nsdi '24

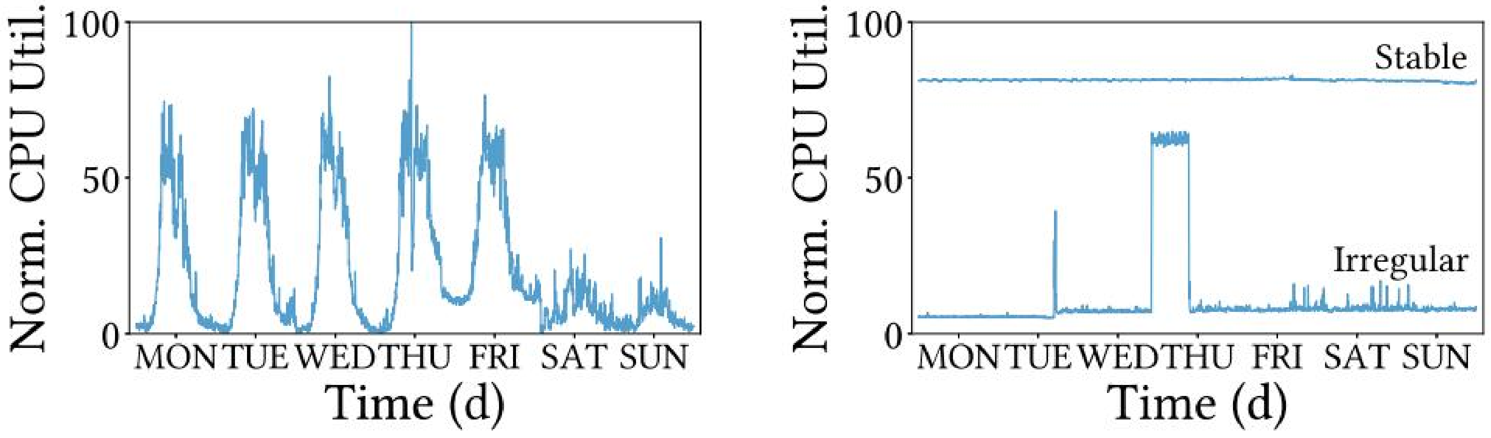
UFO: The Ultimate QoS-Aware CPU Core Management for Virtualized and Oversubscribed Public Clouds

*Yajuan Peng, *Shuang Chen(*equal contribution), Yi Zhao, Zhibin Yu

中国科学院深圳先进技术研究院
SHENZHEN INSTITUTES OF ADVANCED TECHNOLOGY CHINESE ACADEMY OF SCIENCES

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Most cloud data centers operate at **very low resource utilization**.



(a) Diurnal (b) Stable and Irregular

VM CPU utilization patterns in Azure Cloud

(Xiaoting Qin: DSN'23)

Low CPU utilization < 20%

Multi-tenancy | Virtualization | Oversubscription

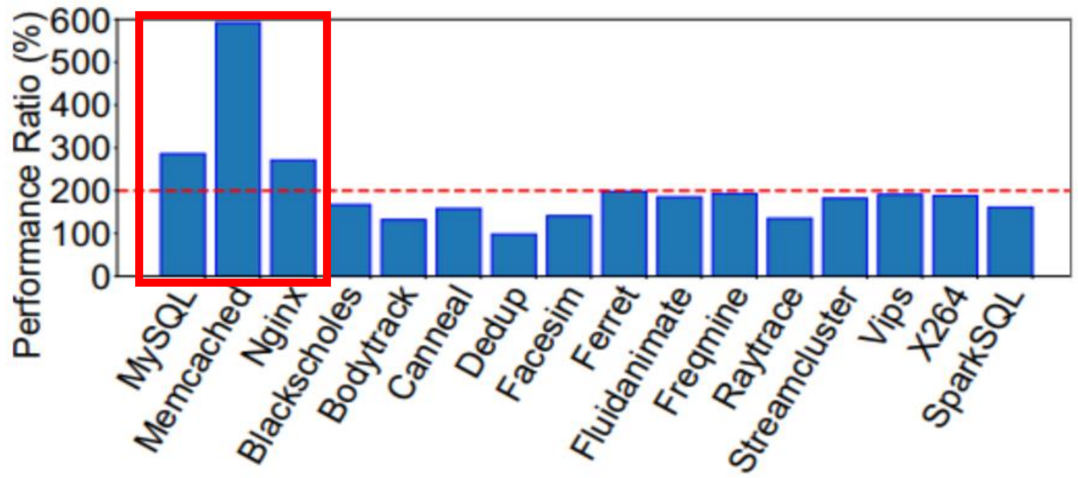
Cloud Applications

Best-effort Applications	Latency-critical Applications
• Throughput-oriented	• Tail latency
• No latency constraint	• Strict QoS constraint

Google Maps

Due to double scheduling, **LC applications** suffer up to **10** times worse.

Previous studies for virtualization optimization focused on **BE**(Best-effort) applications, and are **ineffective** for **LC**(Latency-critical) applications.



name	publication	year	benchmark
Revisiting VM-Agnostic ...	TPDS	2023	parsec3.0, mosbench...
PLE-KVM	VEE	2021	parsec3.0, mosbench...
Virtualization Overhead ...	TPDS	2021	PARSEC, SPLASH2X
Flexible Micro-sliced Cores ..	EuroSys	2018	gmake,swaptions,dedup...
eCS	USENIX ATC	2018	Apache,Psearchy,Pbzip2...

Previous work on LC colocation relies on **application-level inputs** to guide **QoS-aware** resource management.

Algorithm 1 ARQ Resource Scheduling Algorithm.

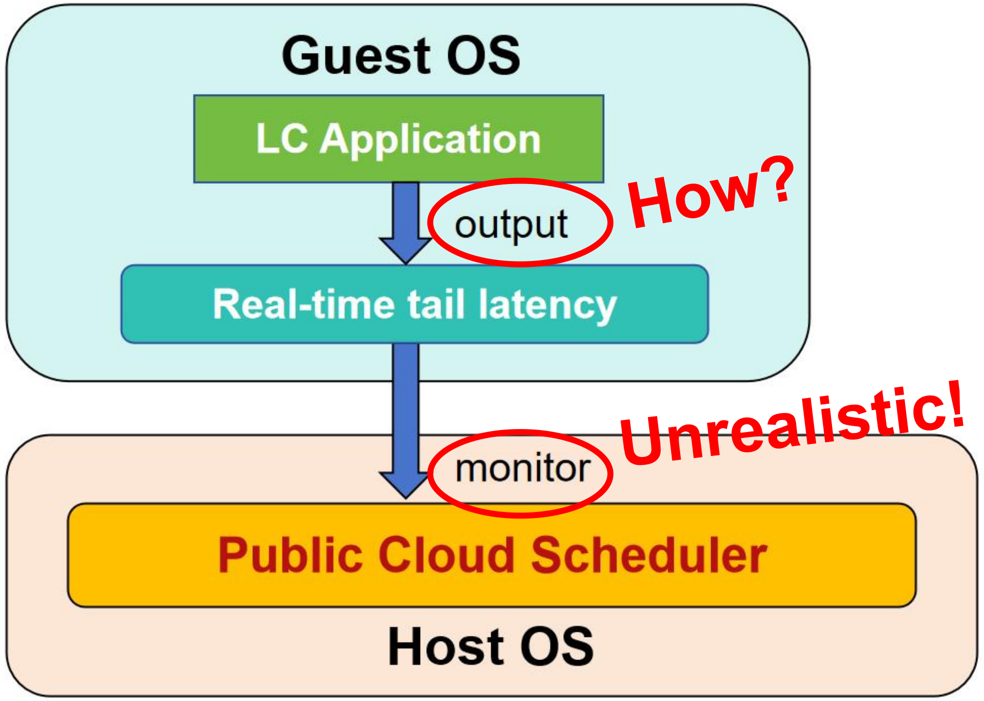
```
function ARQ
    isAdjust ←False, E_S ← 1
    while True do
        Monitor the tail latency values of the LC applications and the IPC values of BE applications periodically
        E'_S ← E_S
        E_S ← computeEntropy()
        // ReT is an array, the elements of which are the remaining tolerance of each LC application.
        ReT ← computeRemainingTolerance()
```

Ah-q (HPCA’23)

Algorithm 1: PARTIES’ main function.

```
// Start from fair allocation of all resources
initialization();
while TRUE do
    monitor tail latency and resource utilization for 500ms;
    adjust_network_bandwidth_partition();
    for each application A do
        slack[A] ← (target[A] - latency[A]) / target[A];
    end
```

PARTIES (ASPLOS’19)



Challenges：

Public Cloud

Virtualization | Oversubscription | QoS-Aware

1) Coordination between Host OS and Guest OS

LC performs 10x worse than BE applications due to the double scheduling problem.

2) Coordination between vCPU Threads and Emulator Threads

LC applications are subject to internal resource contention within a VM.

3) Coordination between Host Core Manager and Guest Applications

No application-level performance metrics inside VMs to help manage resources.

Challenges：

1) Coordination between Host OS and Guest OS

LC performs 10x worse than BE applications due to the double scheduling problem.

