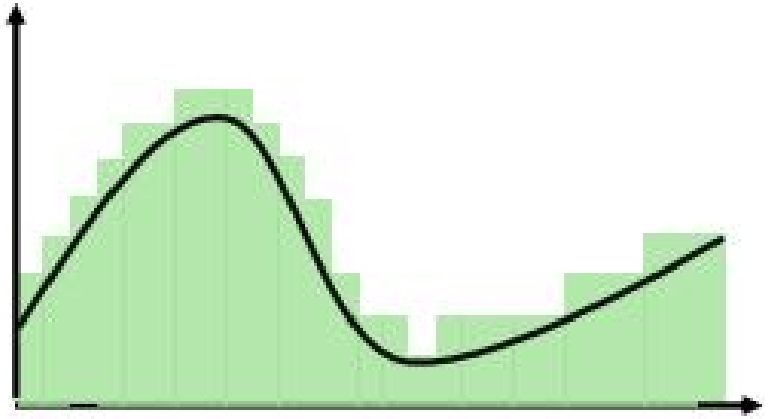


Serverless部署

背景概况



传统云部署

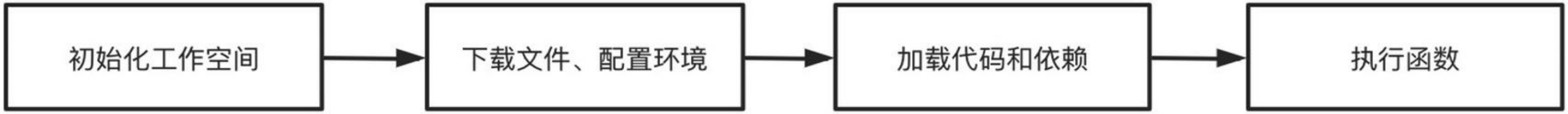


Serverless部署

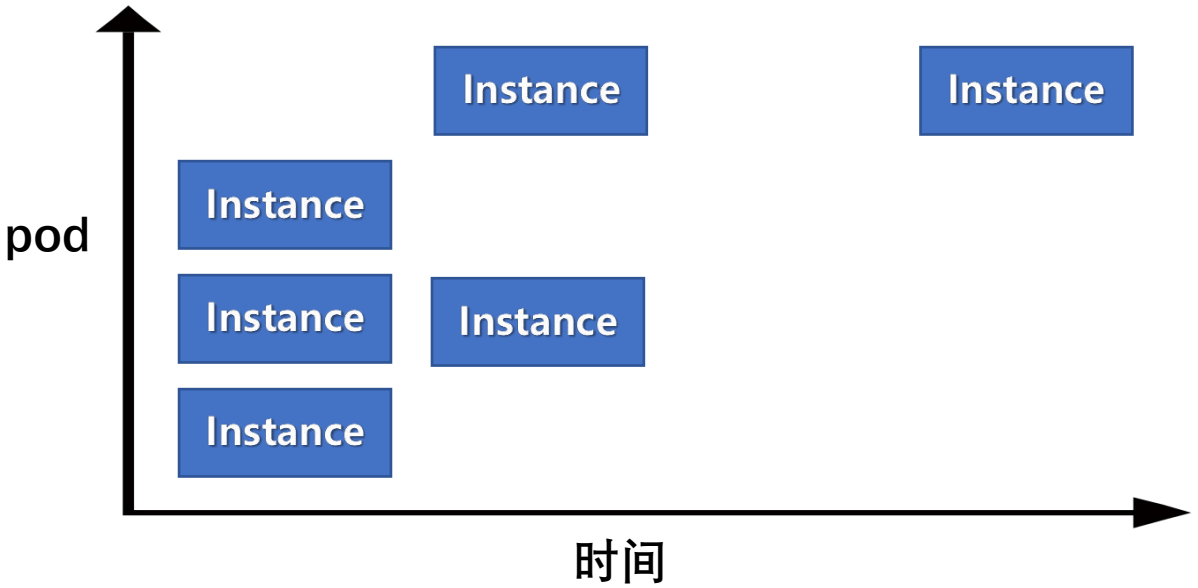
- 基于虚拟机(VM)隔离应用(分钟级)
- 按时间收费
- 扩缩容能力差，反应能力慢
- 需要手动构建运行环境

- 基于实例(Instance)隔离应用(秒级)
- 按使用收费(pay-as-you-go)
- 自动扩缩容，效率高
- 自带运行环境，无需配置

背景概况



serverless服务执行流程

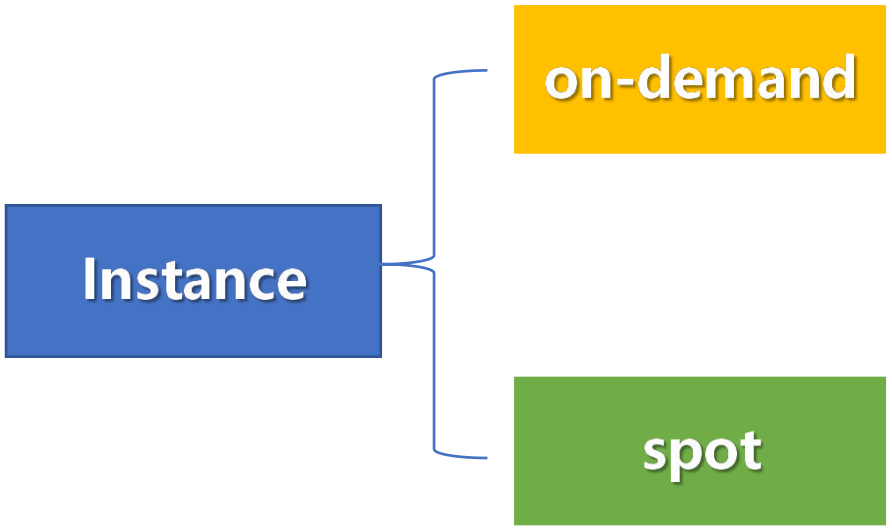


Serverless自动扩缩容

SLO, (Service Level Object) 服务水平目标

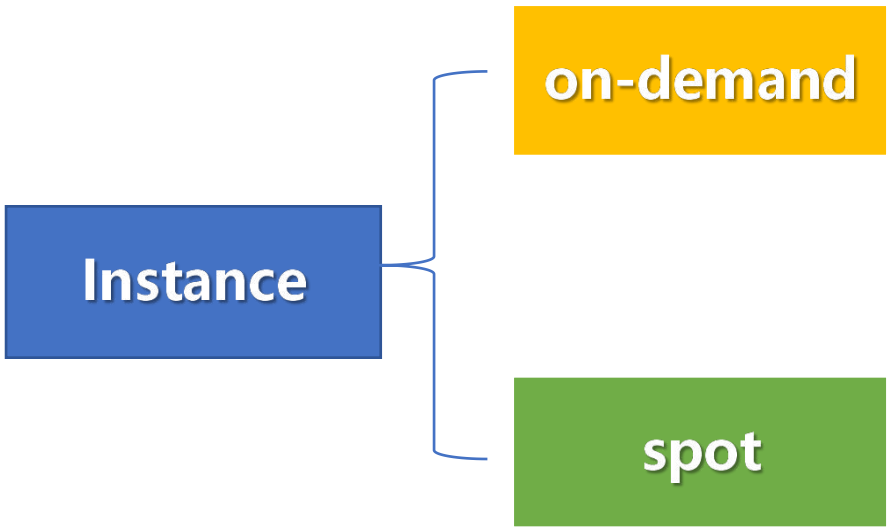
在Serverless中大多数的SLO设定为时延目标，即，使用SLO违规率表示，如果超过规定的SLO时延，则为违规

背景概况



背景概况

计算成本



全价

折扣价

	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.

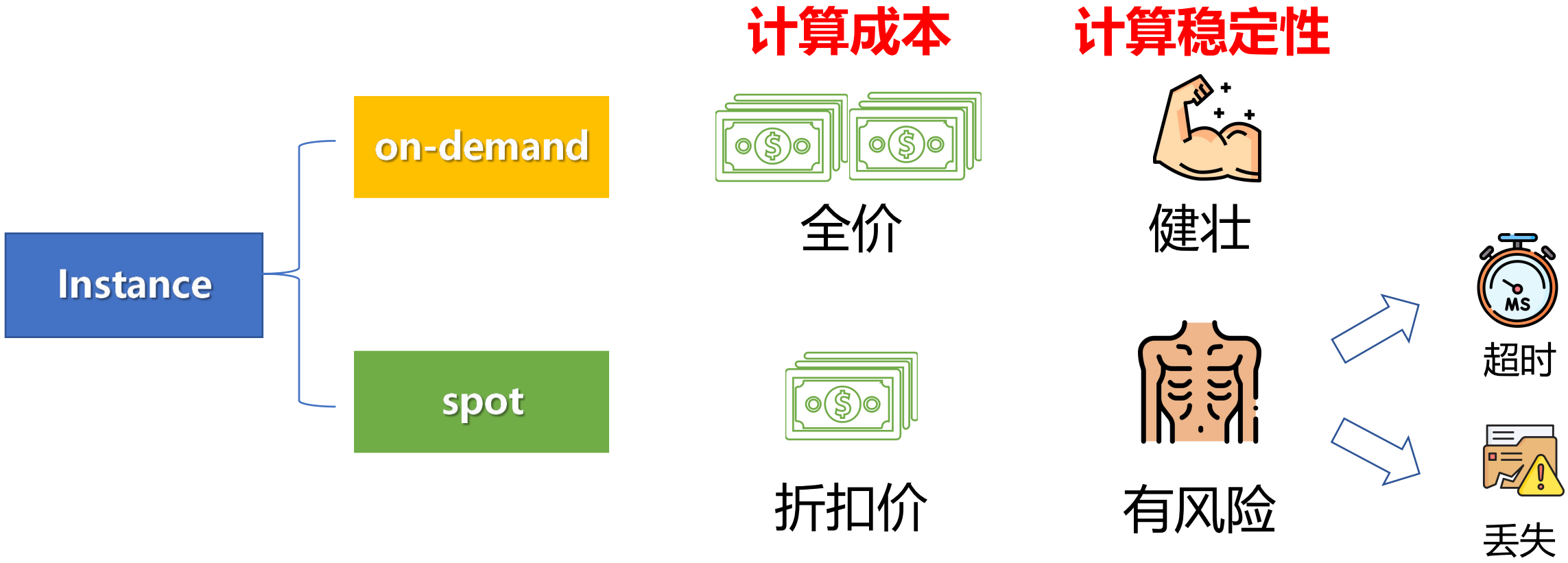
因为计算需求大部分机器学习相关工作，所以将这两类计算资源作为案例

1. on-demand Instance的价格 ≥ spot Instance的价格

背景概况

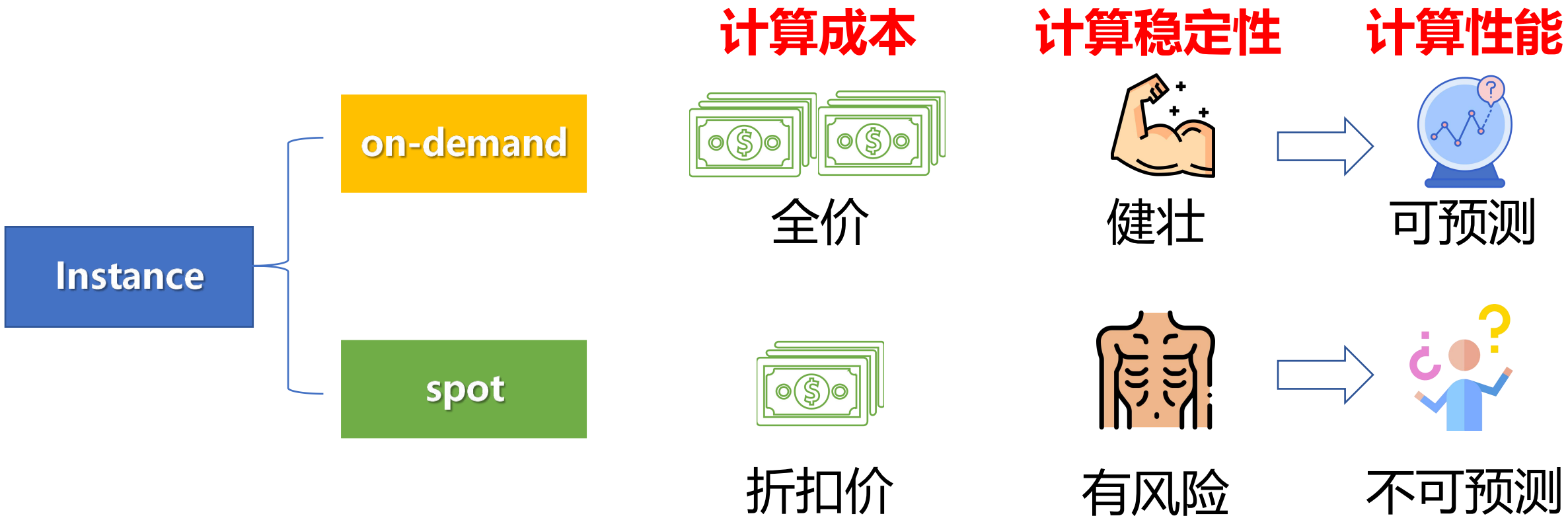
1. on-demand Instance的价格 ≥ spot Instance的价格
2. spot Instance可能存在计算中断(被抢占)的风险

背景概况



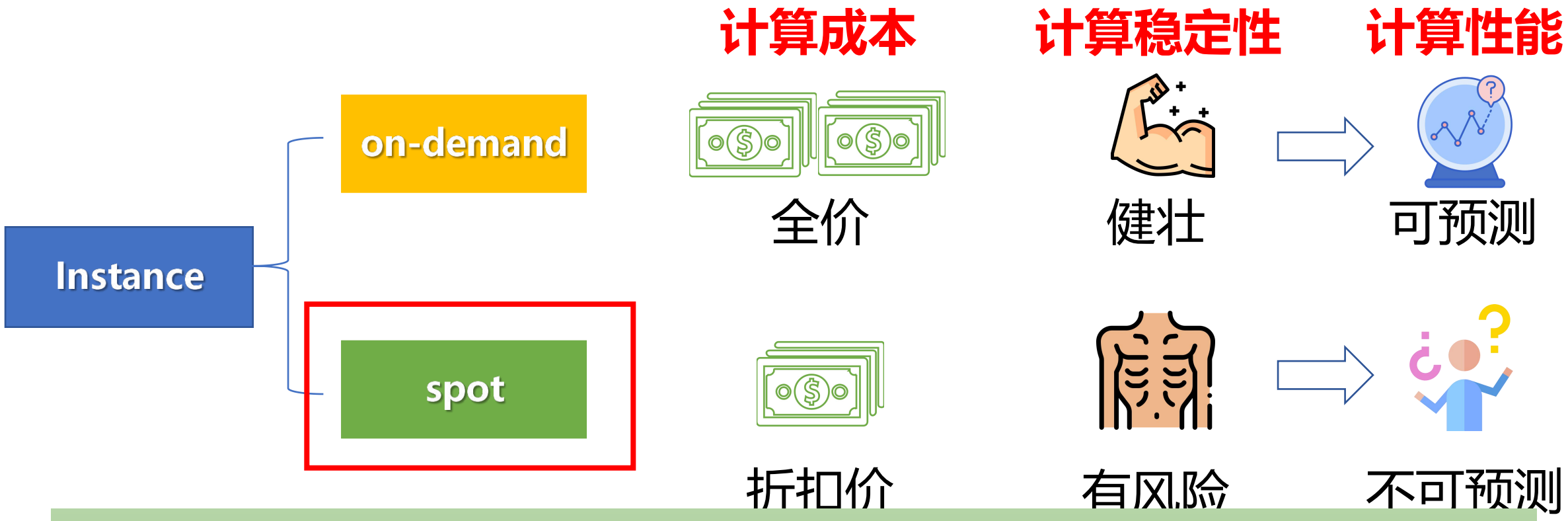
1. on-demand Instance的价格 ≥ spot Instance的价格
2. spot Instance可能存在计算中断(被抢占)的风险

背景概况



1. on-demand Instance的价格 ≥ spot Instance的价格
2. spot Instance可能存在计算中断(被抢占)的风险

背景概况



那要如何准确使用spot呢？或者说在不同状态如何选择spot或者on-demand?

