Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices

Zibo Wang，Pinghe Li，Chieh-Jan Mike Liang，Feng Wu,Francis Y. Yan

Presenter：Chuanhua Fan

2025.05.09

Dictionary

- Background
- Design
- Evaluation
- Related work
- Conclusion
- Think

Dictionary

- **Background**
- Design
- Evaluation
- Related work
- Conclusion
- Think

Background

Achieving resource efficiency while preserving end-user experience is important for cloud application operators.

To ensure a seamless end-user experience, many user-facing latency-sensitive applications impose an SLO.

Background

之前的做法：

cloud application operators resort to resource over-provisioning to avoid SLO violations.

现在的做法：

Recycling excess resources to save a significant amount of resources.

A key enabler for such resource saving is **SLO-targeted resource management.** Its **goal** is to continuously minimize the total resources allocated, while still satisfying the end-to-end latency SLO.

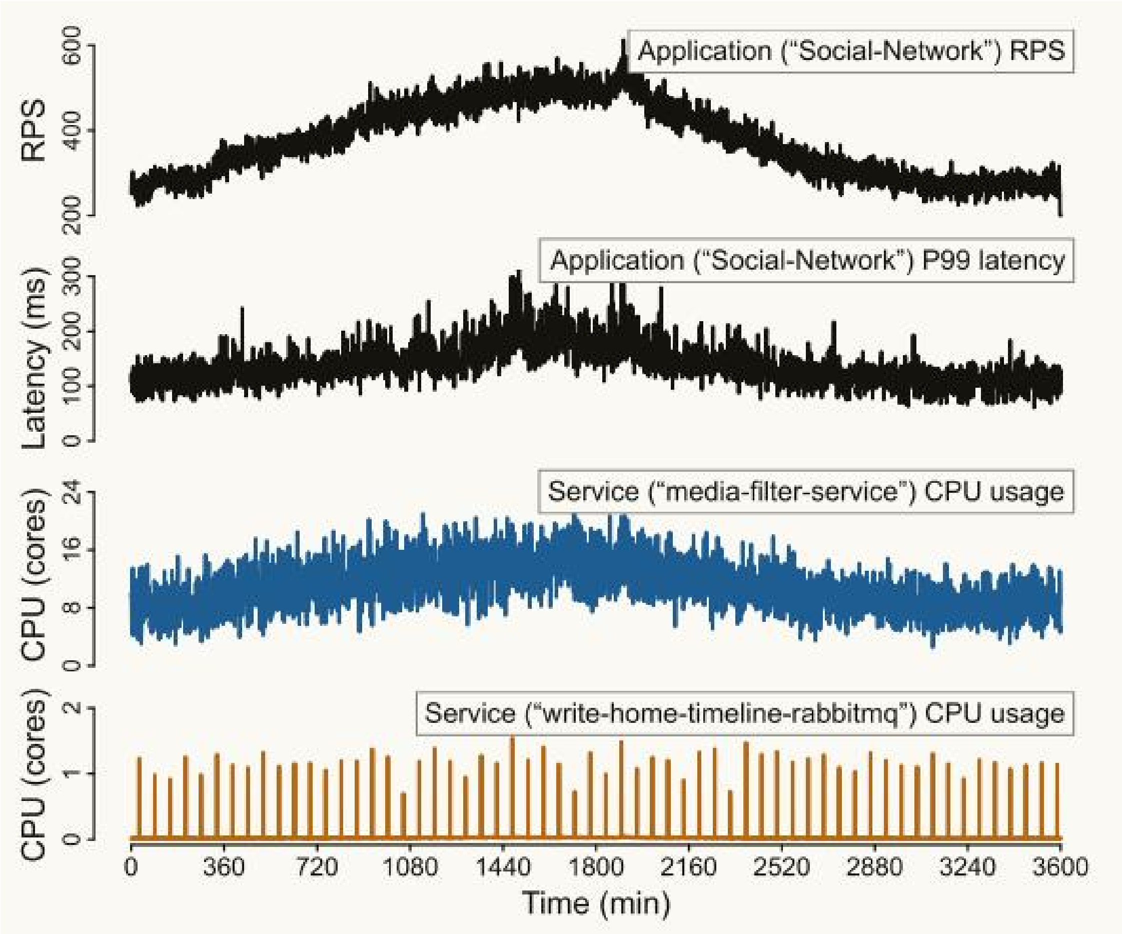
Background

The distributed nature of microservices has brought new difficulties to resource management:

1、由于不同的用户请求对每个服务的压力不同，异构服务可以表现出截然不同的资源使用模式。

2、应用程序性能和每个服务的资源使用情况是不同级别的度量，不一定表现出很强的相关性。

3、在观察分配变化对端到端性能的影响时会产生不希望的延迟。



Background

How does this article address these issues?