2) Coordination between vCPU Threads and Emulator Threads

LC applications are subject to internal resource contention within a VM.

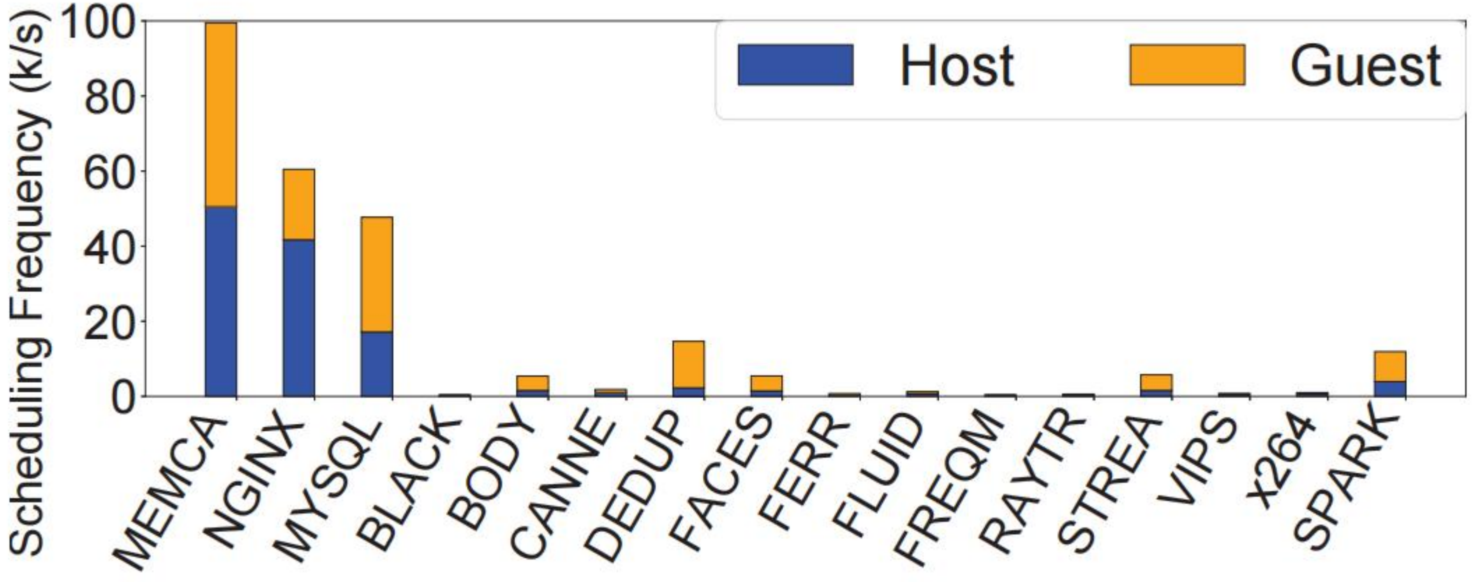
3) Coordination between Host Core Manager and Guest Applications

No application level performance metrics inside VMs to help manage resources.

LC： more context switch overhead

LC applications consists of **numerous sub-millisecond** tasks.

BE applications: **fewer** and **longer** tasks.

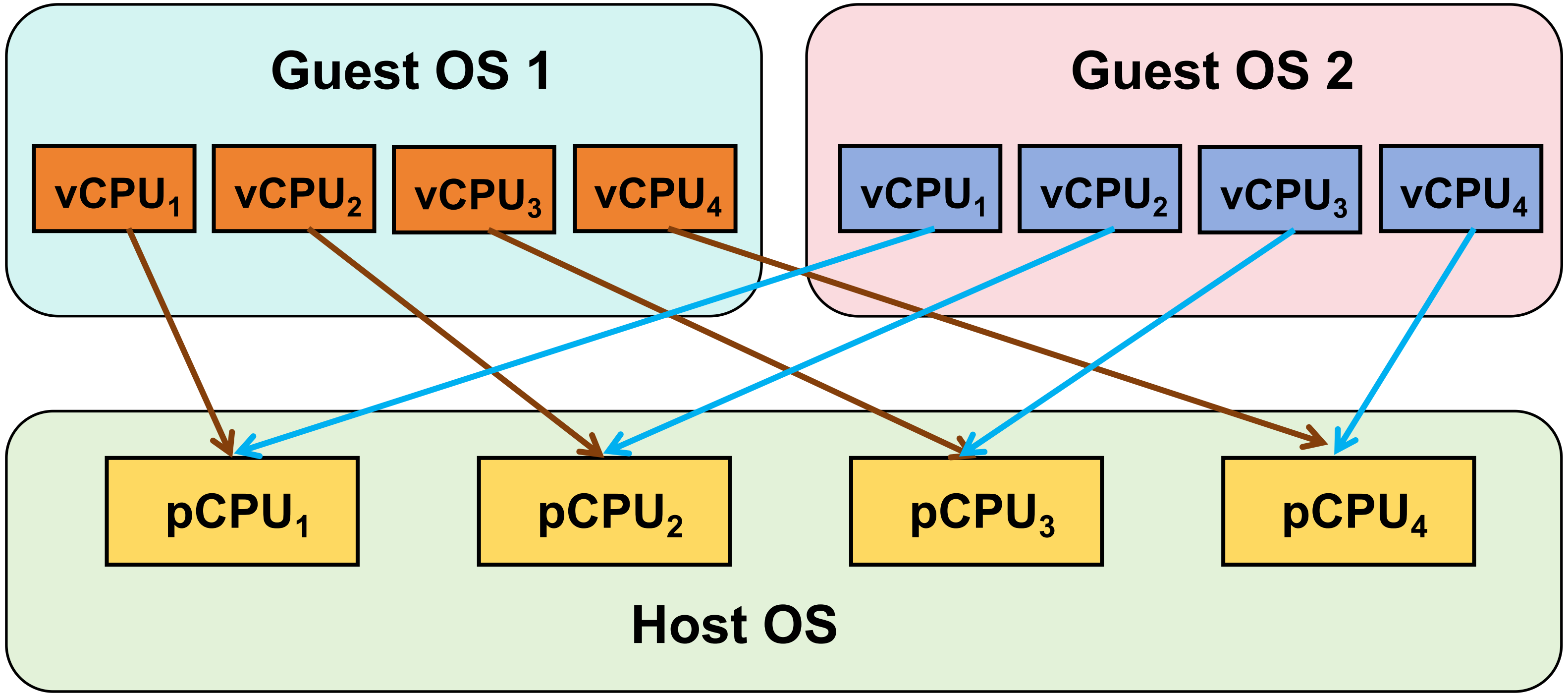


How to fix it?

Default: Rely on the scheduling policy of OS to schedule VMs. All the two VMs share the same 4 pCPUs.

Isolation: Isolate the two VMs, each assigned two pCPUs on the host.

Host-Aware Isolation: On top of **Isolation**, the Guest OS is aware that the VM is allocated with only two pCPUs, and schedules only two vCPUs. (Hot-plug)