2. spot Instance可能存在计算中断(被抢占)的风险

提纲

相关工作

Spot定价和可用性建模。AWS在2009年率先推出了spot Instance，使用招标机制将未使用的云容量货币化。随着其他云提供商采取了类似的策略，定价模式逐渐发展。虽然现货定价相对稳定，但由于其**黑盒**性质，建模spot可用性仍然具有挑战性。**虽然之前的工作试图对抢占模式进行建模，并采用了ML预测方法，但结果并不理想。**

相关工作

Spot定价和可用性建模。AWS在2009年率先推出了spot Instance，使用招标机制将未使用的云容量货币化。随着其他云提供商采取了类似的策略，定价模式逐渐发展。虽然现货定价相对稳定，但由于其**黑盒**性质，建模spot可用性仍然具有挑战性。**虽然之前的工作试图对抢占模式进行建模，并采用了ML预测方法，但结果并不理想。**

使用spot instance的应用程序。spot的成本效益性质促使其采用以节省成本。像**Bamboo**、**Spotnik**和**Srifty**等框架，都是为机器学习使用spot instance而开发的。Narayanan等人显示使用跨多个云的spot instance显著降低机器学习训练成本。计算机缓存利用spot instance进行内存中的数据缓存。然而，**抢占可能会对应用程序的性能产生负面影响，而受最后期限限制的应用程序可能难以有效地利用spot instance。**

相关工作

Spot定价和可用性建模。AWS在2009年率先推出了spot Instance，使用招标机制将未使用的云容量货币化。随着其他云提供商采取了类似的策略，定价模式逐渐发展。虽然现货定价相对稳定，但由于其**黑盒**性质，建模spot可用性仍然具有挑战性。**虽然之前的工作试图对抢占模式进行建模，并采用了ML预测方法，但结果并不理想。**

使用spot instance的应用程序。spot的成本效益性质促使其采用以节省成本。像**Bamboo**、**Spotnik**和**Srifty**等框架，都是为机器学习使用spot instance而开发的。Narayanan等人显示使用跨多个云的spot instance显著降低机器学习训练成本。计算机缓存利用spot instance进行内存中的数据缓存。然而，**抢占可能会对应用程序的性能产生负面影响，而受最后期限限制的应用程序可能难以有效地利用spot instance。**

作业调度与先发制人（抢占）。在抢占设备上运行的作业是在间歇系统上研究的，在这些系统中，**作业可能由于零星的可收获能源而中断。**由于这些设备的计算资源有限，许多研究集中在调度多个实时物联网任务。spot instance引入资源抢占要求在云上的批处理作业。从**云提供商**的角度来看，现有的工作研究如何**最大化收入，或增强运行时保证。**对于**最终用户**，早期的研究探索了**基于最后期限包任务的投标策略**，但由于**定价模式**的变化，**这些方法不适用于当前的现货市场。**最近，Snape对**长期运行**的服务**混合使用spot和on-demand instances**进行了调查。**它针对SLO进行优化，这要求可用的Instance数随时接近目标数量。**而本文研究的**截止日期敏感工作**，在本文中，作业可以长时间闲置，只要能满足截止日期。

Spot Instance数据采集

在三个月内。对九个AWS可用区收集这些跟踪 (three in us-west-2, four in us-east-1, two in us-east-2).

Spot Instance数据采集

在三个月内。对九个AWS可用区收集这些跟踪 (three in us-west-2, four in us-east-1, two in us-east-2).

一个spot V100 Instance的成本约为1美元/小时

Spot Instance数据采集

在三个月内。对九个AWS可用区收集这些跟踪 (three in us-west-2, four in us-east-1, two in us-east-2).

一个spot V100 Instance的成本约为1美元/小时

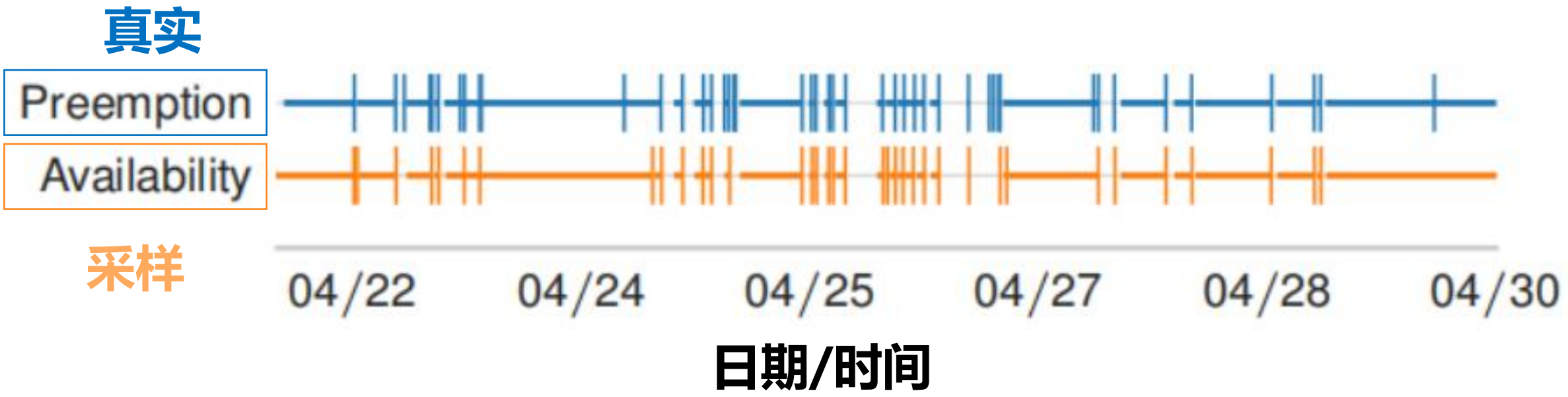
$10,000

Spot Instance数据采集

每十分钟测试一次可用性，然后紧急停止

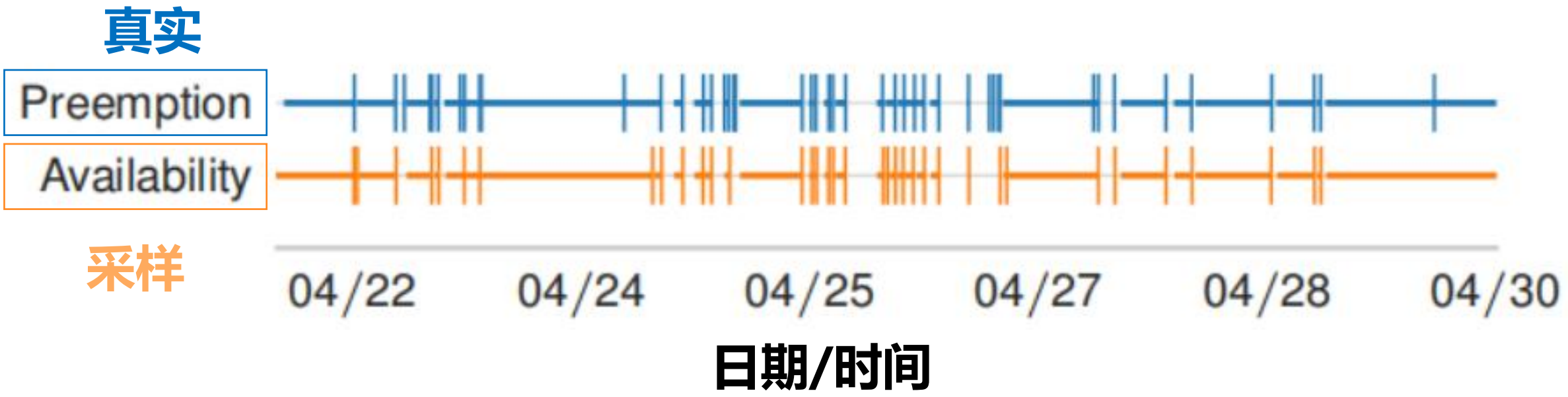
Spot Instance数据采集

每十分钟测试一次可用性，然后紧急停止



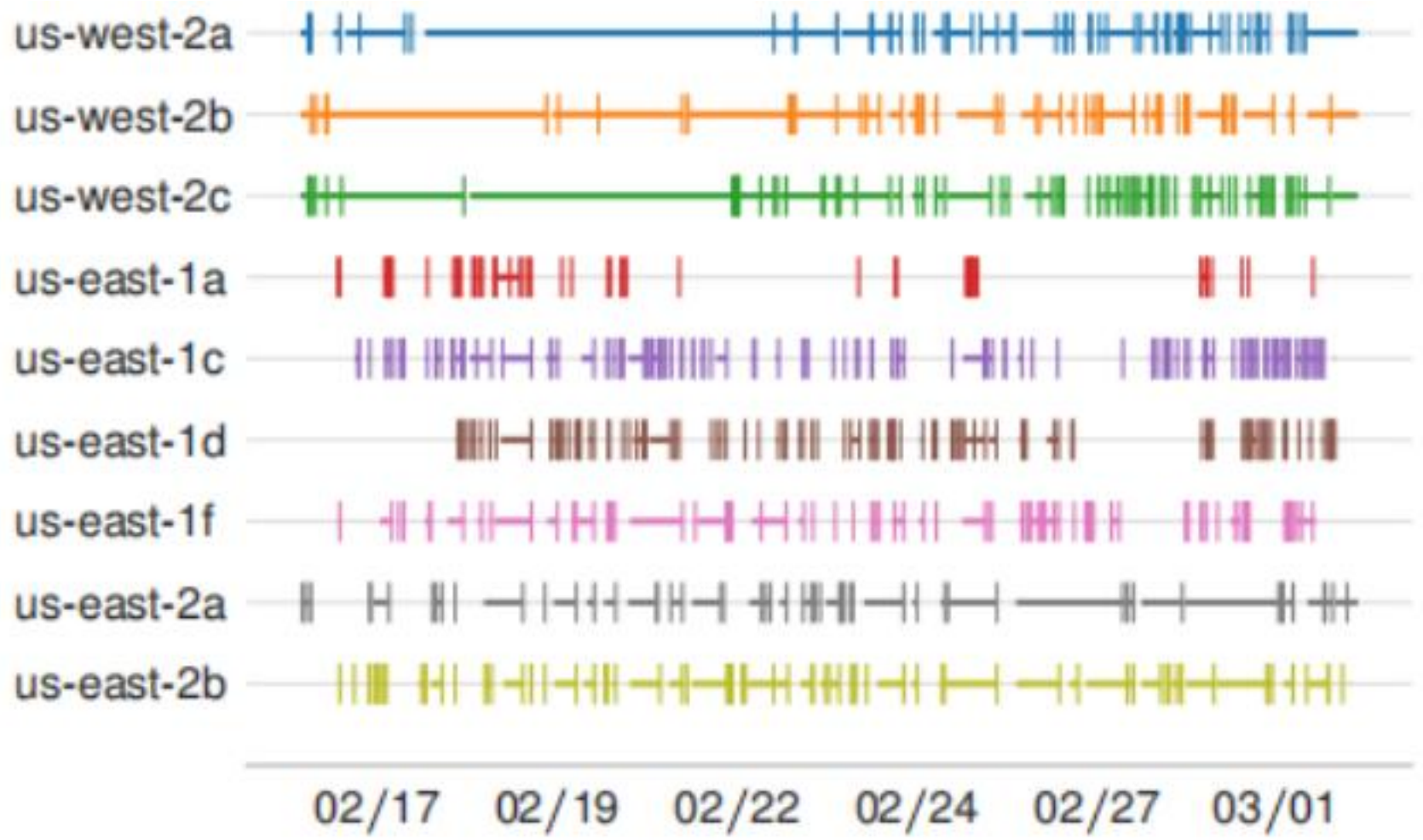
Spot Instance数据采集

每十分钟测试一次可用性，然后紧急停止



这种方法将跟踪收集的成本降低了约100×。

Spot Instance数据采集



spot Instance可用性在不同区域和时间上有显著差异。

Spot Instance数据采集

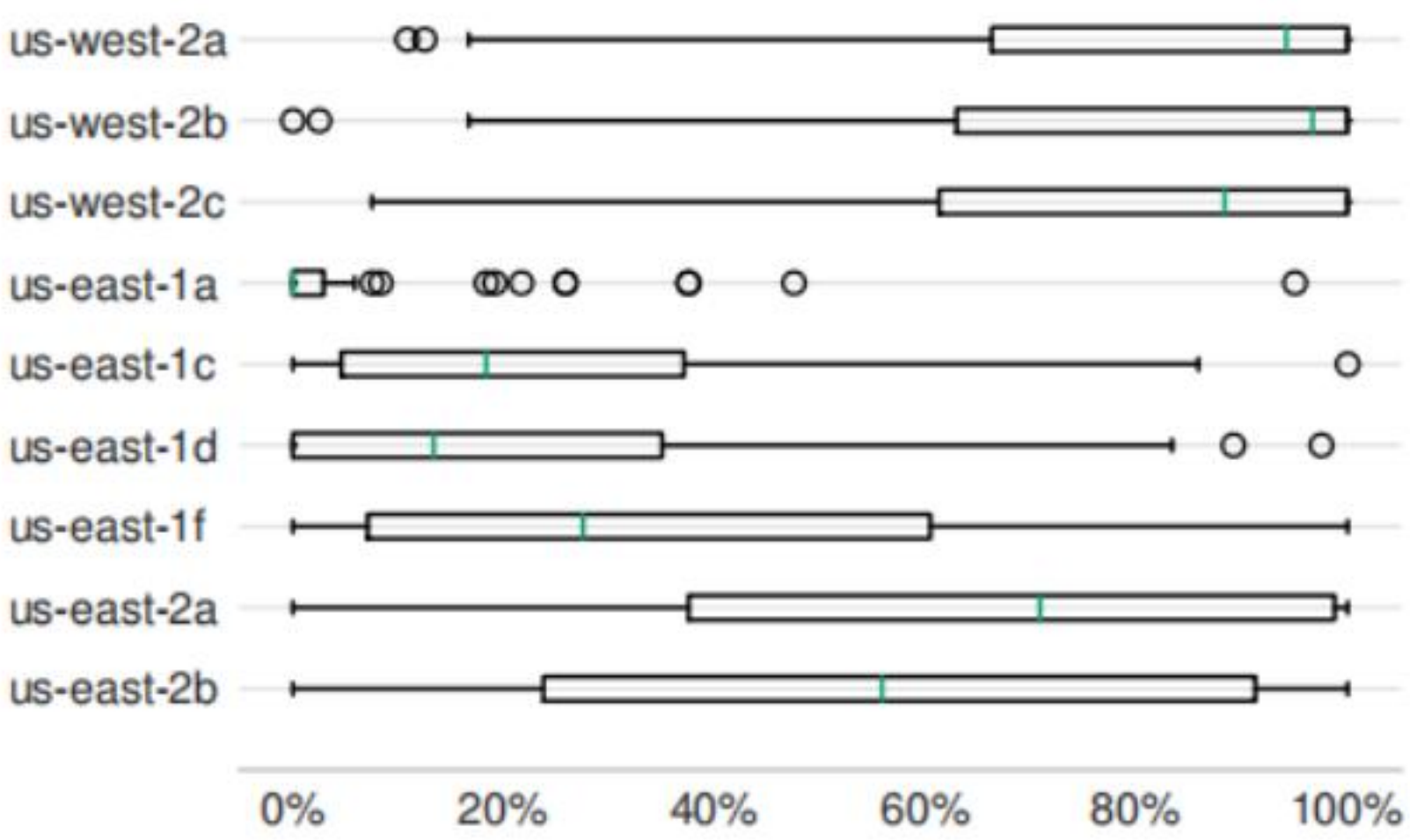
2周的时间内覆盖6小时的窗口(因此，每个区域覆盖14×24/6 = 56个窗口)

可用性：对于每个时间窗口，计算Instance可用的时间与总时间(6小时)的比例。这个比例就是可用性分数，表示为百分比。

Spot Instance数据采集

2周的时间内覆盖6小时的窗口(因此，每个区域覆盖14×24/6 = 56个窗口)