本文采用了分布式系统行为不同的级别，以及应用程序级SLO反馈和服务级资源控制的体系结构解耦机制，为慢速目标微服务设计了Autothrottle。

What is the goal?

最小化基于微服务的应用程序的总CPU分配，同时避免违反用户请求的延迟SLO。

Dictionary

- Background
- **Design**
- Evaluation
- Related work
- Conclusion
- Think

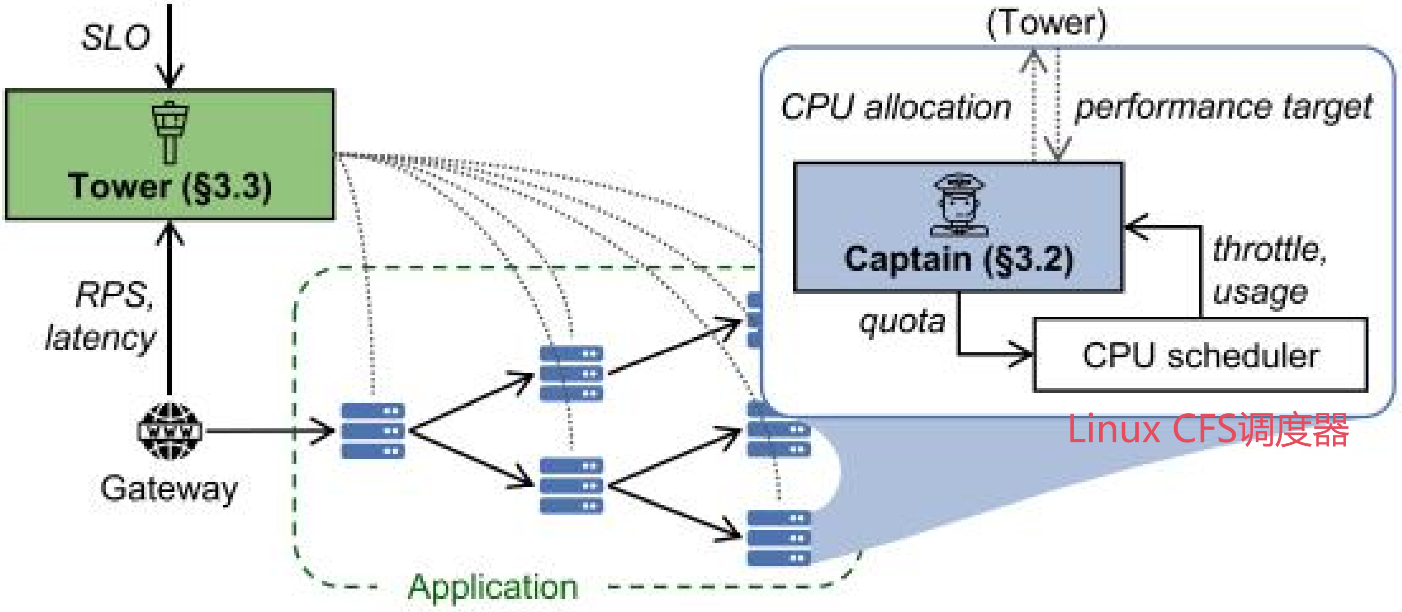
Design

Autothrottle框架

服务控制器——Captain

应用范围控制器——Tower

连接桥梁——CPU throttle ratios



Design

CPU Throttle Ratio（CPU节流比例）：

公式：

$$\text{CPU Throttle Ratio} = \frac{\Delta\text{nr\_throttled}}{\text{Number of CFS Periods in Window}}$$

例：

假设在1秒（10个CFS周期）内，nr_throttled从50增加到55，则节流次数增量Δ=5，节流比例 = 5/10=0.55/10=0.5，即50%的时间窗口内发生了节流。