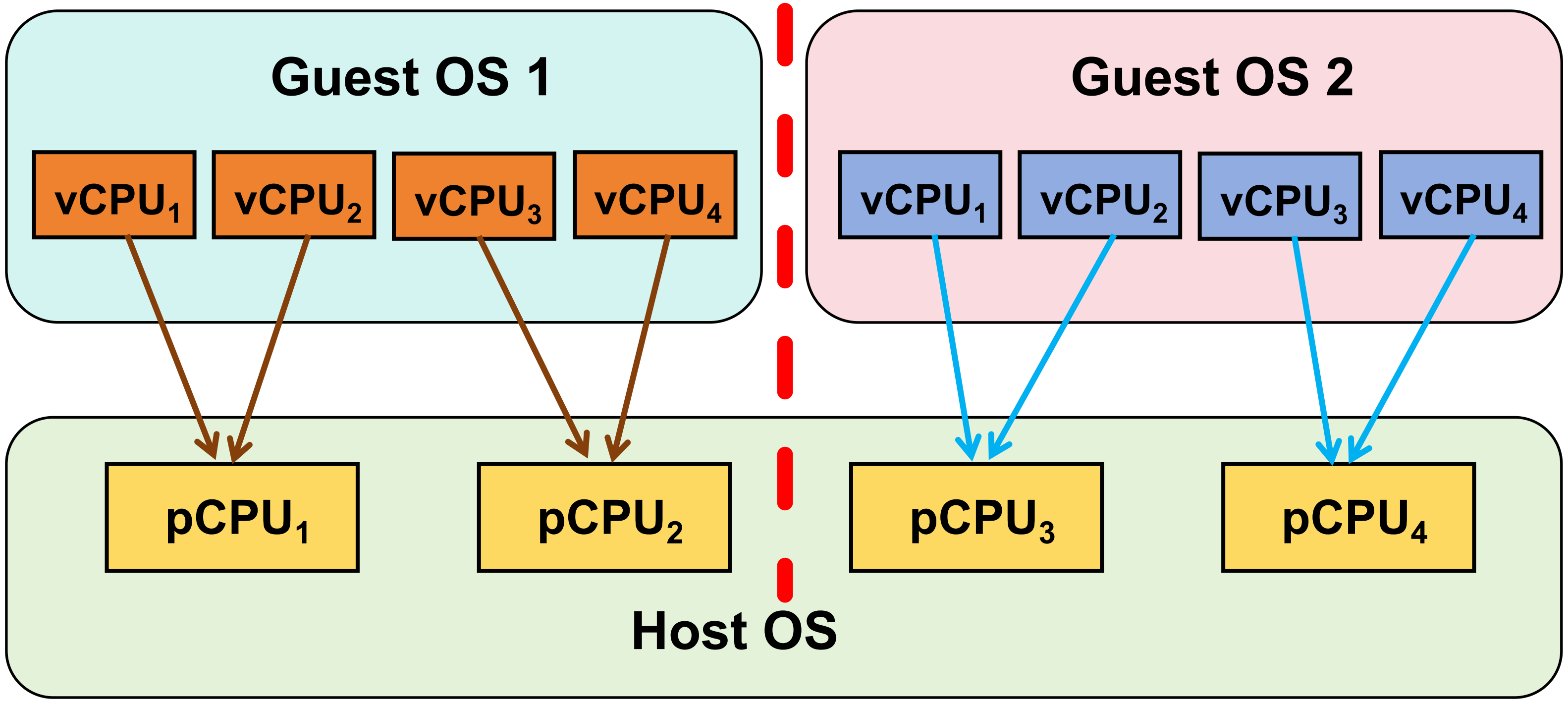


Default

Default: Rely on the scheduling policy of OS to schedule VMs. All the two VMs share the same 4 pCPUs.

Isolation: Isolate the two VMs, each assigned two pCPUs on the host.

Host-Aware Isolation: On top of **Isolation**, the Guest OS is aware that the VM is allocated with only two pCPUs, and schedules only two vCPUs. (Hot-plug)

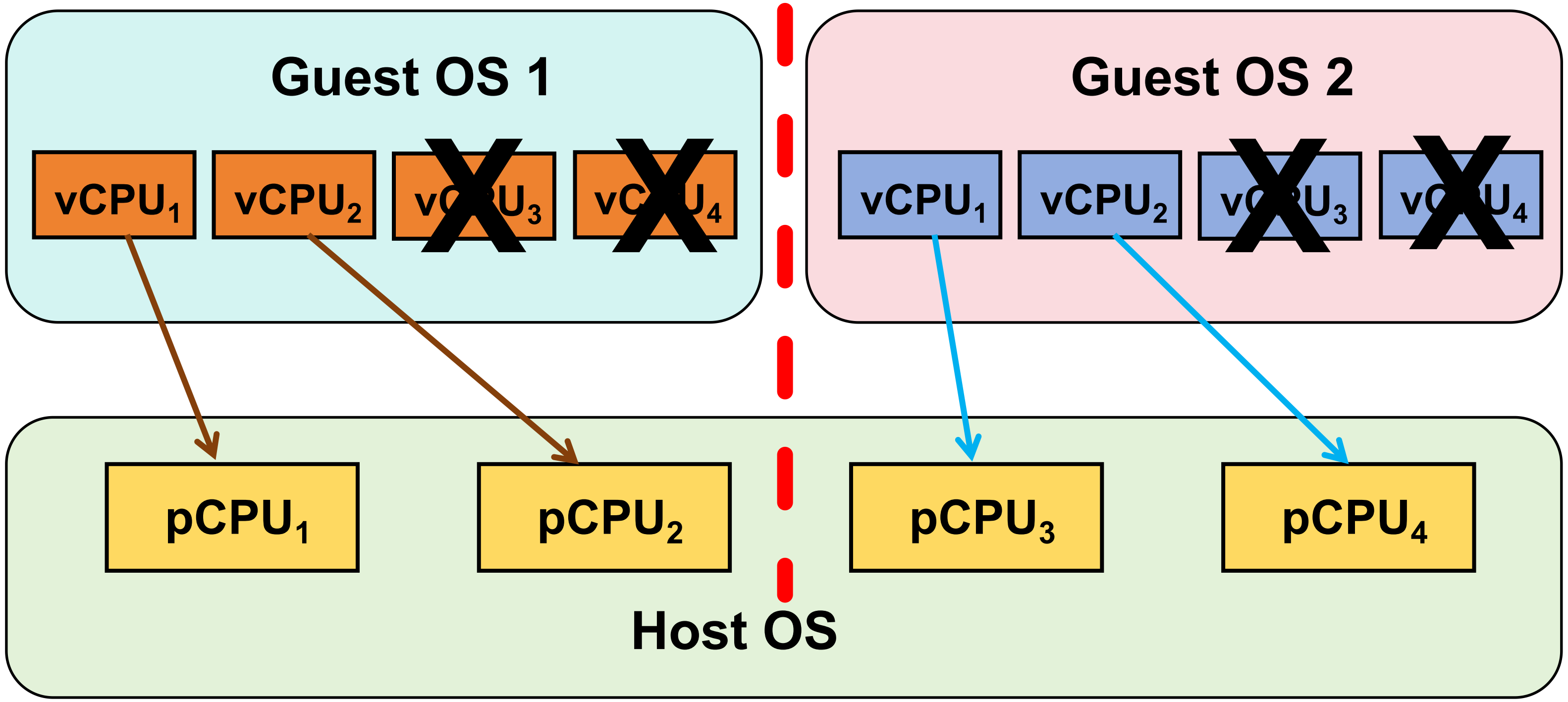


Isolation

Default: Rely on the scheduling policy of OS to schedule VMs. All the two VMs share the same 4 pCPUs.

Isolation: Isolate the two VMs, each assigned two pCPUs on the host.

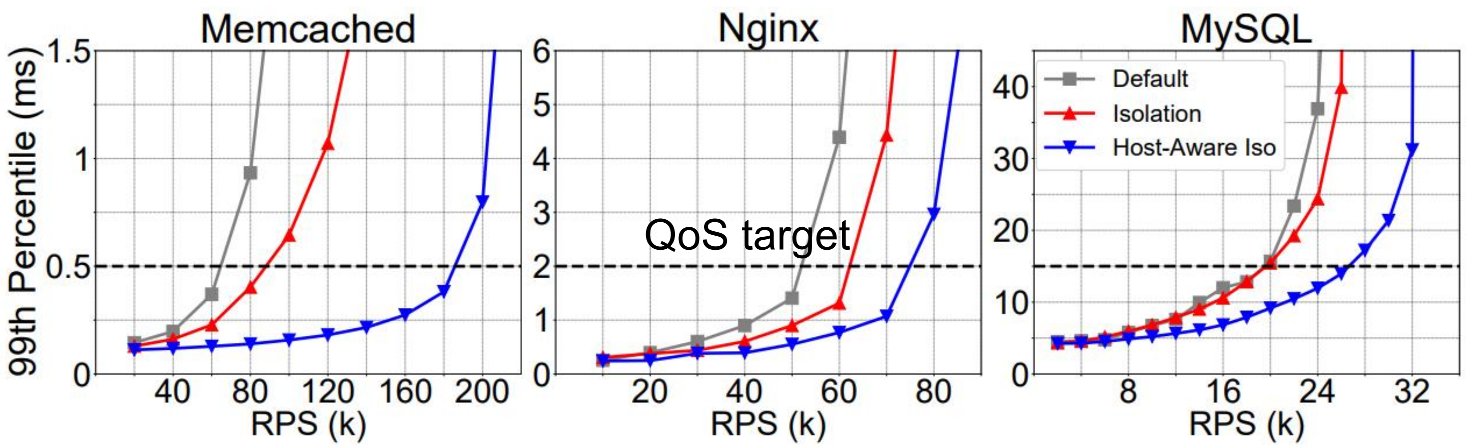
Host-Aware Isolation: On top of **Isolation**, the Guest OS is aware that the VM is allocated with only two pCPUs, and schedules only two vCPUs. (Hot-plug)

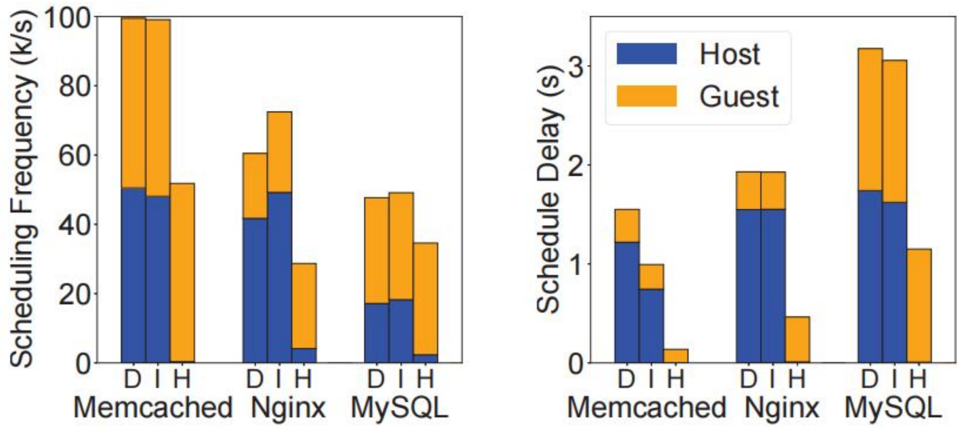


Host-Aware Isolation

Host Aware-Isolation:

Keep the number of vCPU same as pCPU ⇨ Host-Guest coordination





(a) Scheduling Frequency (b) Scheduling Delay

Isolation achieves up to **33%**(average 18%) higher load than **Default**, **Host-Aware Iso** further increases the maximum load under QoS by up to **25% - 125%** than **Isolation**.