可用性：对于每个时间窗口，计算Instance可用的时间与总时间(6小时)的比例。这个比例就是可用性分数，表示为百分比。

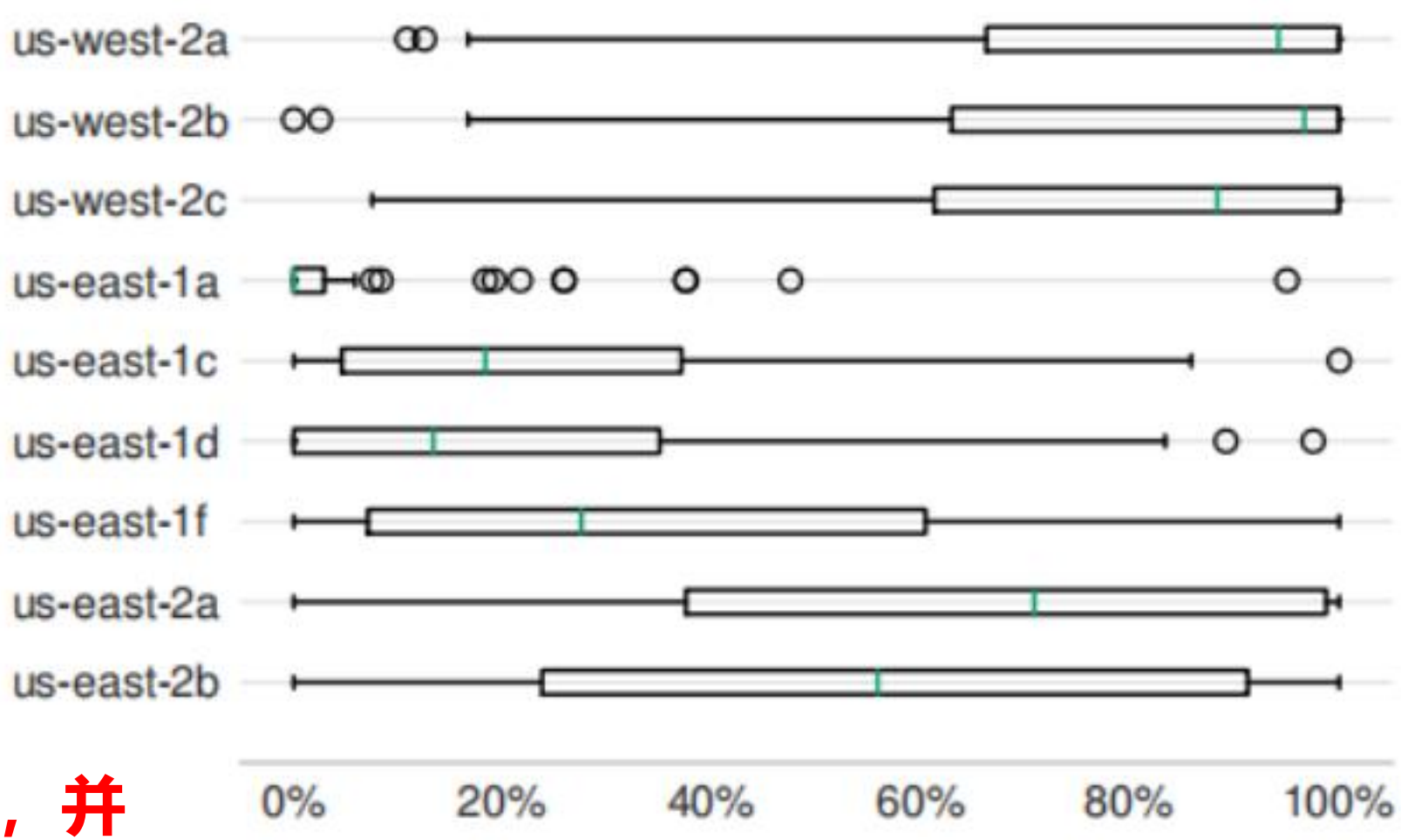


每个区域的56个窗口中的时间点**可用性**分数的分布

Spot Instance数据采集

2周的时间内覆盖6小时的窗口(因此，每个区域覆盖14×24/6 = 56个窗口)

可用性：对于每个时间窗口，计算Instance可用的时间与总时间(6小时)的比例。这个比例就是可用性分数，表示为百分比。

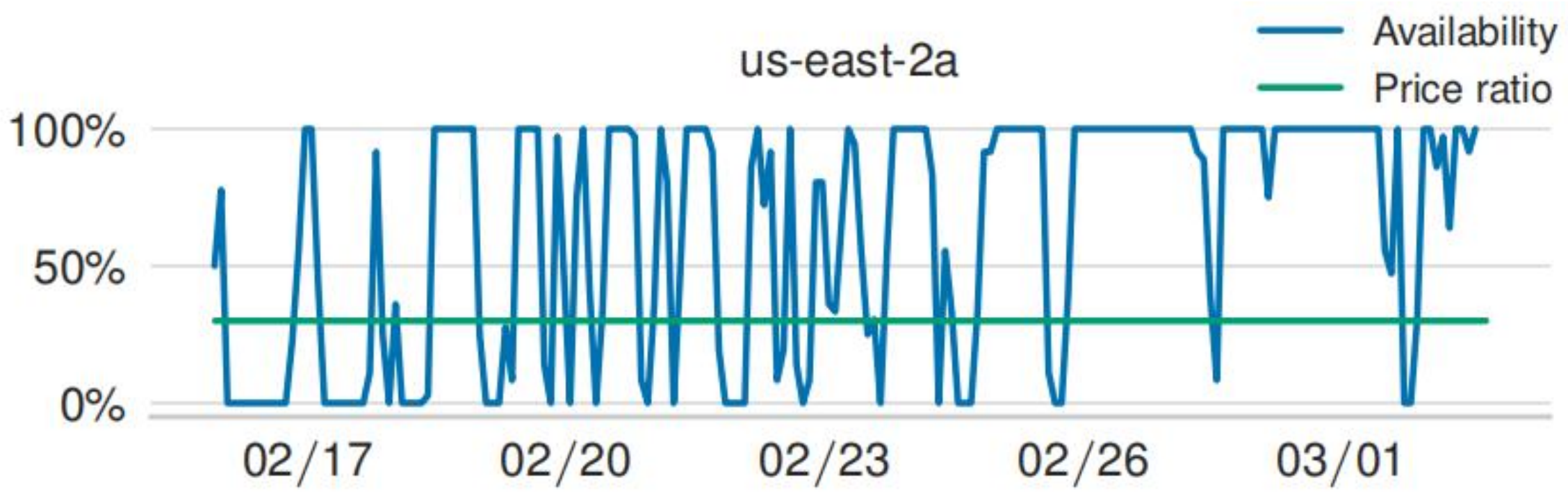


每个区域的56个窗口中的时间点**可用性**分数的分布

不同区域差异性很大，并且是及其不稳定的。

Spot Instance数据采集

Price ratio: spot价格除以on-demand按需价格。



- spot可用性分数的高波动性。
- spot可用性可以在几个小时内从100%跃升到0%。
- Price ratio相对稳定，几乎不变。

提纲

使用spot完成截止日期敏感的工作

关注的是可能出现抢占的长期(数小时到数天)的工作。

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

Table 2: State space for a job.

Idle是一种休眠模式，大家可以认为是处于待机或者等待状态。

使用spot完成截止日期敏感的工作

C(t) 剩余计算时间

R(t) 剩余截止时间

作业的总计算时间是C(0)，截止时间是R(0)。

d 转换延迟(从原来Instance 1转为其他Instance 2的延迟)

on-demand Instance的价格定义为k > 1，

spot Instance的价格定义为1。

目标：使用on-demand和spot Instance，在截止日期R (0)之前完成作业的计算时间C(0)，即C(R(0))≤0。

调度策略

节俭的规则：在C(t)=0之后，计算任务(作业)应保持空闲状态。

安全网的规则：当作业空闲和R (t) < C (t) + 2d时，切换到on-demand Instance运行，并一直保持到作业结束。

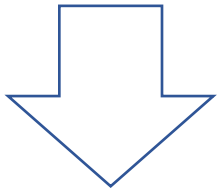
开发规则：一旦开始使用一个spot Instance，保持它，直到它被抢占。

调度策略

节俭的规则：在C(t)=0之后，计算任务(作业)应保持空闲状态。

安全网的规则：当作业空闲和R (t) < C (t) + 2d时，切换到on-demand Instance运行，并一直保持到作业结束。

开发规则：一旦开始使用一个spot Instance，保持它，直到它被抢占。

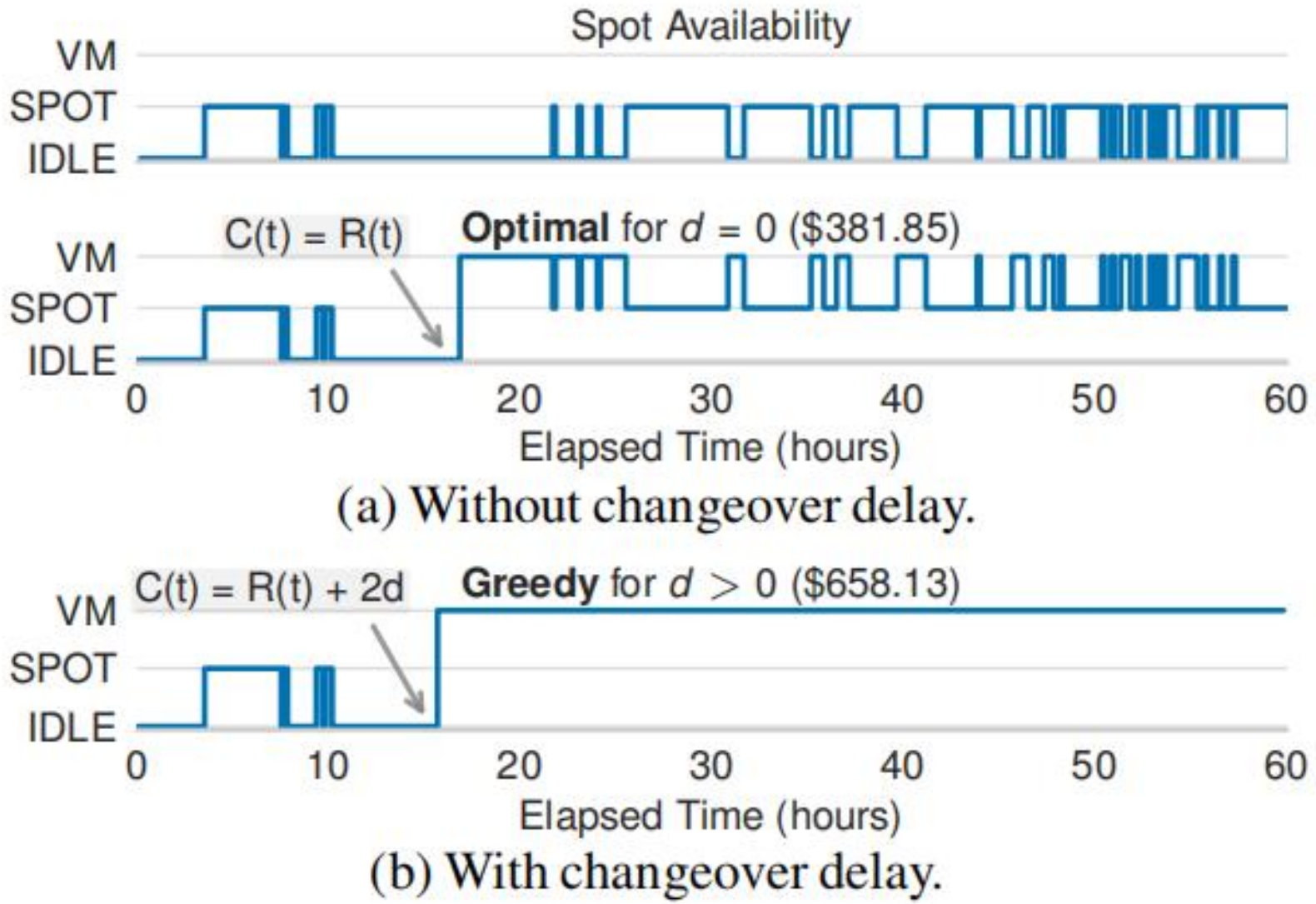


设计贪婪策略

1.保持在任何可用的spot Instance上，直到它被抢占(**开发规则**)，如果没有可用的spot Instance，则继续等待，即在表2中的②和③之间的过渡。

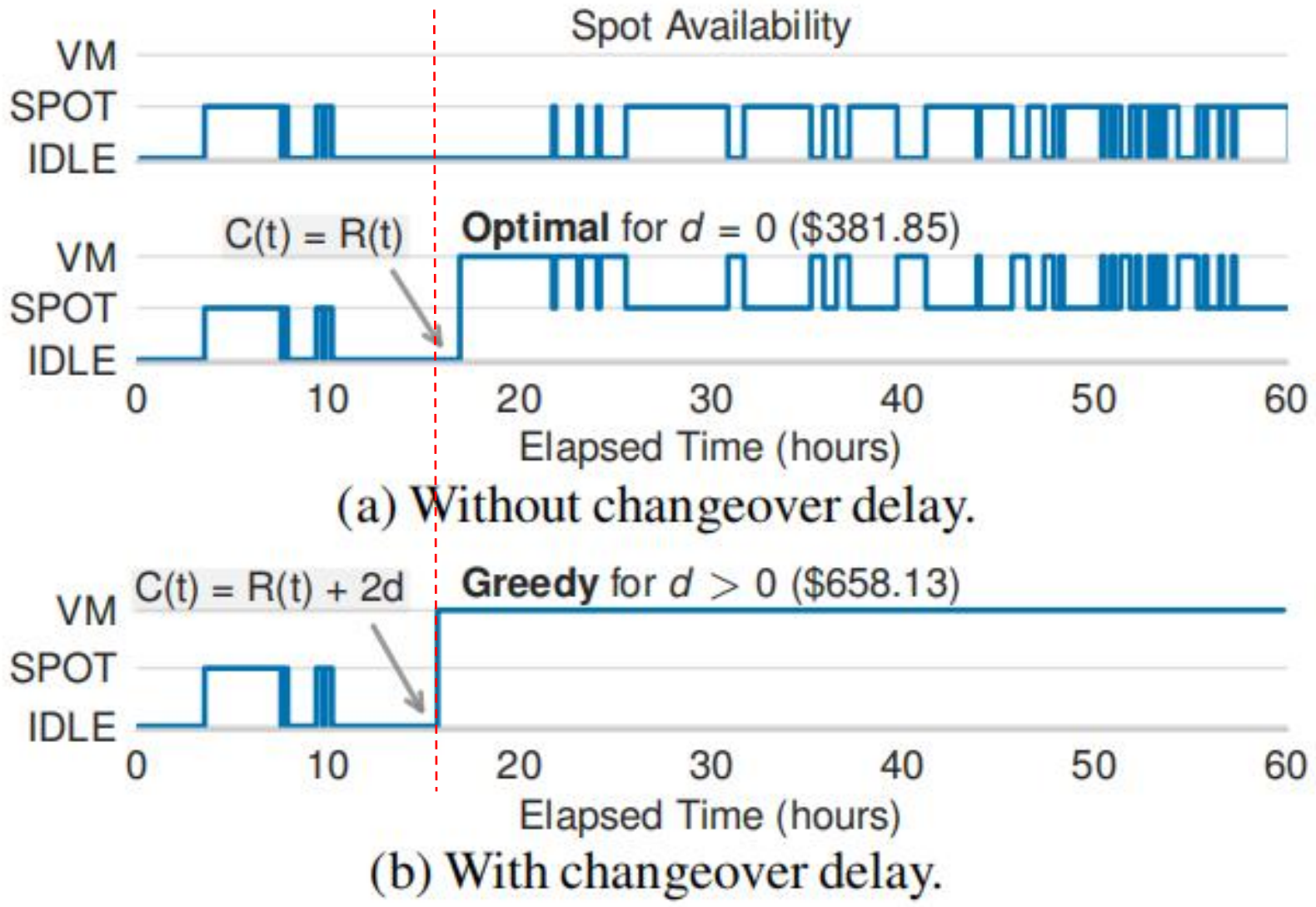
2.(**安全网规则**)当R (t)<C (t)+2d保持不变且作业空闲时，使用demand Instance直到结束。