Design

- **Captain**

1、Multiplicative scale-up

乘法放大，进一步使增量的大小与测量的CPU节流比和目标比率之间的差异成比例。

Algorithm 1: Captain: scaling up and down

```
/* executes every N periods */
throttleCount = throttle count during last N periods;
throttleRatio = throttleCount / N;
margin = max(0, margin + throttleRatio − throttleTarget);
if throttleRatio > α × throttleTarget then
    /* multiplicatively scale up */
    quota = quota × (1 + throttleRatio − α × throttleTarget);
else
    /* instantaneously scale down */
    history = CPU usage history in the last M periods;
    proposed = max(history) + margin × stdev(history);
    if proposed ≤ β_max × quota then
        quota = max(β_min × quota, proposed);
    end
end
```

Design

- **Captain**

1、Multiplicative scale-up

通过调整α，管理人员可以平衡以下两个目标：

- 快速响应真实高负载（α<1）
- 避免错误操作扩大CPU配额（α>1）

Algorithm 1: Captain: scaling up and down

/* executes every N periods */
throttleCount $=$ throttle count during last N periods;
throttleRatio $=$ throttleCount$/N$;
margin $=$ `max`$(0,$ margin $+$ throttleRatio $-$ throttleTarget$)$;
if throttleRatio $> \alpha \times$ throttleTarget **then**
    /* multiplicatively scale up */
    quota $=$ quota $\times$ $(1 +$ throttleRatio $- \alpha \times$ throttleTarget$)$;
else
    /* instantaneously scale down */
    history $=$ CPU usage history in the last M periods;
    proposed $=$ `max`(history) $+$ margin $\times$ `stdev`(history);
    if proposed $\leq \beta_{max} \times$ quota **then**
        quota $=$ `max`$(\beta_{min} \times$ quota, proposed$)$;
    end
end

Design

- **Captain**

2、Instantaneous scale-down

瞬时缩放，防止CPU配额突然发生很大的变化，避免在工作负载高峰期间对短暂的平静反应过度;反之亦然。确保资源管理既敏捷又可控，增加系统的稳定性。

Algorithm 1: Captain: scaling up and down

/* executes every N periods */
throttleCount = throttle count during last N periods;
throttleRatio = throttleCount$/N$;
margin = max(0, margin + throttleRatio − throttleTarget);
if throttleRatio $>$ $\alpha \times$ throttleTarget **then**
  /* multiplicatively scale up */
  quota = quota $\times$ (1 + throttleRatio − $\alpha \times$ throttleTarget);
else
  /* instantaneously scale down */
  history = CPU usage history in the last M periods;
  proposed = max(history) + margin $\times$ stdev(history);
  if proposed $\leq \beta_{max} \times$ quota **then**
    quota = max($\beta_{min} \times$ quota, proposed);
  end
end