Why?

Lower: Schedule Frequency, Schedule Delay, VM Exits, VM Exit Handling Time, Cache Miss.

Challenges：

1) Coordination between Host OS and Guest OS

LC performs 10x worse than BE applications due to the double scheduling problem.

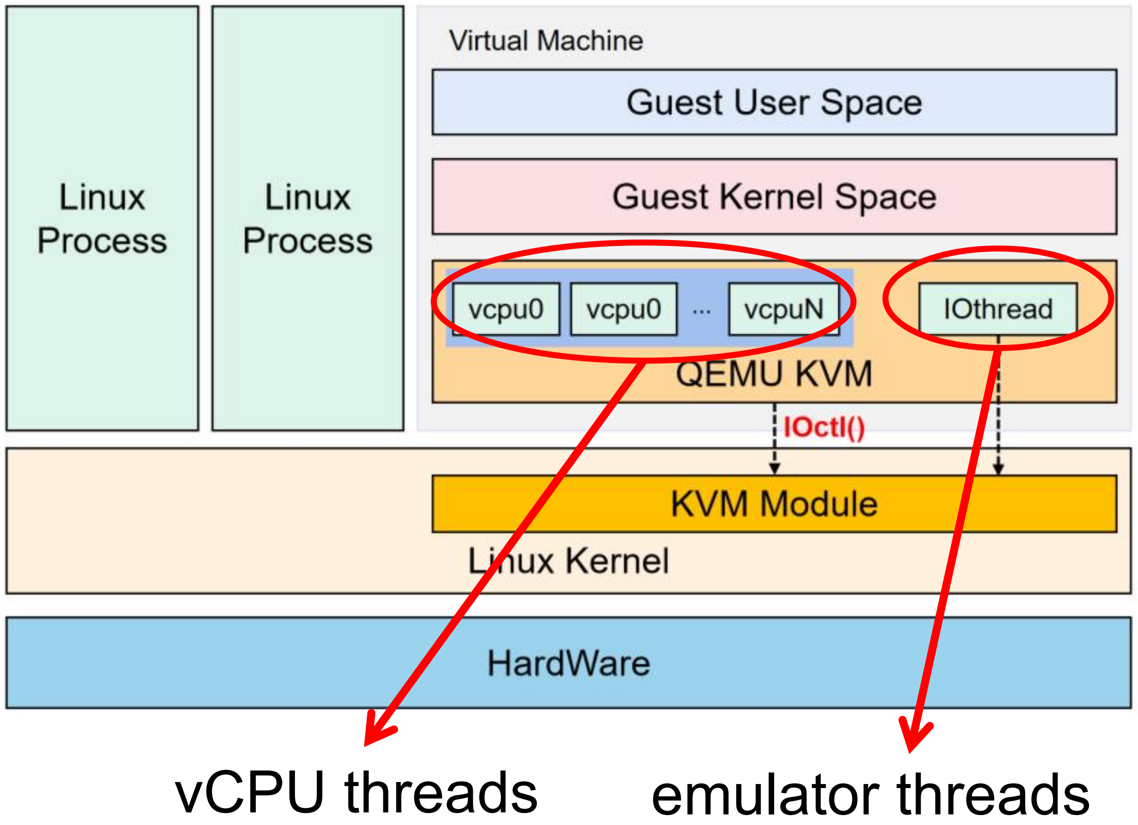
2) Coordination between vCPU Threads and Emulator Threads

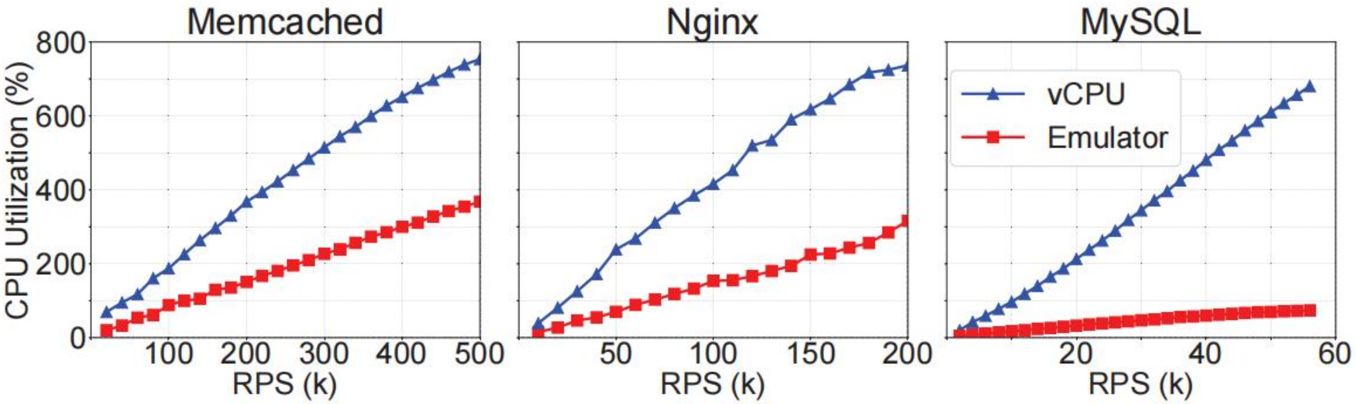
LC applications are subject to internal resource contention within a VM.

3) Coordination between Host Core Manager and Guest Applications

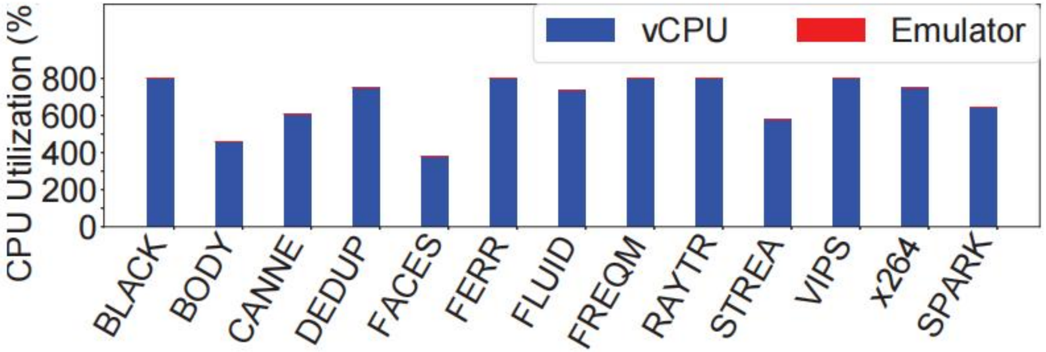
No application-level performance metrics inside VMs to help manage resources.

Emulator threads cause resource contention within a VM.