调度策略



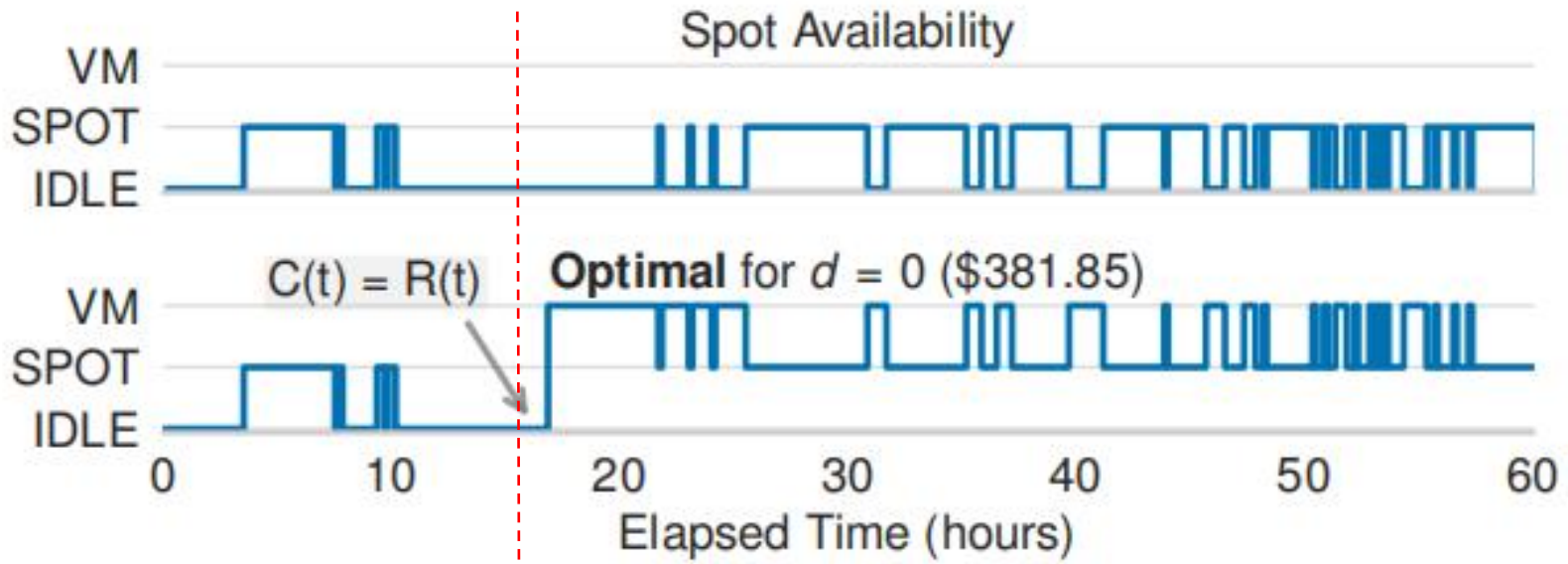
关于AWS上实际现场可用性的策略的决策跟踪示例。

调度策略



关于AWS上实际现场可用性的策略的决策跟踪示例。

调度策略



(a) Without changeover delay.

C(t) = R(t) + 2d **Greedy** for $d > 0$ ($658.13)

我们能在不假设未来知识的同时，比贪婪策略做得更好吗?

(b) With changeover delay.

关于AWS上实际现场可用性的策略的决策跟踪示例。

理论分析

证明： 在**最坏**和**平均**情况下的情况下比贪婪更好的策略的存在性。

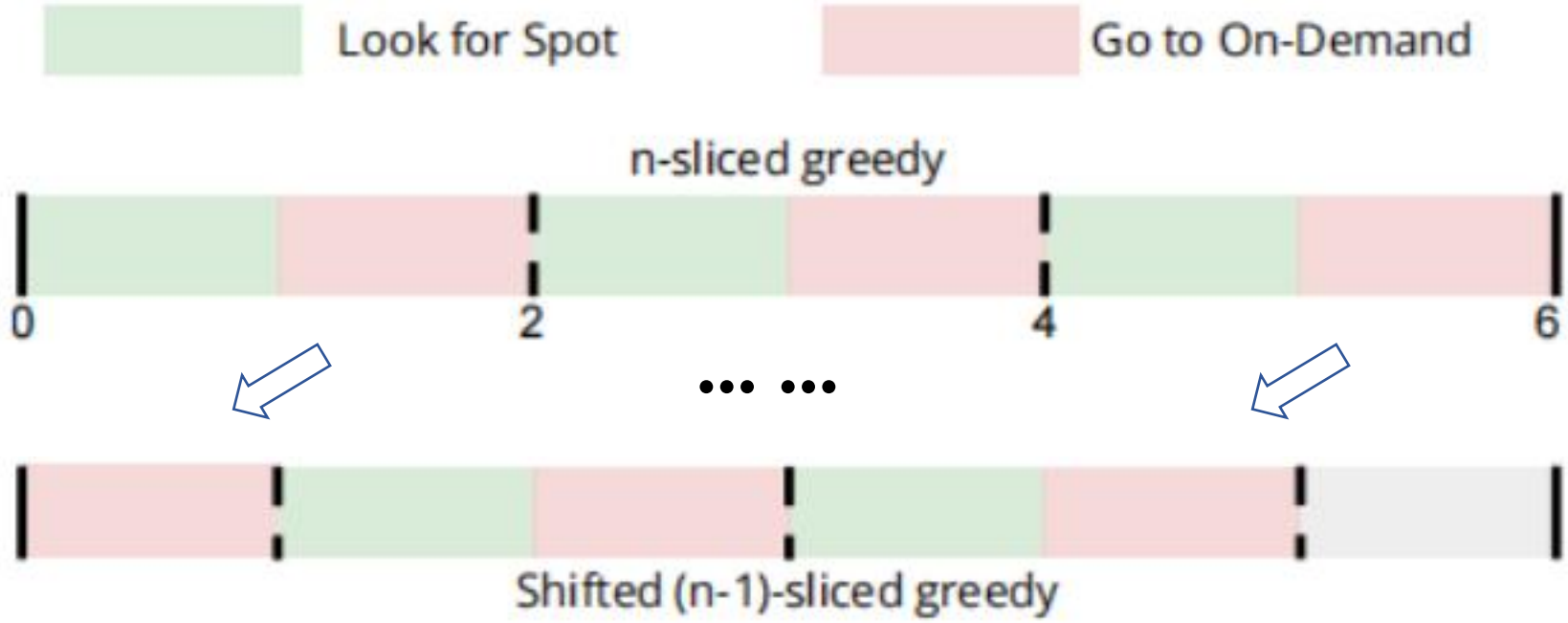
理论分析

证明： 在**最坏**和平均情况下的情况下比贪婪更好的策略的存在性。

随机移位贪婪(RSF)策略的切片示例，其中截止日期R (0) =6，计算时间C(0)=3，切片n=3。虚线表示切片的边界。

理论分析

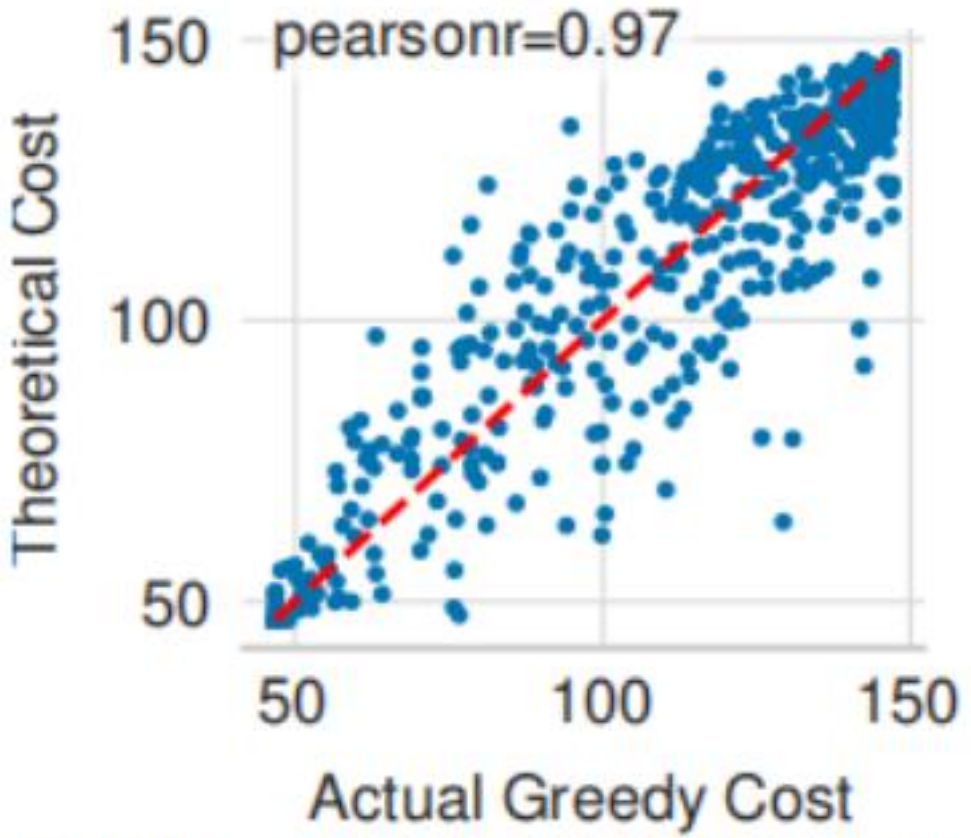
证明： 在**最坏**和平均情况下的情况下比贪婪更好的策略的存在性。



随机移位贪婪(RSF)策略的切片示例，其中截止日期R (0) =6，计算时间C(0)=3，切片n=3。虚线表示切片的边界。

理论分析

证明： 在最坏和**平均**情况下的情况下比贪婪更好的策略的存在性。



(a) Theoretical vs actual greedy cost with delay $d = 0.01h$.

$C(0) = 48$ h

$R(0) \in [52, 92]$ h

$d = 0.01$ h

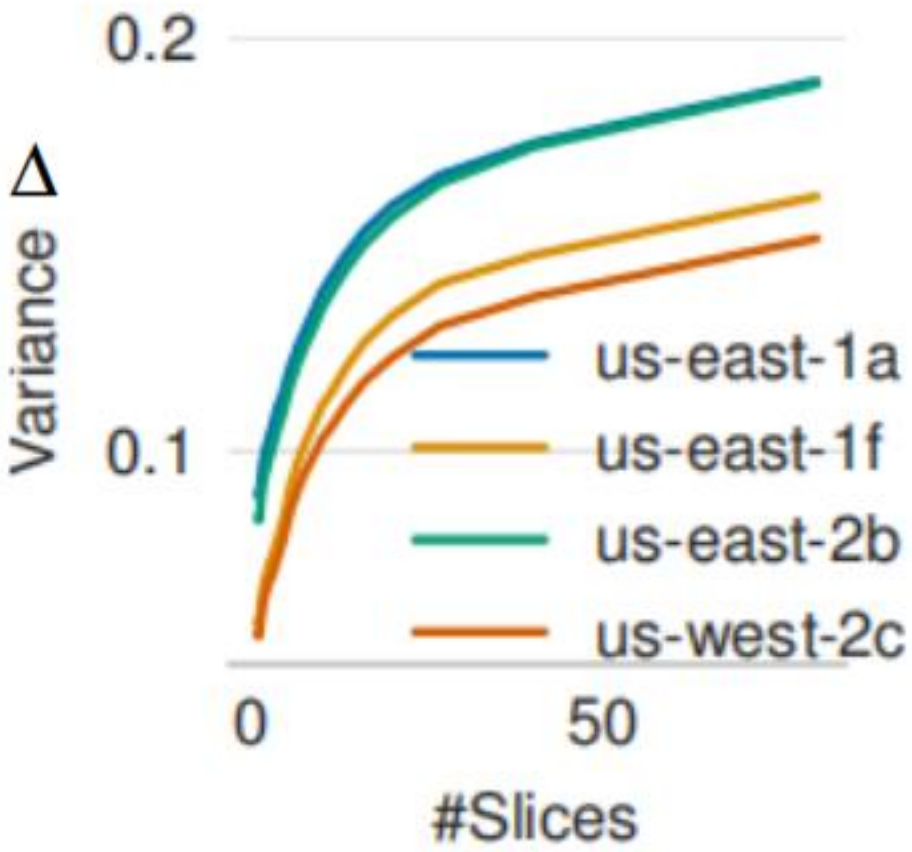
理论分析

证明：在最坏和**平均**情况下的情况下比贪婪更好的策略的存在性。

例如，**如果spot在抢占后有4小时的平均寿命和1小时的平均等待时间。**然后，spot有一个使用比率，r = 4/(4+1)= 0.8，使用它的作业每单位时间产生0.8个进度。

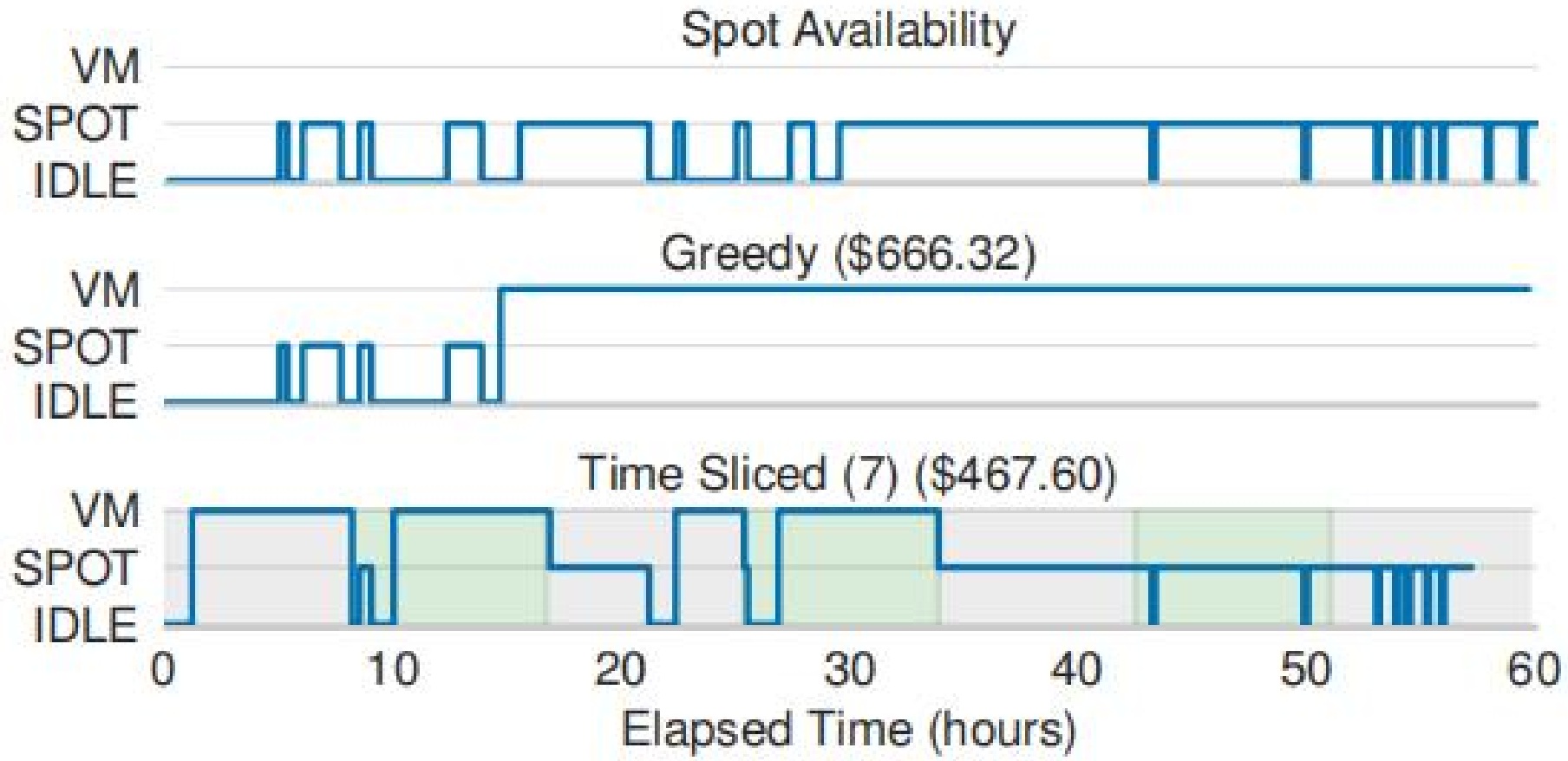
$$\Delta = \frac{R(0) - C(0)}{(1-r)^3}(\hat{v} - v)$$