Design

- **Captain**

3、Rollback mechanism after scaling down

恢复“鲁莽”的缩减。最后加上等于两个配额之差的额外CPU配额，分配多一点的CPU，以考虑由于错误的缩小而可能发生的潜在处理延迟。

Algorithm 2: Captain: rollback mechanism

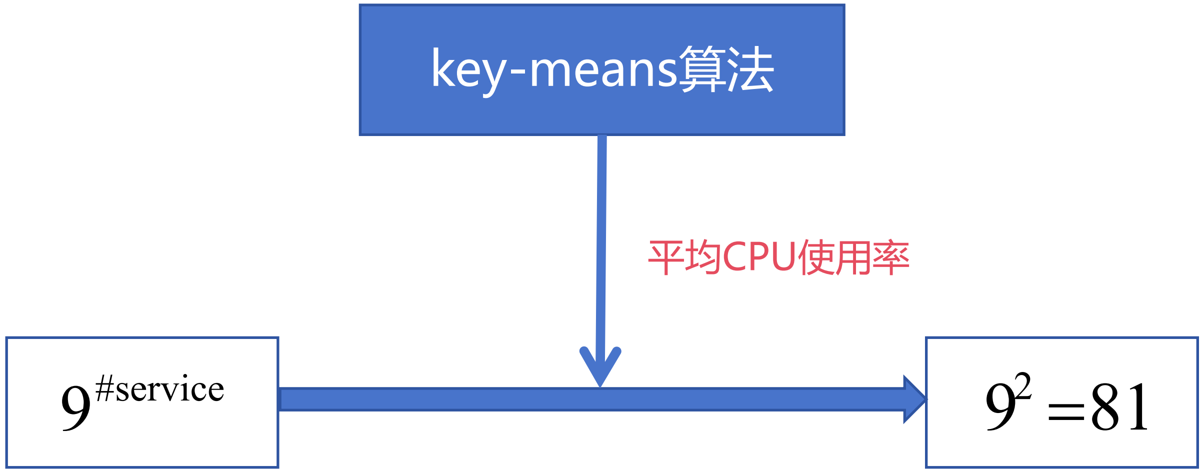
/* executes every period for N periods after each scale-down */
lastQuota = CPU quota before scale-down;
throttleCount = throttle count since scale-down;
throttleRatio = throttleCount/N;
if throttleRatio $>$ $\alpha \times$ throttleTarget **then**
    /* revert to the previous (higher) quota before scale-down with an additional allocation equal to the quota difference */
    quota = lastQuota + (lastQuota − quota);
    margin = margin + throttleRatio − throttleTarget;
end

Design

- **Tower**

1、使用Contextual Bandits在线学习，根据RPS动态调整每个服务的CPU节流目标，在满足延迟SLO的同时尽量少用CPU资源。

2、减少行动空间



Dictionary

- Background
- Design
- **Evaluation**
- Related work
- Conclusion
- Think

Evaluation

1、基准应用程序（三个微服务应用测试对象）

- Train-Ticket
- Hotel-Reservation
- Social-Network

2、比较基线（对照组）

- Kubernetes (K8s-CPU and K8s-CPU-Fast)
- Sinan（ML方案）

3、四种流量模式：

- Diurnal
- Constant
- Noisy
- Bursty

4、长期测试：使用21天真实云厂商日志，验证策略的长期稳定性。

5、工具选择：使用Locust

Evaluation

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	30.4	58.0 (↓47.59%)	41.2 (↓26.21%)	278.4 (↓89.08%)
Constant	21.7	24.8 (↓12.50%)	27.3 (↓20.51%)	279.9 (↓92.25%)
Noisy	15.5	23.6 (↓34.32%)	17.7 (↓12.43%)	251.8 (↓93.84%)
Bursty	17.7	27.1 (↓34.69%)	21.9 (↓19.18%)	268.3 (↓93.40%)

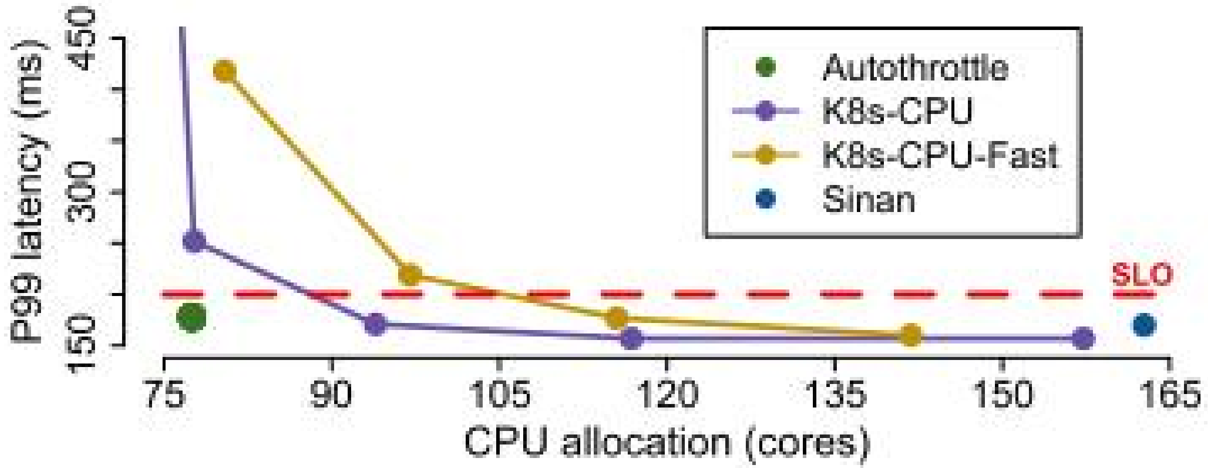
(a) Train-Ticket application (SLO: 1,000 ms P99 latency)