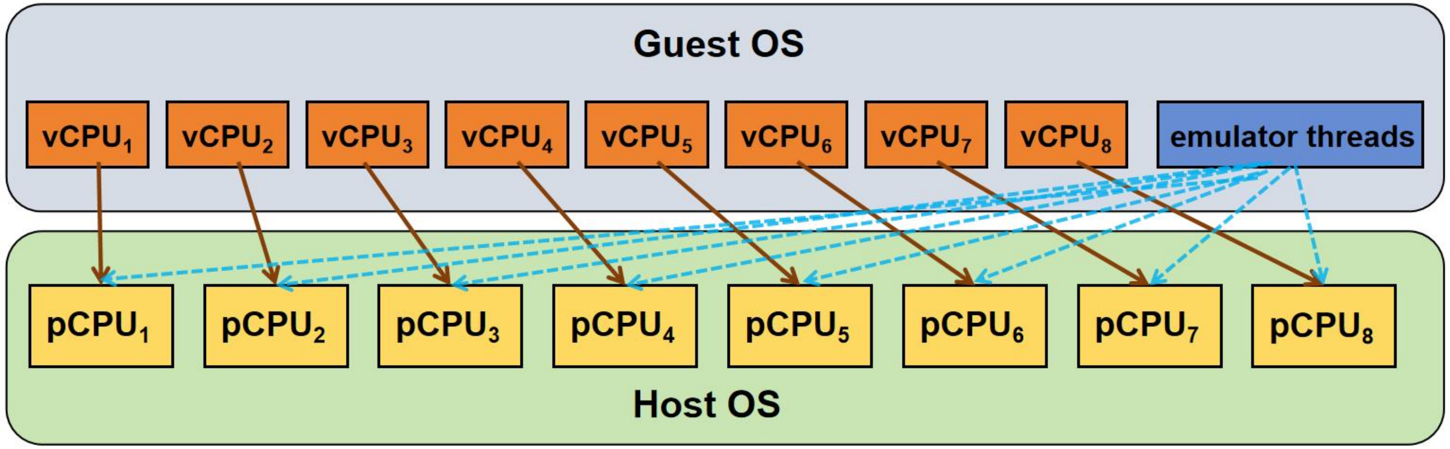


LC: active, related to input load

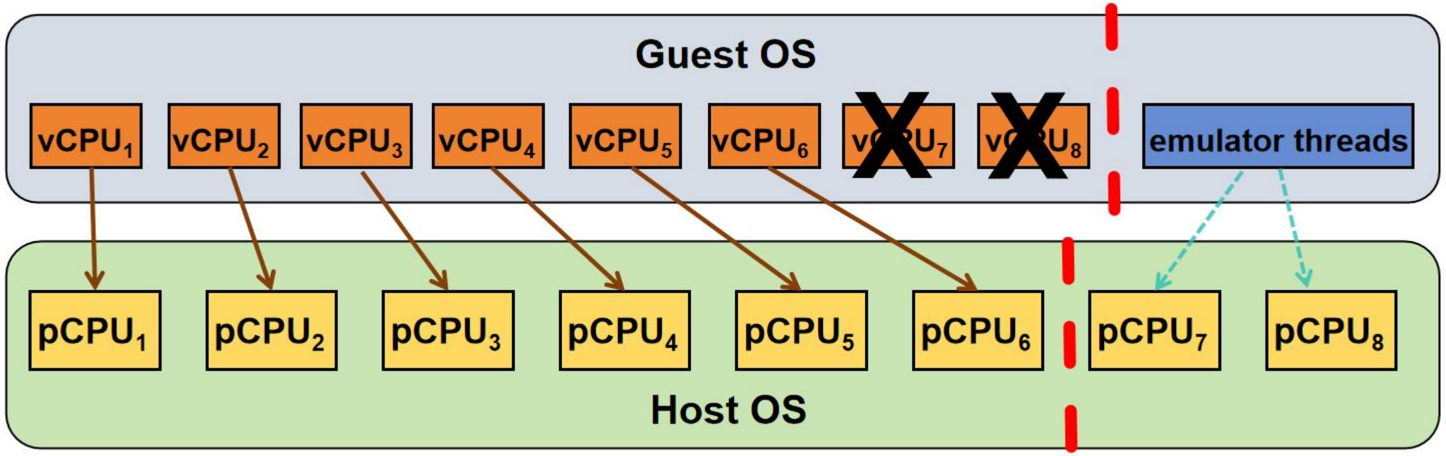


BE: almost have no usage

vCPU thread group and emulator thread group have different core demands, and interfere with each other when sharing cores.



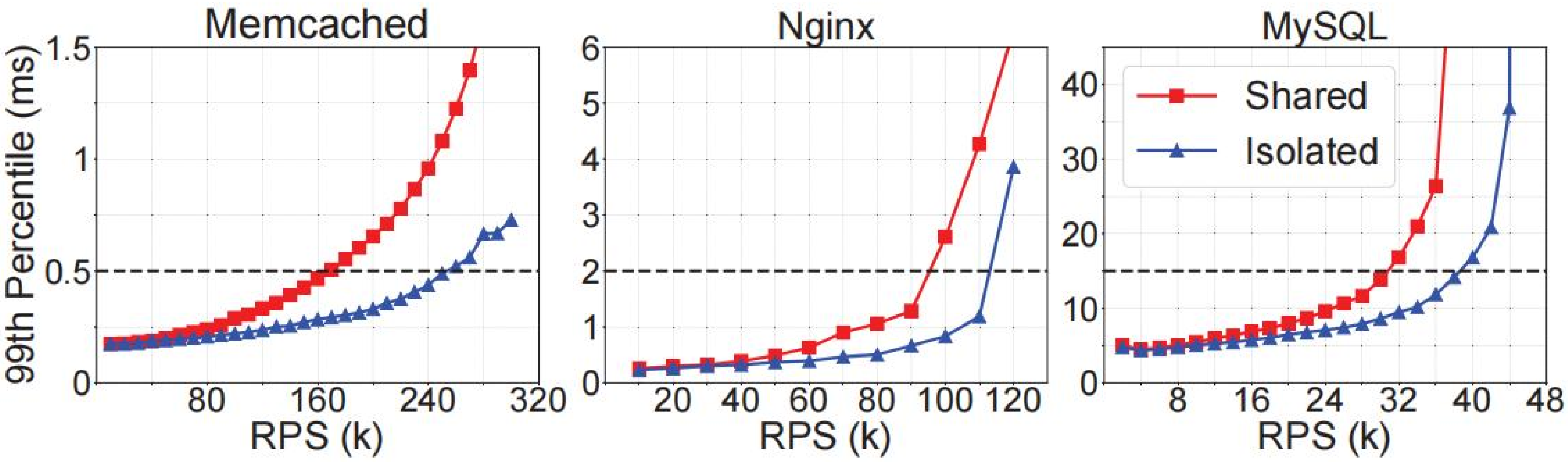
Shared:
vCPU threads share 8 pCPUs with emulator threads.
(Default set)



Isolated:
Partition 8 pCPUs into 6 and 2 cores, and adopt **Host-Aware Isolation** in the vCPU core group.

Isolation inner VM ⇨

Coordination between vCPUs Threads and Emulator Threads



Compared with **Shared**, **Isolated** achieves **15% - 50%** higher input load.

- CPU utilization is a great indicator of application's input load;
- Core allocation of both vCPU and emulator threads should be dynamically adjusted based on input load.

Challenges：

1) Coordination between Host OS and Guest OS

LC performs 10x worse than BE applications due to the double scheduling problem.

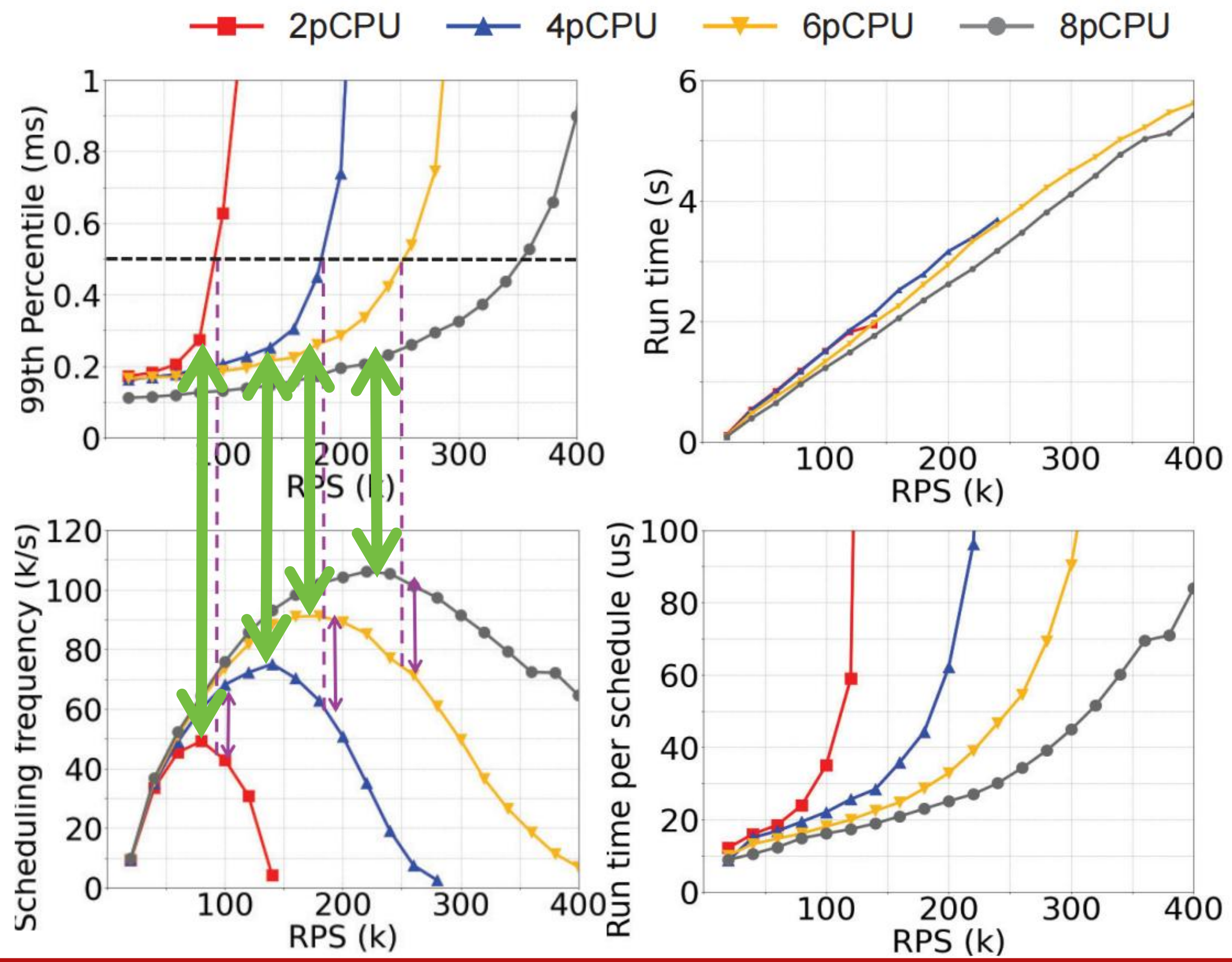
2) Coordination between vCPU Threads and Emulator Threads

LC applications are subject to internal resource contention within a VM.