其中，v^是长度为R(0)/n的切片上的方差，v是长度为R(0)的轨迹的方差。

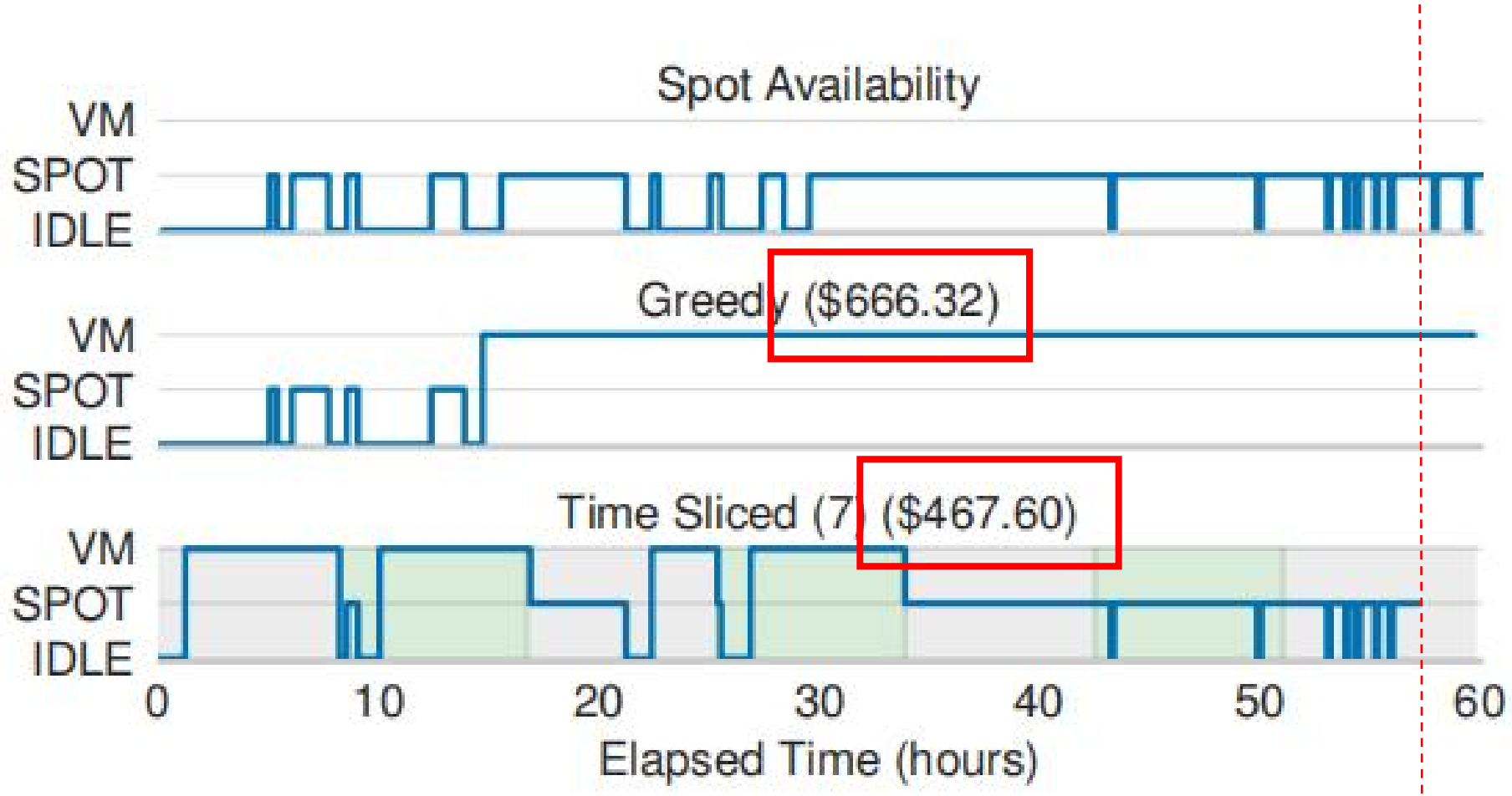


(b) Variance vs number of slices with an 80-hour deadline.

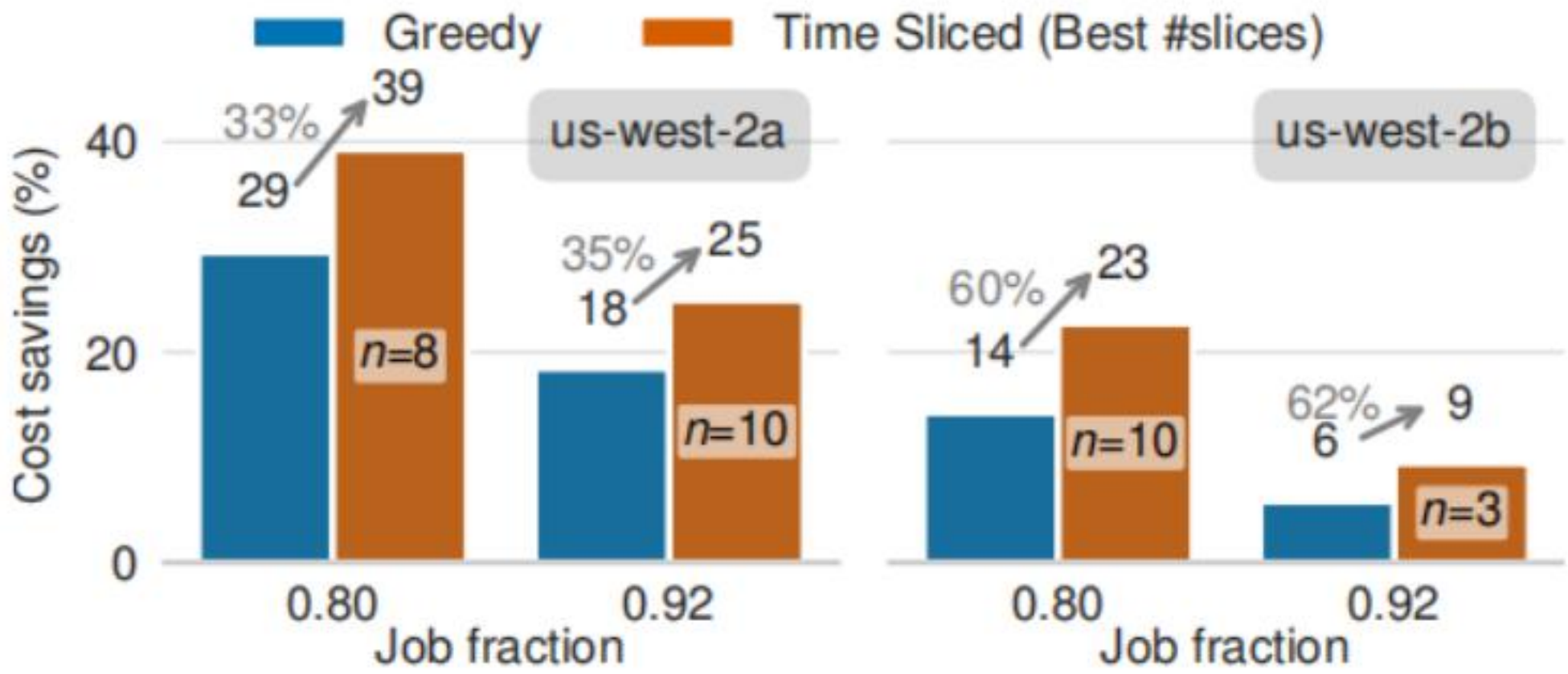
模型设计


Spot Availability
VM
SPOT
IDLE
Greedy ($666.32)
VM
SPOT
IDLE
Time Sliced (7) ($467.60)
VM
SPOT
IDLE
0
10
20
30
40
50
60
Elapsed Time (hours)

模型设计


Spot Availability
VM
SPOT
IDLE
Greedy ($666.32)
VM
SPOT
IDLE
Time Sliced (7) ($467.60)
VM
SPOT
IDLE
0
10
20
30
40
50
60
Elapsed Time (hours)

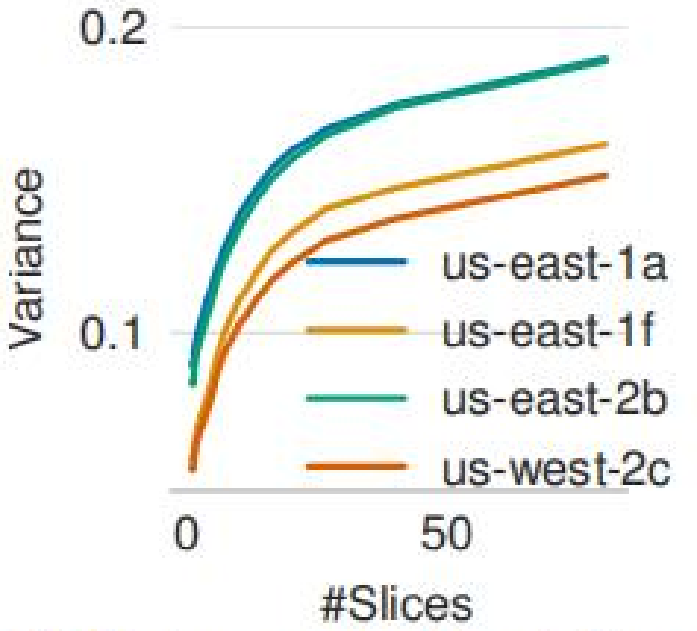
模型设计


Greedy
Time Sliced (Best #slices)
Cost savings (%)
40
20
0
us-west-2a
us-west-2b
33%
39
29
n=8
35%
25
18
n=10
60%
23
14
n=10
62%
9
6
n=3
0.80
0.92
Job fraction
0.80
0.92
Job fraction

模型设计

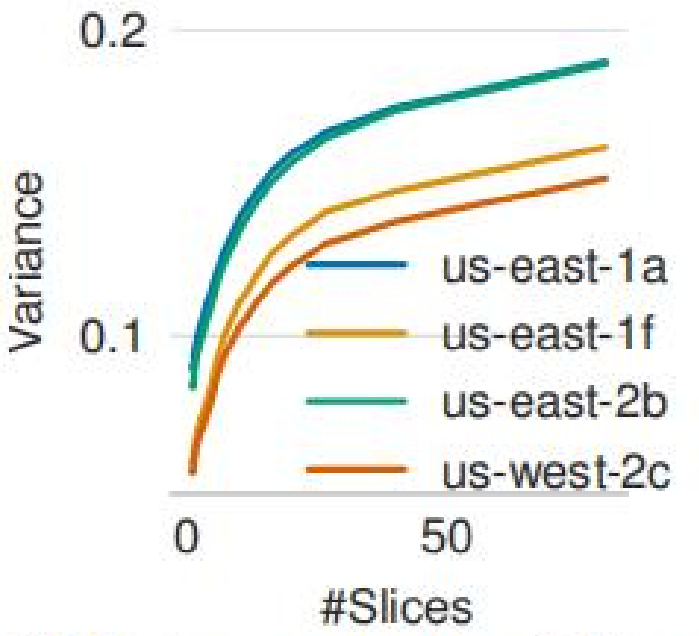
虽然最佳切片数n的时间切片策略优于贪婪策略，但在不同情况下选择最优n是不现实的。

统一进度(无参数策略)



(b) Variance vs number of slices with an 80-hour deadline.

统一进度(无参数策略)



(b) Variance vs number of slices with an 80-hour deadline.

在三个 instance状态之间切换：

idle, spot, and on-demand。

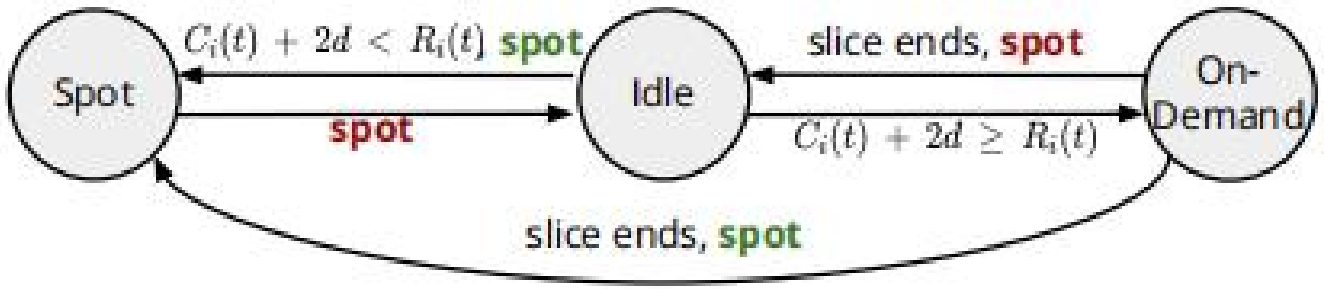
该策略基于前面的3条规则，有以下规则：

1. **统一进度**：当作业空闲且cp(t)<ep(t)时，切换到on-demand Instance并保持以赶上进度。
2. **承担风险**：随时可用(即使切换到cp(t) <ep(t))。待在spot Instance，直到它被抢占(开发规则)。
3. **滞后**：on-demand作业时，直到cp(t)≥ep(t+2d)。

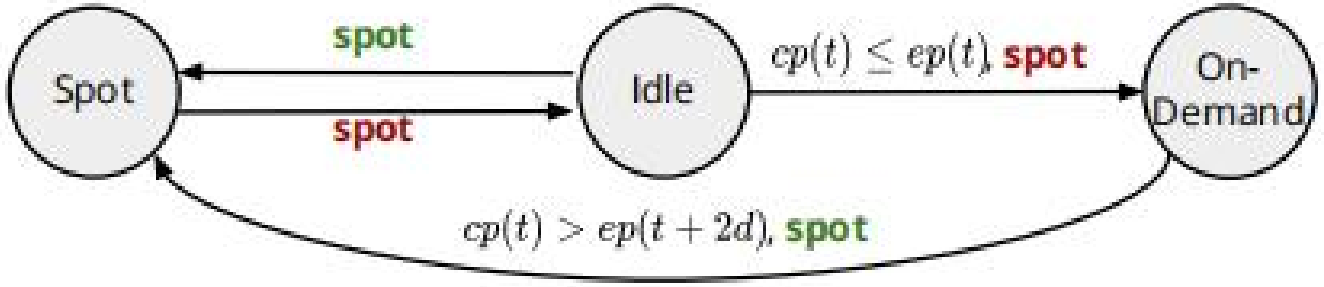
当前的进度，cp(t)= C(0)−C(t)，保证满足预期的进度，ep(t)：

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

统一进度(无参数策略)



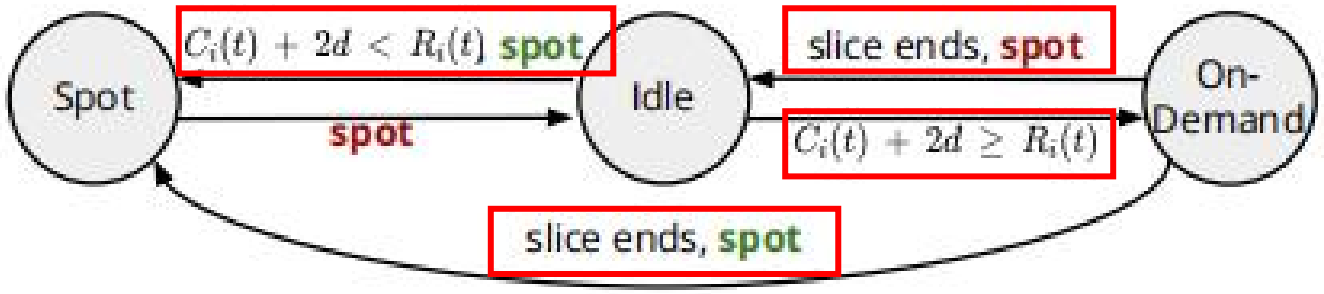
(a) Time Sliced



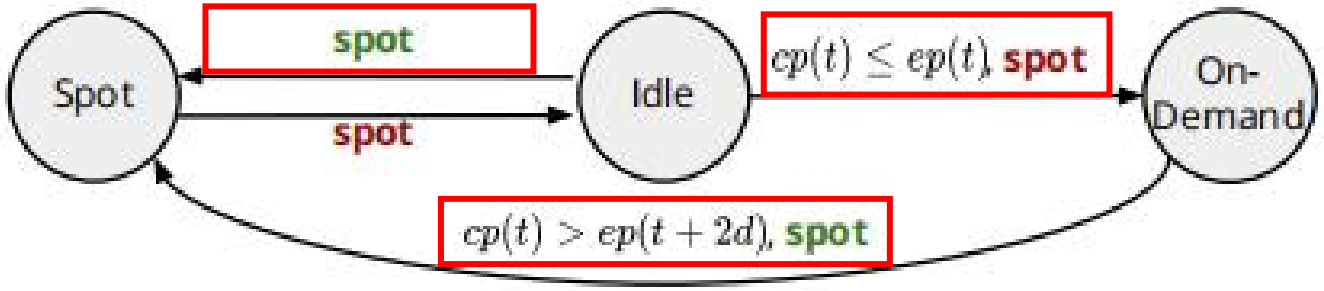
(b) Uniform Progress

Figure 10: State machine diagram for Time Sliced and Uniform Progress. **spot** means spot unavailable and **spot** means spot available. The Safety Net Rule is left out for simplicity.