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	77.5	93.9 (↓17.47%)	115.5 (↓32.90%)	162.7 (↓52.37%)
Constant	88.7	115.6 (↓23.27%)	118.8 (↓25.34%)	149.7 (↓40.75%)
Noisy	57.5	66.5 (↓13.53%)	105.1 (↓45.29%)	105.2 (↓45.34%)
Bursty	50.0	67.5 (↓25.93%)	99.7 (↓49.85%)	111.9 (↓55.32%)

(b) Social-Network application (SLO: 200 ms P99 latency)

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	15.3	15.7 (↓2.55%)	16.5 (↓7.27%)	45.5 (↓66.37%)
Constant	11.2	11.5 (↓2.61%)	11.3 (↓0.88%)	21.2 (↓47.17%)
Noisy	10.8	12.1 (↓10.74%)	11.6 (↓6.90%)	65.9 (↓83.61%)
Bursty	10.1	15.7 (↓35.67%)	10.9 (↓7.34%)	63.1 (↓83.99%)

(c) Hotel-Reservation application (SLO: 100 ms P99 latency)



在160核集群上的实验结果，Autothrottle在所有应用程序中都优于基线。

将相当数量的CPU分配给Autothrottle时，K8s-CPU和K8s-CPU- fast将违反SLO

Evaluation

计算：

- P99延迟与CPU节流的Pearson相关系数
- P99延迟与CPU利用率的Pearson相关系数

现象：

CPU节流比CPU利用率表现出和P99延迟更高的相关性，存在更强的线性关系。

结论：

CPU节流与应用程序延迟的高相关性，所以我们使用CPU节流作为中间代理指标。

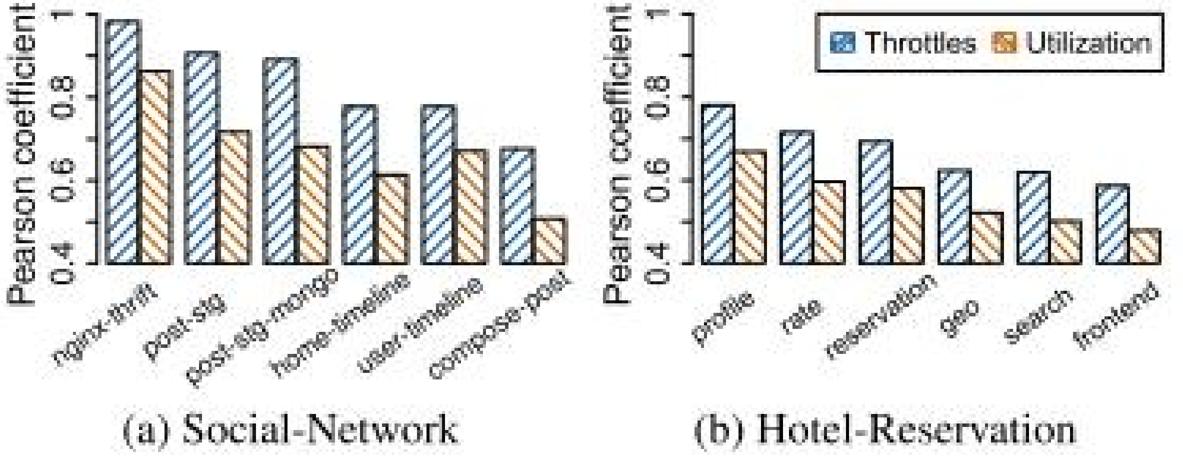
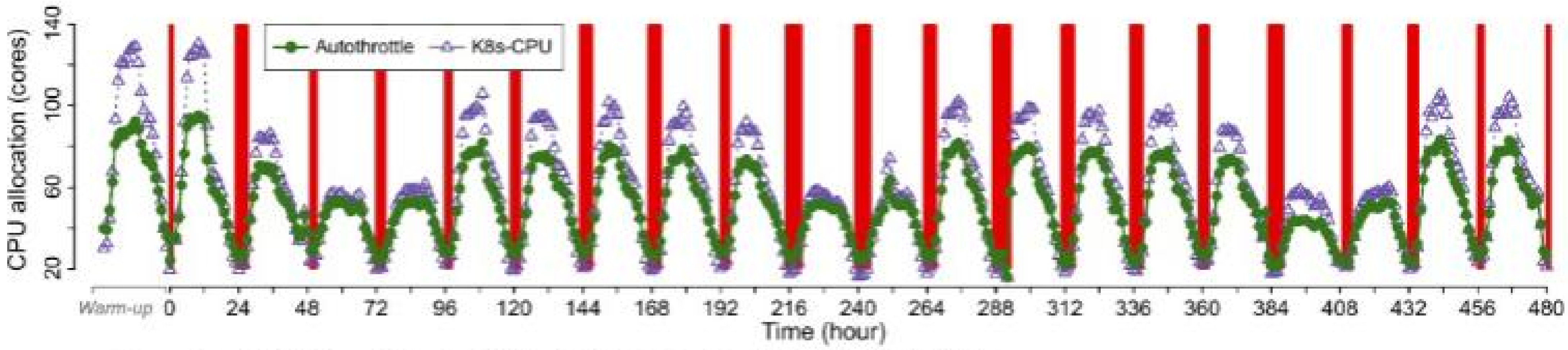