3) Coordination between Host Core Manager and Guest Applications

No application-level performance metrics inside VMs to help manage resources.

Scheduling Frequency in Guest OS represents p99 ⇨ Coordination between Host Core Manager and Guest Applications



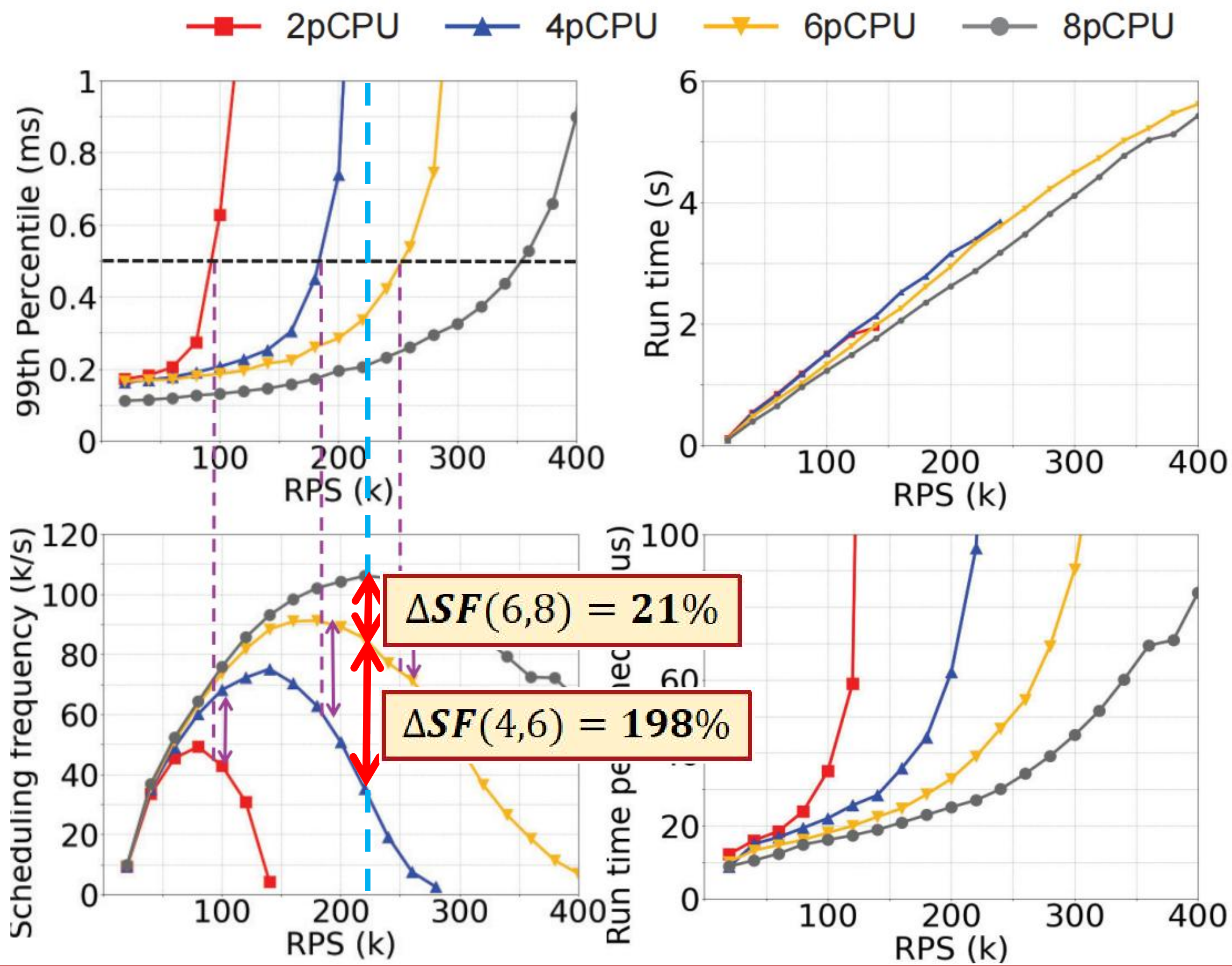
1) SF curve: quadratic function

$$y = ax^2 + bx + c$$

8U: $y = -1.837x^2 + 860.2x + 4713$

2) SF curve's peak point

Scheduling Frequency in Guest OS represents p99 ⇨ Coordination between Host Core Manager and Guest Applications



1) SF curve: quadratic function

$$y = ax^2 + bx + c$$

2) SF curve's peak point

3) Threshold [x]

$$\Delta SF = \frac{SF[c+2] - SF[c]}{SF[c+2]} < x$$

$\Delta SF(4,6) = 198\% > x$

$\Delta SF(6,8) = 21\% < x$

output: 6 cpus

UFO: Feedback, Dynamically, Core allocation

- **Prioritize for LC applications**

UFO's goal is to meet QoS for LC applications through modeling of SF in Guest OS.

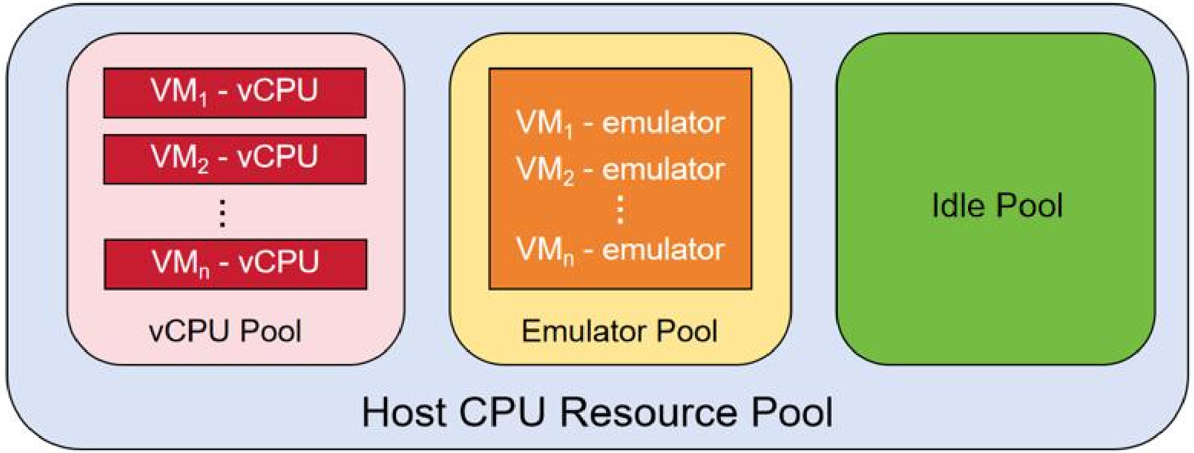
- **Optimize for virtualized and oversubscribed public clouds**

Fix double scheduling through Guest-Host coordination and vCPU-emulator isolation.