统一进度(无参数策略)



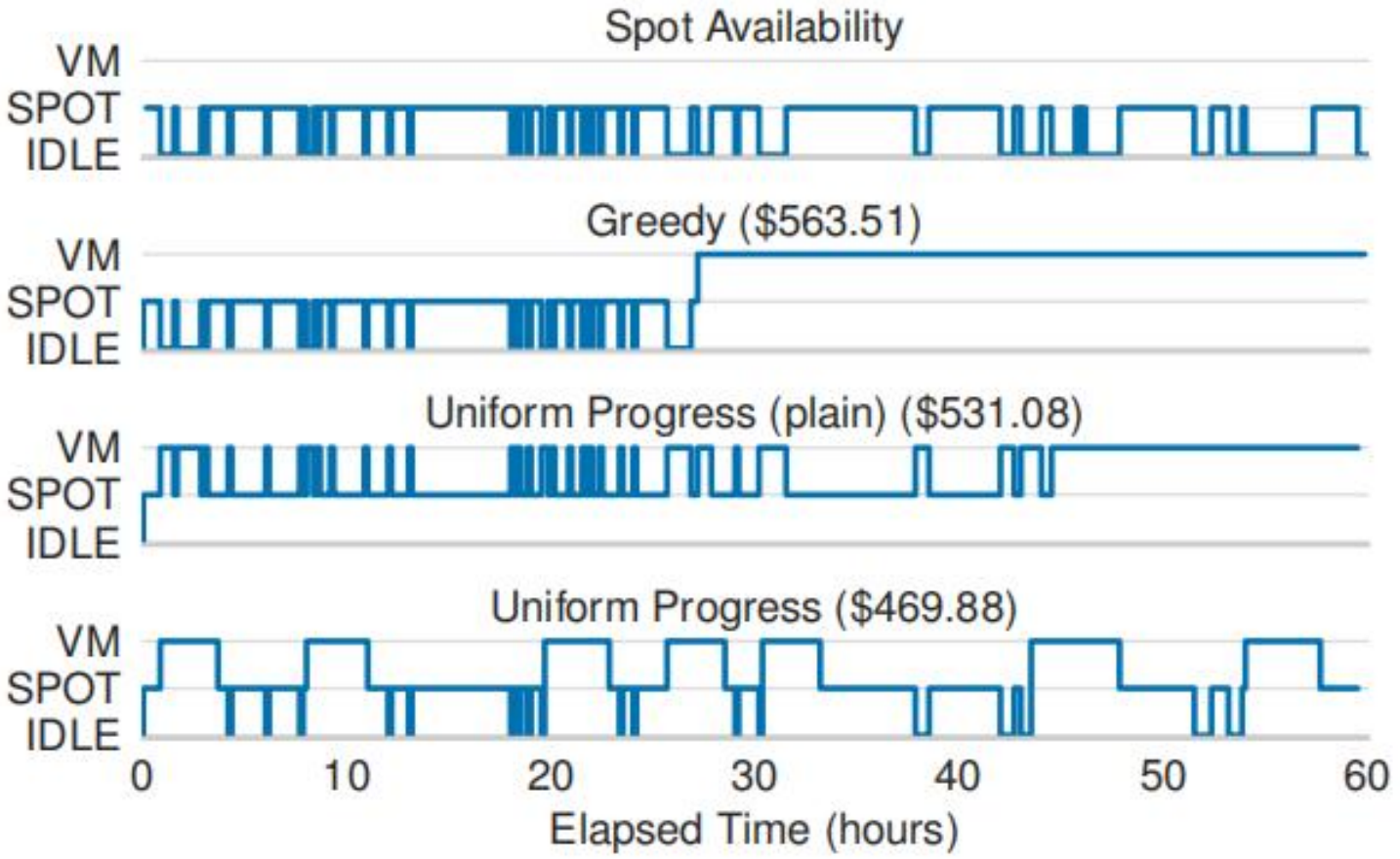
(a) Time Sliced



(b) Uniform Progress

Figure 10: State machine diagram for Time Sliced and Uniform Progress. **spot** means spot unavailable and **spot** means spot available. The Safety Net Rule is left out for simplicity.

统一进度(无参数策略)



plain是规则1和2，完整的是规则1,2,3

统一进度策略规则：

1. **统一进度**：当作业空闲且cp(t)<ep(t)时，切换到按需on-demand Instance并保持以赶上进度。
2. **承担风险**：随时可用(即使切换到cp(t) <ep(t))。待在spot，直到它被抢占(开发规则)。
3. **滞后**：on-demand作业时，直到cp(t)≥ep(t+2d)。

无所不知的策略

它假设了完整的未来知识，并产生了理论上的最优计划。

无所不知的策略

它假设了完整的未来知识，并产生了理论上的最优计划。

公式是一个整数线性规划（ILP）问题，可以用ILP求解器来求解。

minimization problem:

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

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

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

$$\forall t, x(t) \le s(t), x(t) \le 1-s(t-1), x(t) \ge s(t)-s(t-1) \tag{9}$$

$$\forall t, y(t) \le v(t), y(t) \le 1-v(t-1), y(t) \ge v(t)-v(t-1) \tag{10}$$

1. $a(t)$：这是一个函数，表示在时间 t 是否有现货实例（spot instance）可用。如果 $a(t)$ 为真（或1），则表示在时间 t 有现货实例可用；如果为假（或0），则表示没有现货实例可用。
2. $s(t)$ **和** $v(t)$：这两个函数表示策略在时间 t 选择使用哪种类型的实例。通常，$s(t)$ 可能表示选择使用现货实例，而 $v(t)$ 表示选择使用按需实例（on-demand instance）。如果 $s(t)$ 为真，那么在时间 t 策略选择了现货实例；如果 $v(t)$ 为真，则选择了按需实例。
3. $x(t)$ **和** $y(t)$：这两个函数代表在时间 t 发生的改变延迟（changeover delays）。$x(t)$ 可能表示现货实例上的改变延迟，而 $y(t)$ 表示按需实例上的改变延迟。这些延迟可能发生在实例启动、停止或从一个实例类型切换到另一个实例类型时。

多Instances/作业的情况

多个instance的问题现在相当于单个Instance，状态的一对一映射。

集群状态：单instance作业的N个 spot、N个on-demand或无instance映射到spot、on-demand或idle状态。

Spot状态：如果可用的spot instance为a (t)<N，则它相当于在单instance场景中不可用的spot，并且一个(t)=N映射到一个可用的spot。

类似于把集群看做一个整体，多个Instances视为一个Instance。

提纲

实验部分

实验设置

$$job\ fraction\ \frac{C(0)}{R(0)} > 0.6$$

1.数据收集：研究者们在AWS上收集了spot Instance的可用性跟踪数据，这些数据用于分析和评估云计算资源分配策略的性能。

2.时间范围和Instance类型：收集的数据包括从2022年10月26日开始的为期两周的可用性跟踪，涵盖了四种Instance类型：p3.2xlarge、p3.16xlarge(配备1/8个V100 GPU)、p2.2xlarge和p2.16xlarge(配备1/8个K80 GPU)。

3.可用区：数据收集涉及两个可用区：us-west-2a和us-west-2b。

4.更长时间的跟踪：此外，从2023年2月15日开始，还收集了为期两个月的p3.2xlarge Instance在九个区域(包括us-east-1、us-east-2和us-west-2)的可用性跟踪数据。

5.多Instances跟踪：对于多个Instance，研究者们收集了在三个AWS区域(us-east-1f、us-east-2a、us-west-2c)中4个p3.2xlarge Instances的为期两周的抢占(preemption)跟踪数据，以及在三个区域(us-east-2a、us-west-2b、us-west-2c)中16个p3.2xlarge Instances的为期两周的可用性跟踪数据。

6.探测间隔：所有可用性跟踪数据都是以10分钟为间隔进行探测收集的。

7.相关性：研究表明，可用性跟踪和抢占跟踪之间存在高度相关性，这意味着在可用性跟踪上评估策略的性能应该能够反映它们在现实世界中的表现。

8.基准测试和系统评估：(1) Greedy, (2) Time Sliced (Best #slices) (3) Uniform Progress (Ours), (4)Uniform Progress (w. next spot oracle), (5)Omniscient, (6) Omniscient (8 slices)

实验评估与分析

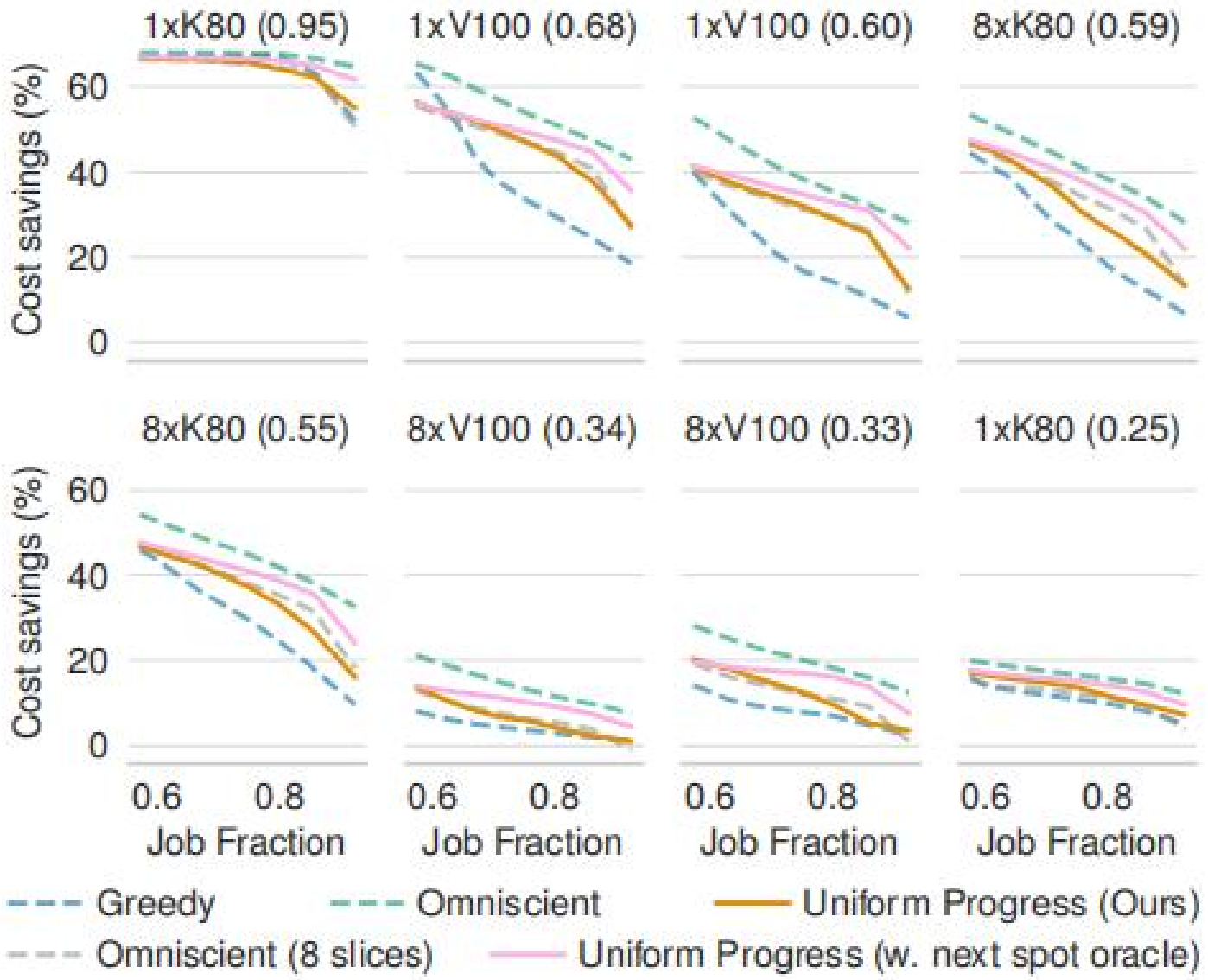
Policy	On-Demand (hours)	Spot (hours)	Spot Util.
On-Demand	48.0 ± 0.0	0.0 ± 0.0	0%
Greedy	30.8 ± 17.7	17.2 ± 17.7	63%
Uniform Progress	25.1 ± 15.3	22.9 ± 15.4	84%
Omniscient	20.7 ± 15.5	27.4 ± 15.5	100%

计算时间C(0)设置为48小时

$\frac{C(0)}{R(0)} = 0.8$ on the 2-week traces

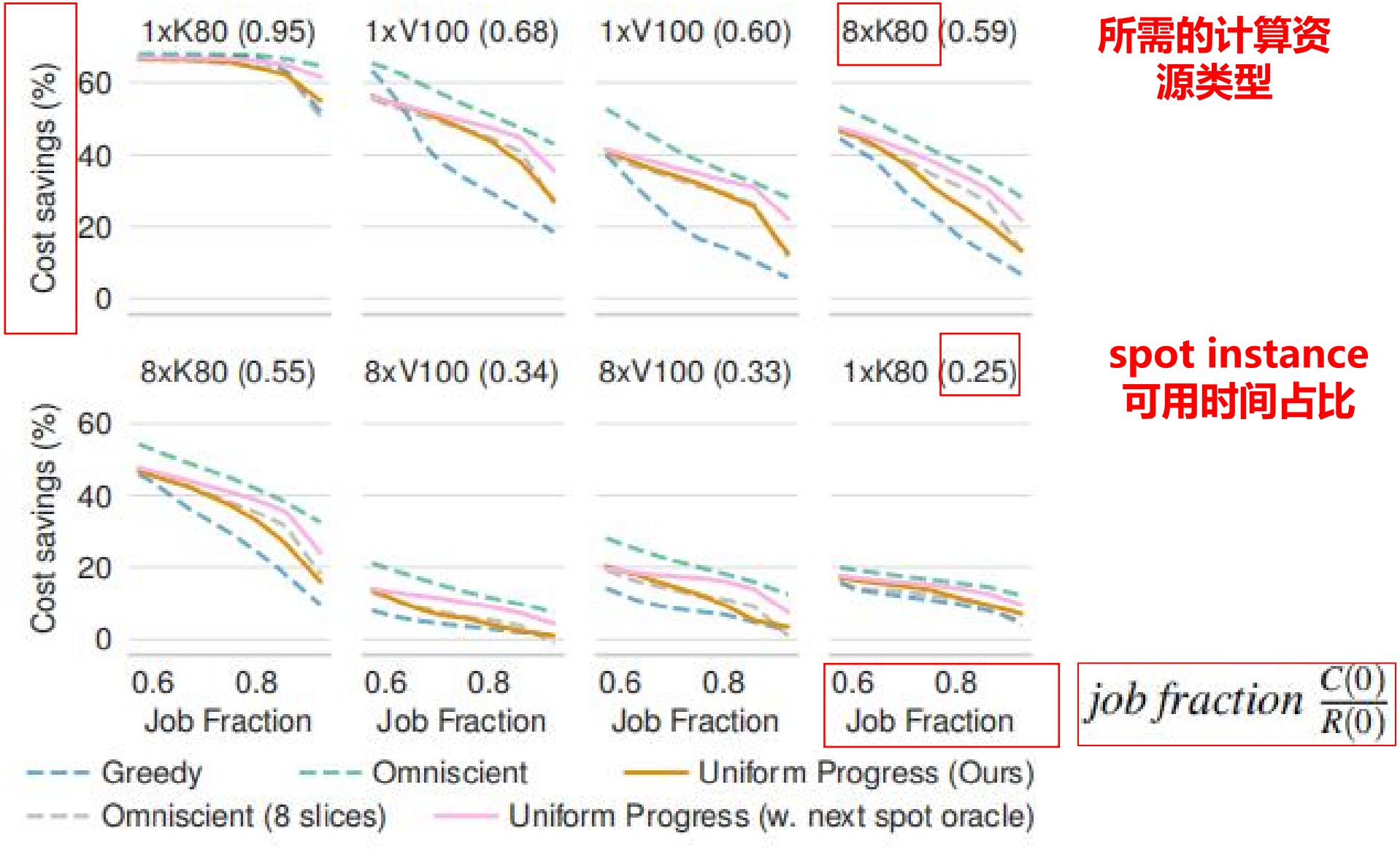
转换延迟d设置为0.2小时，成本在所有实验中按按需成本标准化。

实验评估与分析


1xK80 (0.95)
1xV100 (0.68)
1xV100 (0.60)
8xK80 (0.59)
8xK80 (0.55)
8xV100 (0.34)
8xV100 (0.33)
1xK80 (0.25)
Cost savings (%)
60
40
20
0
0.6
0.8
Job Fraction
Greedy
Omniscient
Uniform Progress (Ours)
Omniscient (8 slices)
Uniform Progress (w. next spot oracle)