Figure 7: As a proxy metric, CPU throttles exhibit a higher correlation with application latencies than CPU utilization. The figure shows top microservices with highest CPU usage.

Evaluation

1、Long-term evaluation

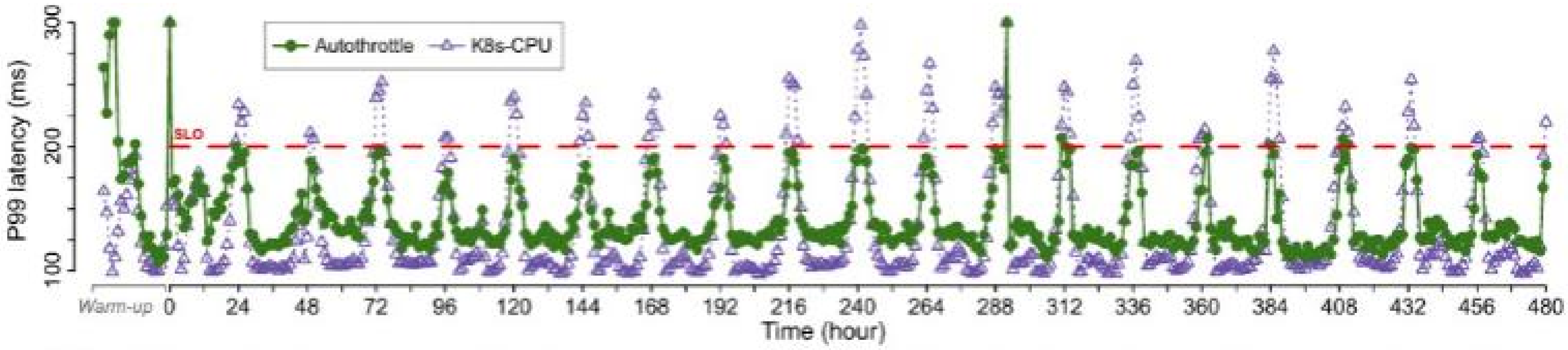


(a) Autothrottle分配的cpu和K8s-CPU基线。红框表示K8s-CPU违反SLO规定的小时数。

- **将Autothrottle与性能最好的基准K8s-CPU进行比较**

Evaluation

1、Long-term evaluation



(b) Social-Network's P99 latency, as achieved by Autothrottle and the K8s-CPU baseline. Dashed red line illustrates the 200 ms SLO.

- **该图显示了社交网络每小时的P99延迟**
- **观察结果是：Autothrottle能够连续地保持接近200毫秒SLO的P99延迟**
- **结论：应用程序使用Autothrottle性能更稳定**