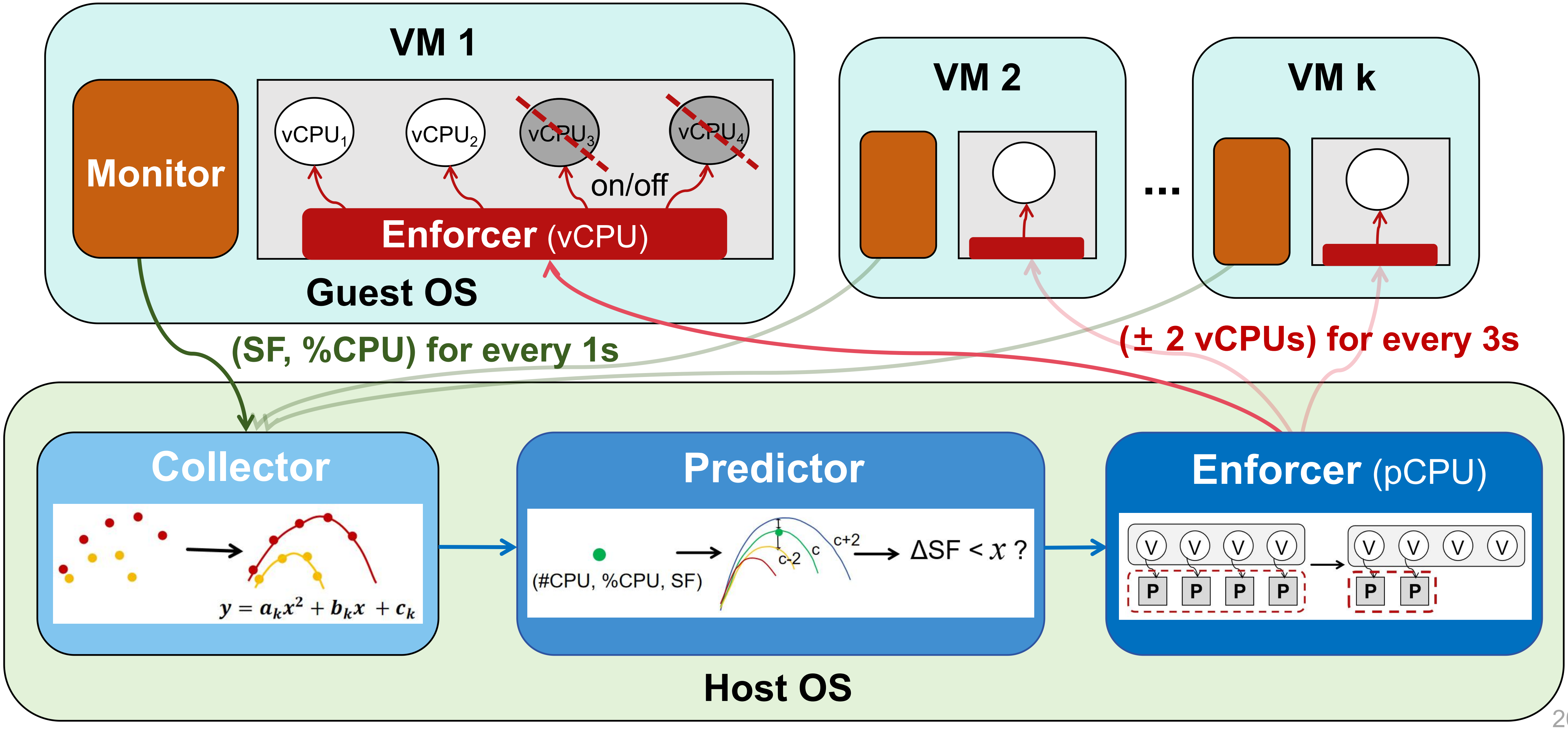
- **Focus on core management**

Higher performance with fewer resources.

- **Accommodate more VMs under QoS on a single host.**



UFO architecture



Experimental Setup

Table 2: Latency-critical applications.

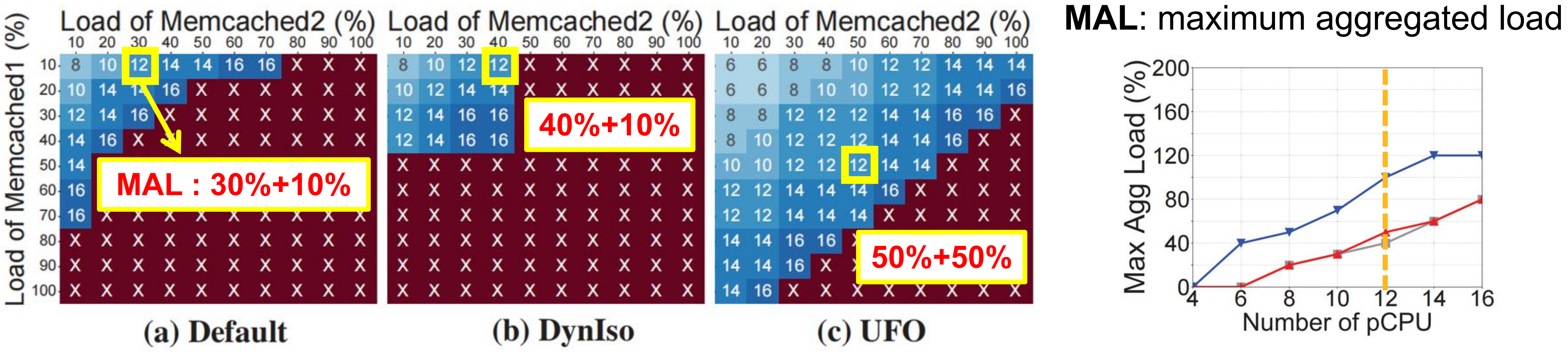
Application	**Memcached**	**Nginx**	**MySQL**
Domain	Key-value store	Web server	Database
QoS Target	0.5ms	2ms	15ms
Max Load under QoS	350k	120k	50k
Load Generator	Mutated	wrk2	sysbench
Dataset	One million <key,value> pairs	10,000 html files of 4KB each	20 tables, each with one million entries
Request Type	100% GET requests	Get file content	OLTP transactions, each with 18 select and 2 update queries

VM Size: 8 vCPU, 16 GB memory

Hyperthreading: Enabled

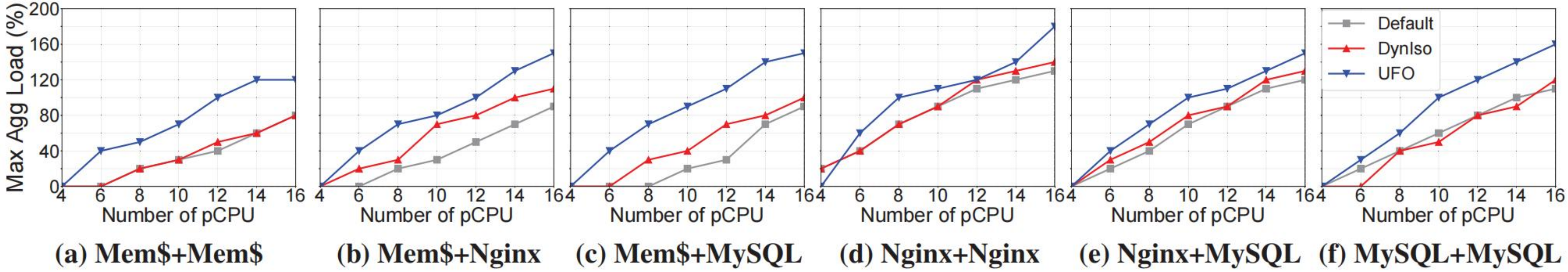
Baselines: Default and DynIso

Constant Load: Colocation of 2 VMs, evaluate resource efficiency.



UFO achieves up to 60% higher MAL than DynIso under same #pCPUs.

UFO saves up to 50% cores than DynIso under the same input load.



Dynamic Load: Colocation of 3 VMs, evaluate fluctuating load.

Nginx: Diurnal load fluctuations [1]

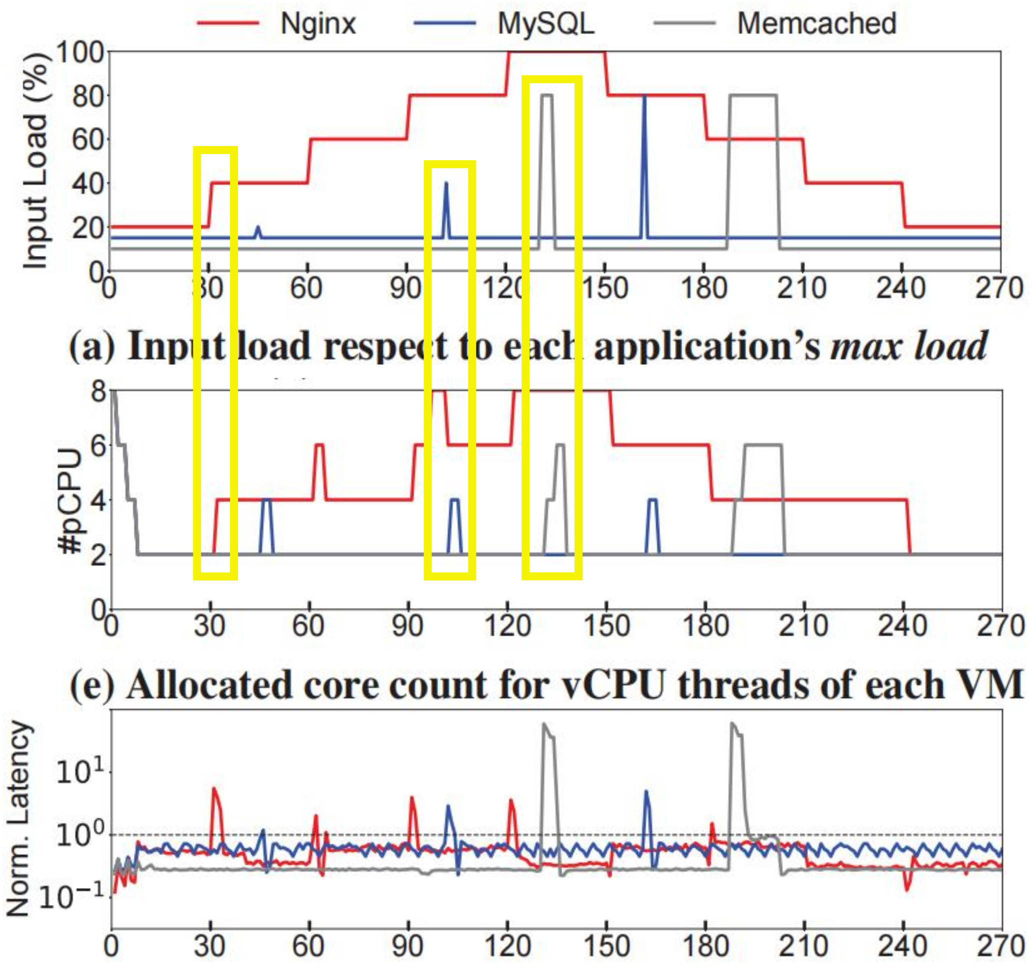
- UFO reacts to one second after any load change is detected, and performs better as more samples are collected.