实验评估与分析

节省的成本比
与全部使用on-demand相比，越多越好



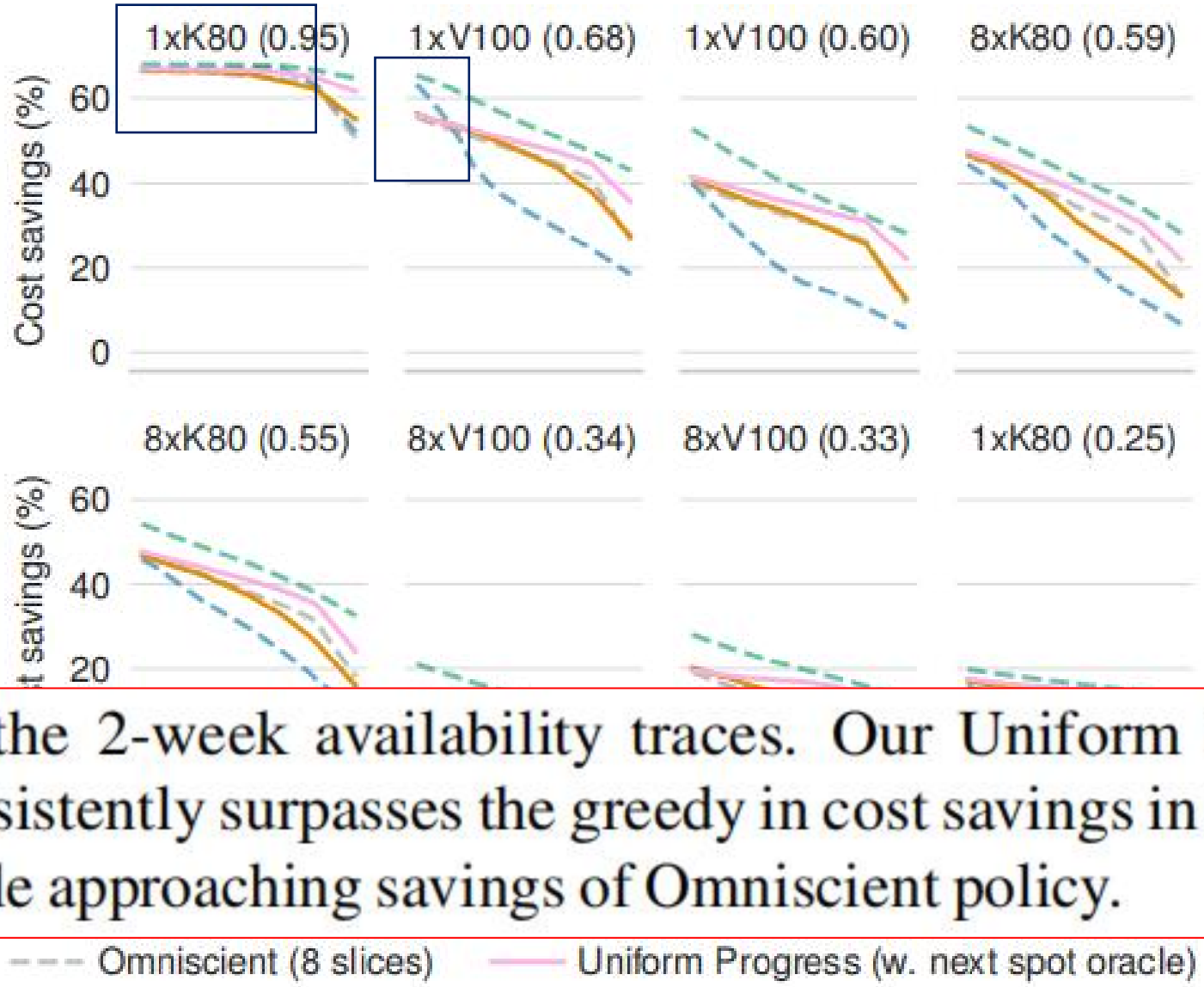
所需的计算资源类型

spot instance 可用时间占比

$$job\ fraction\ \frac{C(0)}{R(0)}$$

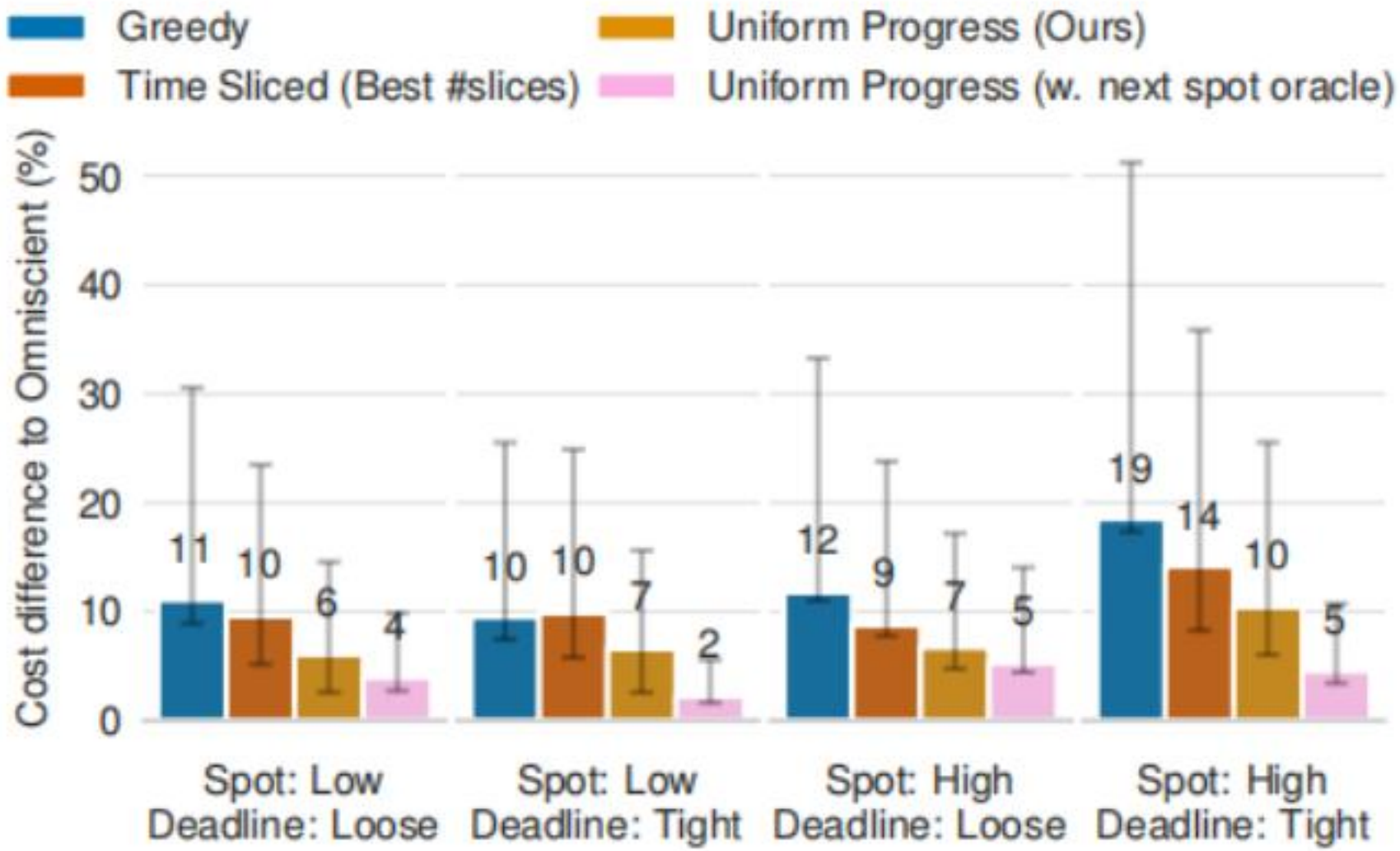
实验评估与分析

spot instance 可用时间占比相对较高，并其任务的时间延迟需求相对松弛状态下，不一定优于贪婪算法。



on the 2-week availability traces. Our Uniform Progress consistently surpasses the greedy in cost savings in all cases, while approaching savings of Omniscient policy.

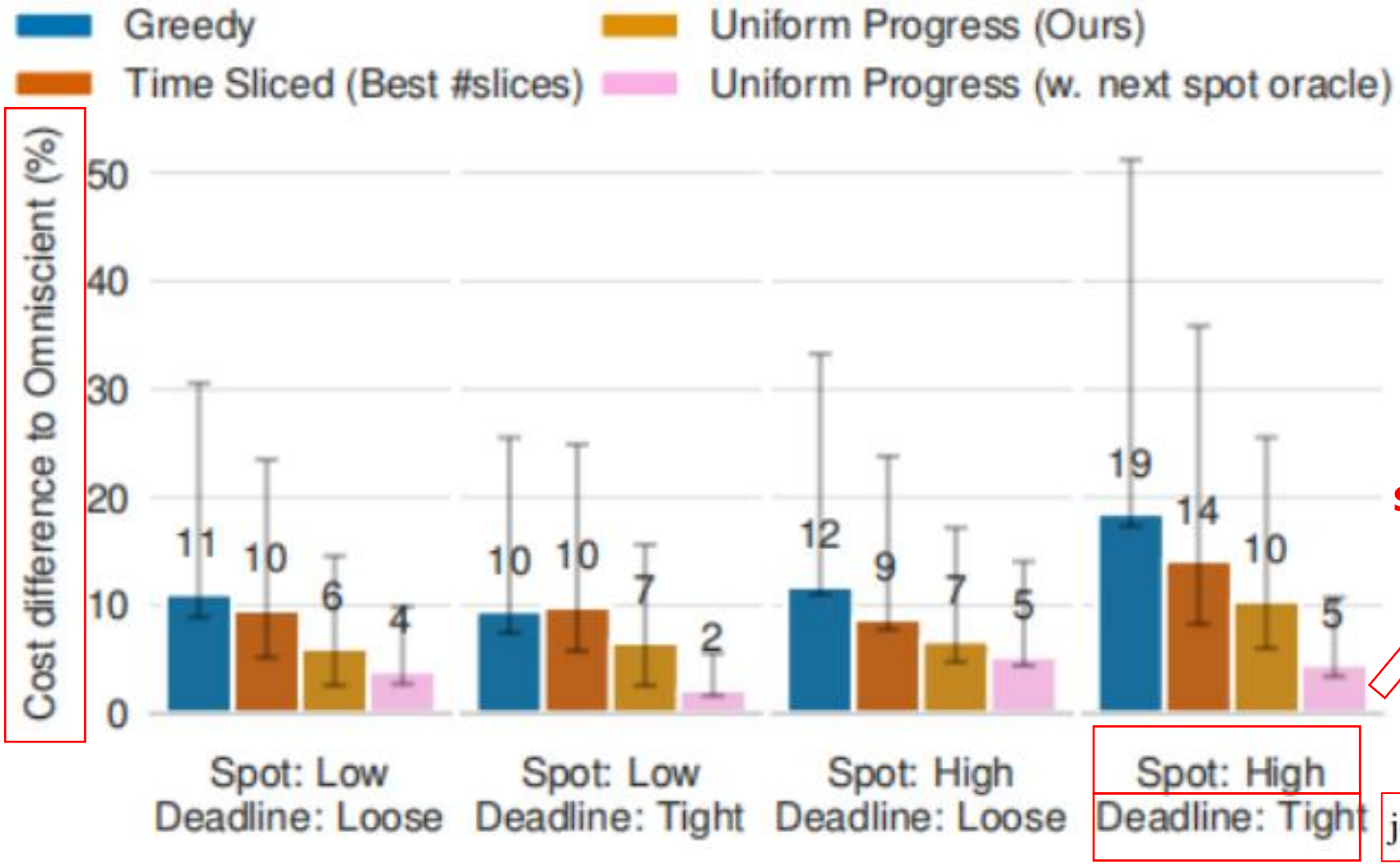
实验评估与分析


Greedy
Uniform Progress (Ours)
Time Sliced (Best #slices)
Uniform Progress (w. next spot oracle)
Cost difference to Omniscient (%)
0
10
20
30
40
50
11
10
6
4
10
10
7
2
12
9
7
5
19
14
10
5
Spot: Low
Deadline: Loose
Spot: Low
Deadline: Tight
Spot: High
Deadline: Loose
Spot: High
Deadline: Tight

实验评估与分析

与无所不知策略相比的成本差异
(成本标准化，越低更好)