Evaluation

2、Large-scale evaluation

现象：Autothrottle需要分配的CPU内核更少

与性能最好的基准K8s-CPU和K8s-CPU-fast相比，Autothrottle最多可节省28.24%CPU内核和至少5.92%CPU内核。

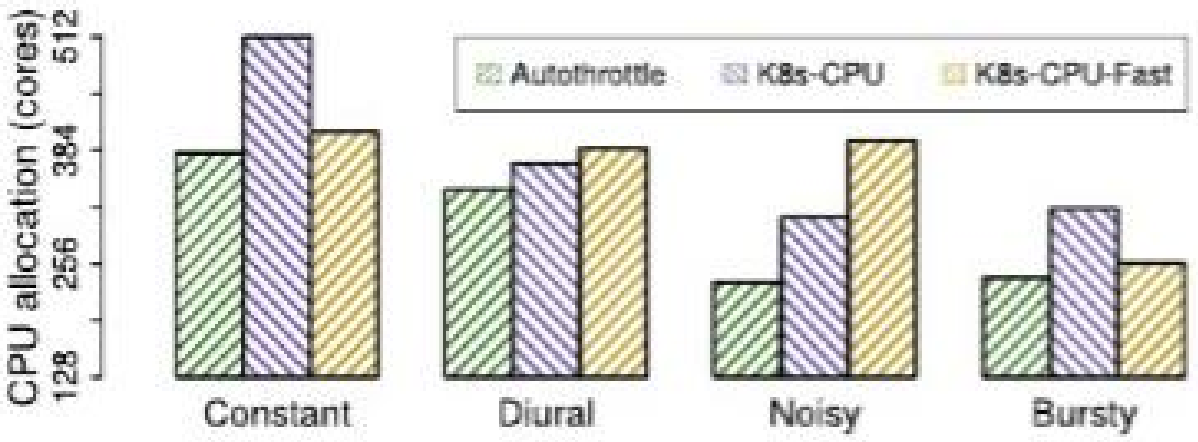


图10:Autothrottle和基线分配的CPU核数，以满足Social-Network的P99 SLO。图中显示了Autothrottle在512核集群上的可伸缩性。

Dictionary

- Background
- Design
- Evaluation
- **Related work**
- Conclusion
- Think

Related work

1、垂直扩展

方法	机制	创新/局限
Kubernetes VPA	基于利用率阈值启发式调整	简单易用，依赖静态阈值
Autopilot	历史数据 + multi-armed bandit选择	动态适应，需长期数据积累
Sinan	ML预测SLO违规概率	精准但模型训练成本高
FIRM	RL定位根本原因并垂直扩展	针对SLO根本原因，策略收敛慢
Autothrottle	双层设计和CPU节流比例指标	实时响应，直接保障网络SLO

Related work

2、代理指标

传统指标	问题	Autothrottle方案
CPU利用率	高利用率≠SLO违规	CPU节流指标：直接反映资源竞争
队列长度	忽略单个请求复杂度，队列分散使测量难	闭环控制动态调整配额，保障吞吐量
排队延迟	取决于线程模型，需手动测试	无侵入监控，自适应阈值
总结	传统指标滞后/误导	创新：节流信号 + 实时反馈

Related work

3、横向与混合扩展

方法	机制	适用场景
Kubernetes HPA	基于利用率/QPS调整副本数	通用场景，响应延迟高
GRAF	图神经网络建模服务依赖	复杂依赖系统，计算开销大
COLA	多服务协同调整	避免局部优化，需全局协调
混合扩展	垂直优先（短期）+水平（长期）	平衡速度与弹性，策略设计复杂

Dictionary

- Background
- Design
- Evaluation
- Related work
- **Conclusion**
- Think

Conclusion

Autothrottle是一个两级学习辅助资源管理框架，用于慢速目标微服务。

微服务架构中，端到端延迟由多个服务共同决定，但直接通过延迟调整每个服务的资源分配面临以下挑战：

1、延迟反馈滞后（例如请求需经过多个服务处理）。

2、不同服务对延迟的贡献差异大（某些服务可能是瓶颈）。

Autothrottle通过CPU节流比作为中间代理指标，将全局的延迟SLO转化为每个服务的本地资源控制目标。

Dictionary

- Background
- Design
- Evaluation
- Related work
- Conclusion
- **Think**

Think

- 目前仅针对CPU资源进行优化，而实际系统中内存、网络带宽或磁盘I/O可能同样关键，未来可扩展框架以支持多种资源（例如内存）联合优化。
- 论文假设请求类型分布恒定，但在实际场景中，请求类型的动态变化可能导致性能目标失效。可探索基于请求类型的细粒度目标调整，或引入请求分类器作为上下文输入。

Thank you!