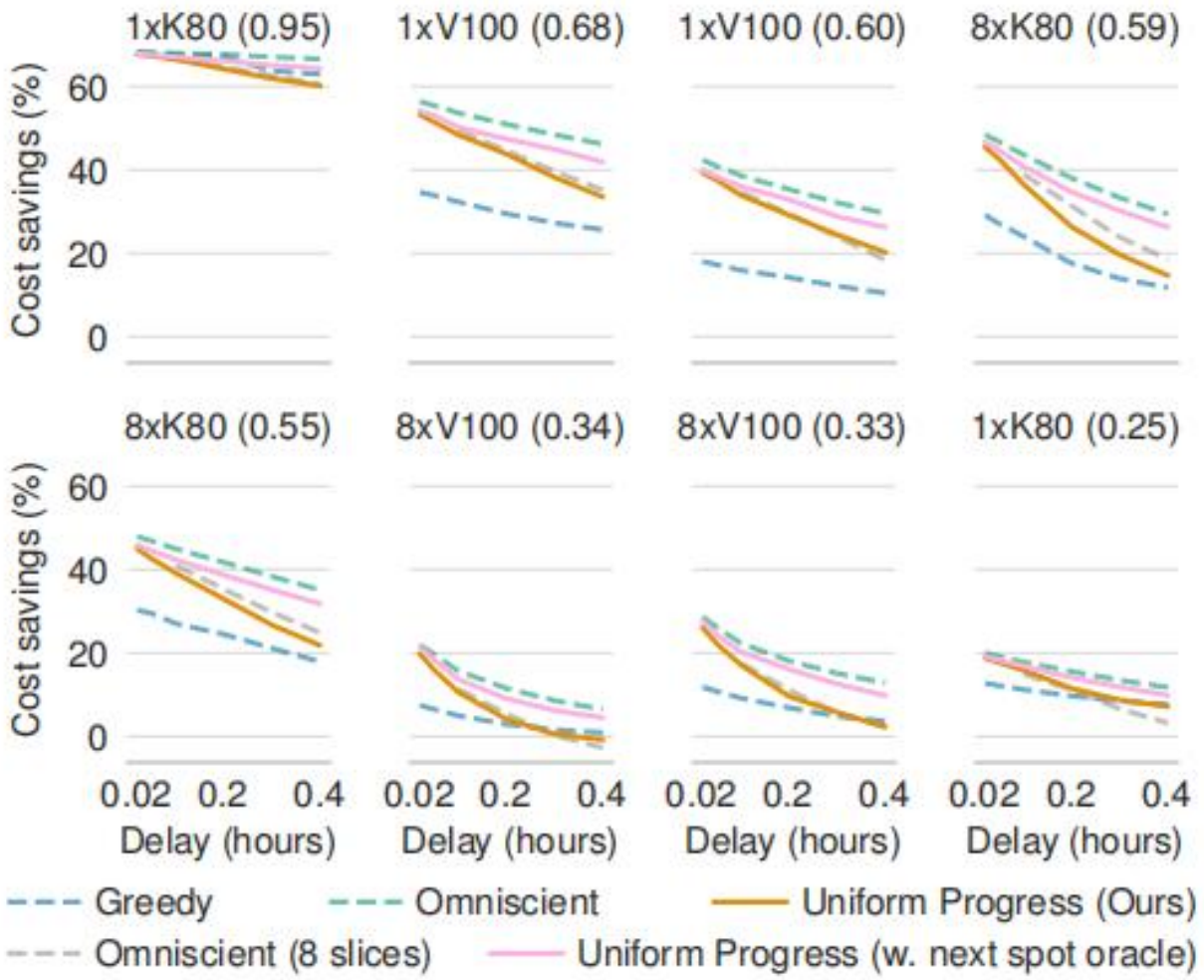
无所不知策略的结果可认为是理论上最优的结果



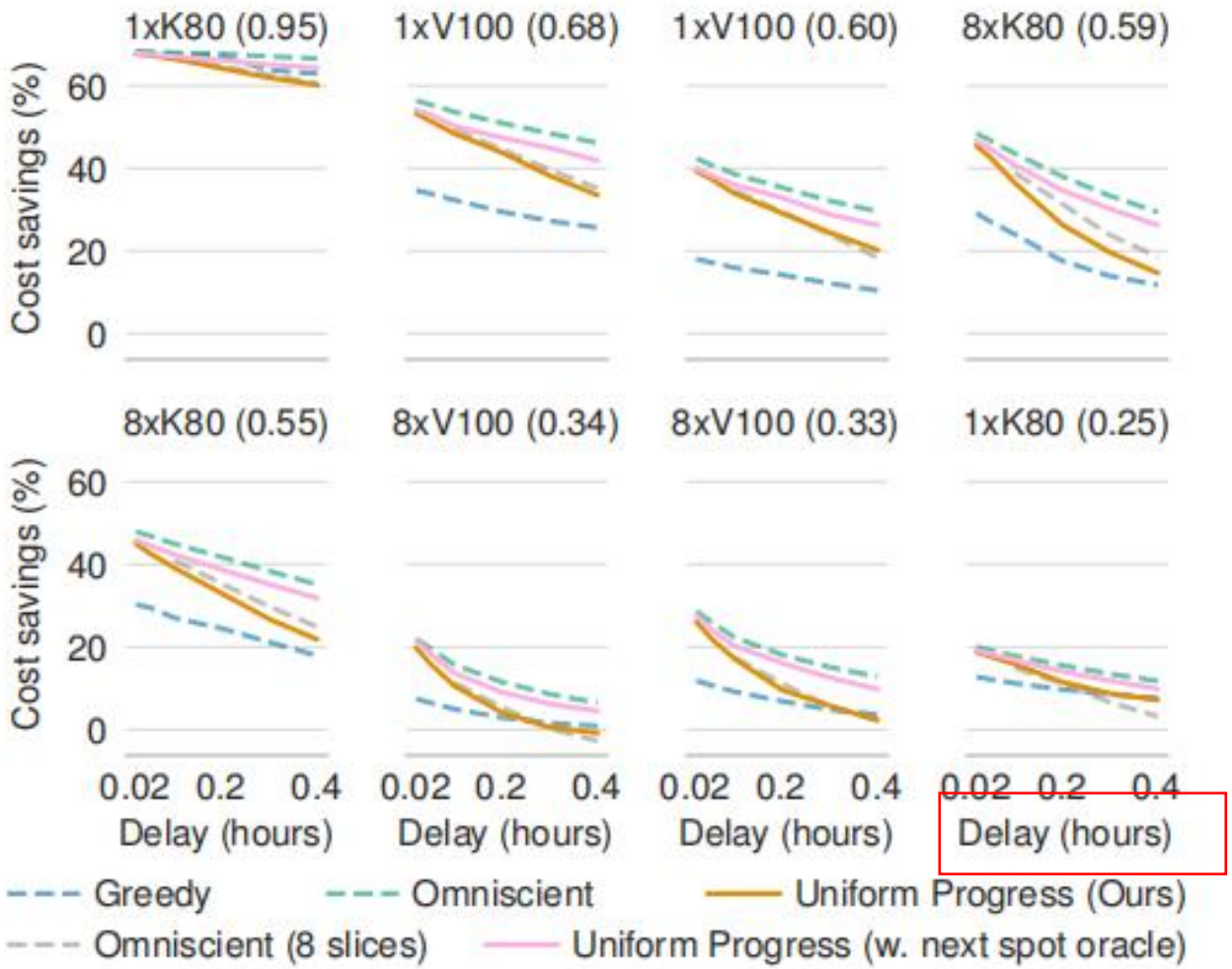
spot instance 可用时间占比 >50%

job fraction $\frac{C(0)}{R(0)} > 75\%$

实验评估与分析


1xK80 (0.95)
1xV100 (0.68)
1xV100 (0.60)
8xK80 (0.59)
8xK80 (0.55)
8xV100 (0.34)
8xV100 (0.33)
1xK80 (0.25)
Cost savings (%)
60
40
20
0
0.02
0.2
0.4
Delay (hours)
Greedy
Omniscient
Uniform Progress (Ours)
Omniscient (8 slices)
Uniform Progress (w. next spot oracle)

实验评估与分析



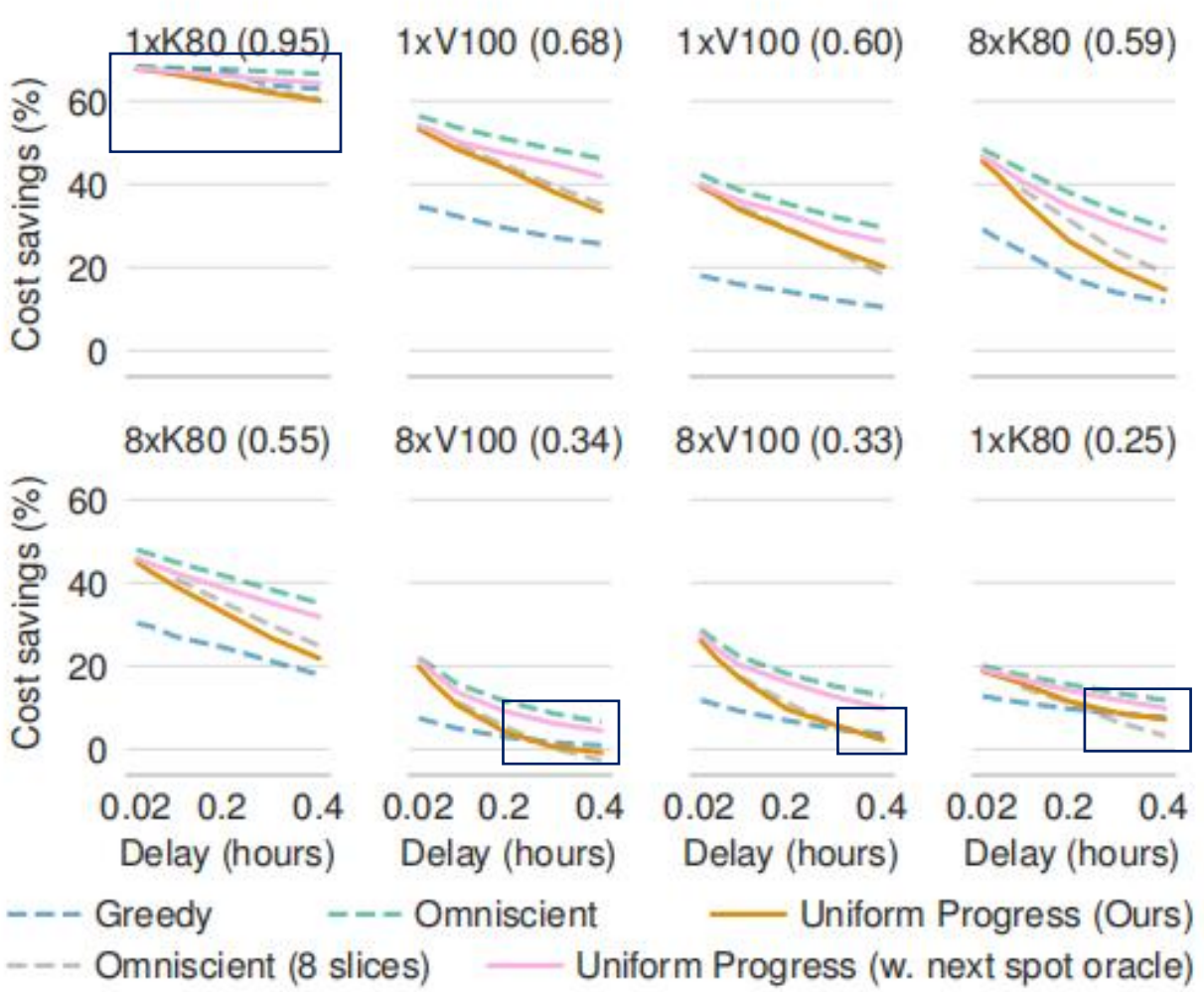
d 转换延迟

实验评估与分析

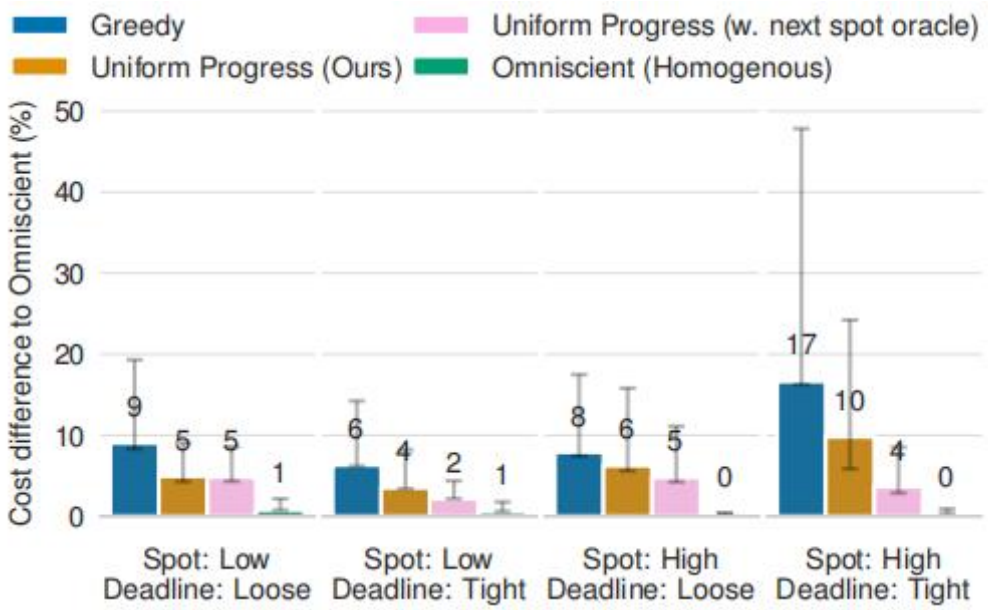
（1）spot instance可用时间占比相对较高，

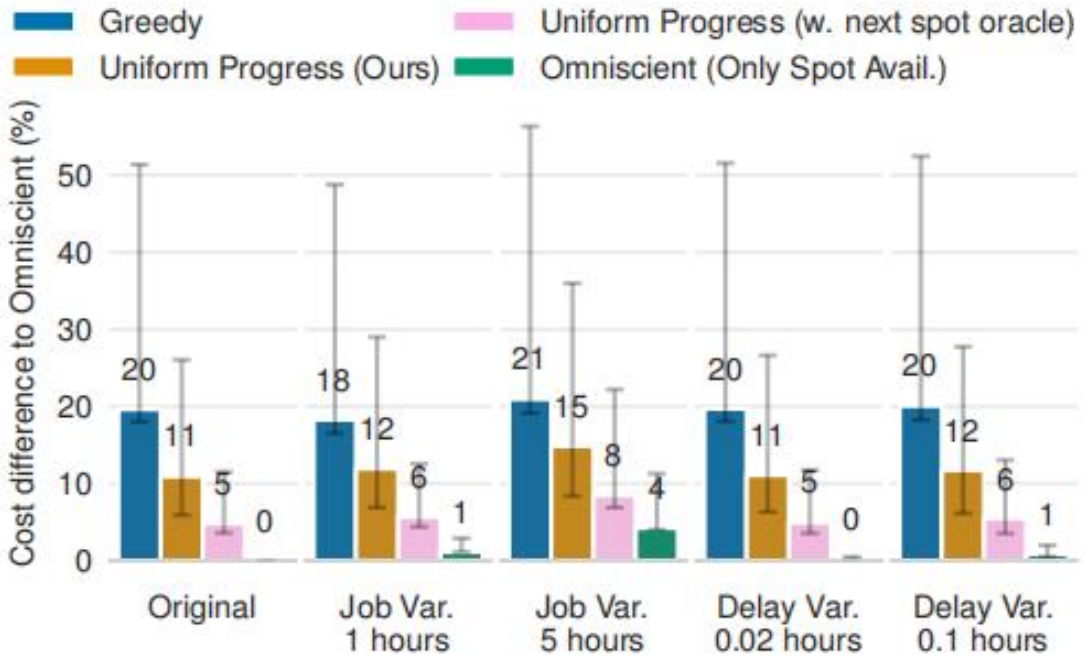
（2）或者spot instance可用时间占比相对较低并且转换延迟相对较高，

不一定优于贪婪算法。



实验评估与分析-多instances集群(4个instance)





实验部分

实际使用（实现）

$$job\ fraction\ \frac{C(0)}{R(0)}$$

在**AWS和GCP平台**上验证了策略，使用真实世界的抢占跟踪，spot可用性从70%到90%不等。
诸如转换延迟和其他系统滞后等指标直接从实现过程中进行测量，并包含在评估中。
任务包含三种工作负载(机器学习(ML)训练、生物信息学、数据分析)，设置每个工作（计算任务）的两个不同的截止日期(工作分数分别为90%和75%)。

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.

1. **Workload**：工作负载的类型，即任务的类别。
2. **Location**：工作负载运行的地理位置或云服务区域。
3. **Instance Type**：用于执行工作负载的云服务Instance类型。
4. **Spot Price (Discount)**：使用spot Instance的价格以及相对于on-demand Instance价格的折扣百分比。
5. **Computation**：完成工作负载所需的计算时间。
6. **Deadlines**：工作负载的截止时间，这些时间是根据工作负载的90%和75%完成度来确定的。
7. **Changeover Delay**：Instance切换的延迟时间，包括虚拟机(VM)配置、环境设置和作业恢复进度丢失时间的总和。

实验部分

实际使用（实现）

Workload	On-demand	Uniform Progress	
		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).

实验部分

实际使用（实现）

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

总结与思考

总结

- 通过**利用spot和on-demand instances的混合**来最小化**延迟敏感**作业的成本。
- 对spot instance的全面分析，并提出了一个理论框架来评估最坏情况和平均情况下的策略（**证明存在比使用贪婪算法结果更好的结果**）。
- **提出一个简单、无参数、有效，而不依赖于现场可用性的假设和策略（统一进度策略）**。
- 我们使用3个月的真实世界轨迹的实证研究表明，与贪婪策略相比，在节省成本方面有显著改善，在单个或多个instances上，最优策略的差距约为2×。
- 实现了一个原型，并展示了统一进度在三个实际工作负载上的有效性，将成本降低了27%-84%。
- **收集并开放了spot instance数据集。**

思考

1. 这个paper有什么问题，基于这个paper还能做什么?

- 从前面的实验中，我们看到本文所提算法并不是总优于贪心算法，那我们能不能设计个策略同时包含这两种方案，当遇到不同环境状态，自适应选择那个策略或者哪种方案。
- 多instances集群方面，我感觉是可以改进的，理论上不能将集群看做一个整体。

2. 这个paper提到的idea，能不能用在自己的方向/project上面?

- 都是混合部署：spot（不稳定，低成本）与on-demand（健壮，高成本）之间的选择问题，迁移到webassembly（功能不完备，低计算时延）和docker（功能完备，高计算时延）之间的选择问题。转换延迟---->冷启动延迟

3. 这个paper能不能泛化，需要较为熟悉这个小方向?

- 我认为这篇论文可能可以泛化到传统系统资源调度问题上，但是具体细节暂时没有想法。

感谢各位老师和同学！
请大家提出宝贵意见！