MySQL: Sub-second load bursts [2]

- UFO is not able to react quickly enough to the burst of sub-second.

Memcached: Bursts with increasing duration [2]

- The responsiveness of UFO depends on the number of steps to adjust.



(a) Input load respect to each application's *max load*

(e) Allocated core count for vCPU threads of each VM

(f) Tail latency normalized to QoS target under UFO

[1] Applied machine learning at Facebook (HPCA'18)

[2] Shenango (NSDI'19)

Dynamic Load: Colocation of 3 VMs, evaluate fluctuating load.

Nginx: Diurnal load fluctuations [1]

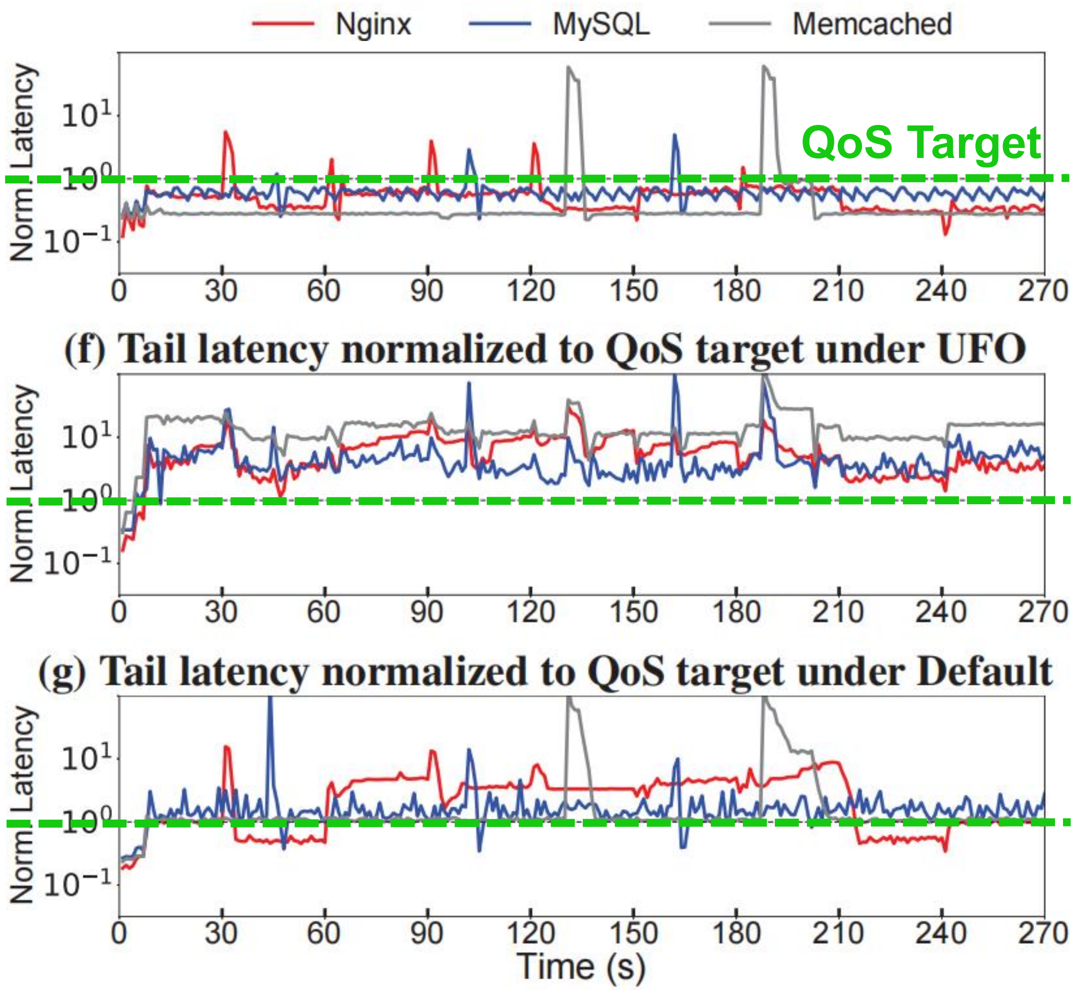
- UFO reacts to one second after any load change is detected, and performs better as more samples are collected.

MySQL: Sub-second load bursts [2]

- UFO is not able to react quickly enough to the burst of sub-second.

Memcached: Bursts with increasing duration [2]

- The responsiveness of UFO depends on the number of steps to adjust.



(f) Tail latency normalized to QoS target under UFO

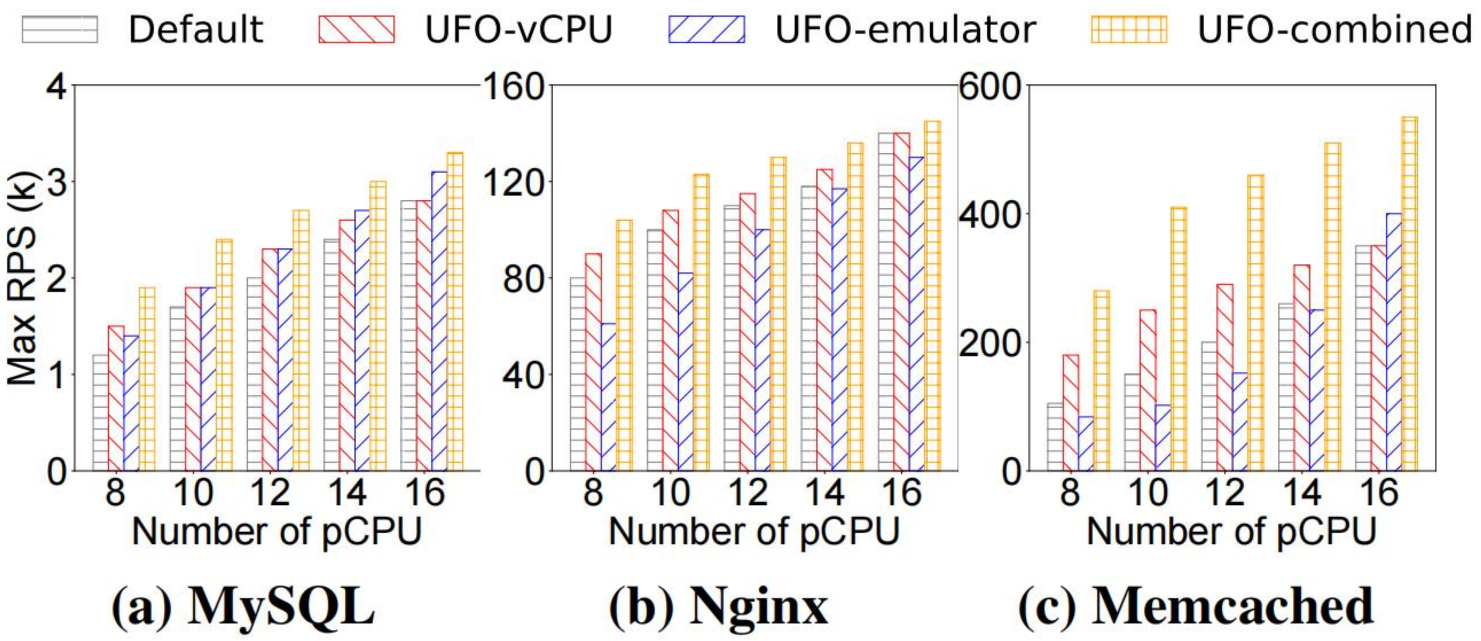
(g) Tail latency normalized to QoS target under Default

(h) Tail latency normalized to QoS target under DynIso

[1] Applied machine learning at Facebook (HPCA'18)

[2] Shenango (NSDI'19)

Decomposition of UFO



UFO-vCPU achieves 19.8% higher load on average under QoS than Default.

UFO-combined: 69.8% higher load

UFO-emulator: cause vcpu staking

Conclusion

UFO: The Ultimate QoS-Aware CPU Core Management for Virtualized and Oversubscribed Public Clouds

- Three levels CPU coordination
  - Host OS & Guest OS
  - Inner VM: vCPU threads & emulator threads
  - Host scheduler & Guest Applications
- Dynamic management based on QoS
- Higher resource efficiency

  Save up to 50% (average of 22%) cores under the same colocation scenario

Thank you!

Q&A

*Yajuan Peng, *Shuang Chen, Yi Zhao, Zhibin Yu

中国科学院深圳先进技术研究院
SHENZHEN INSTITUTES OF ADVANCED TECHNOLOGY
CHINESE ACADEMY OF SCIENCES

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Appendix 1: Impact on VM Exits and Caches under Host-Aware Iso

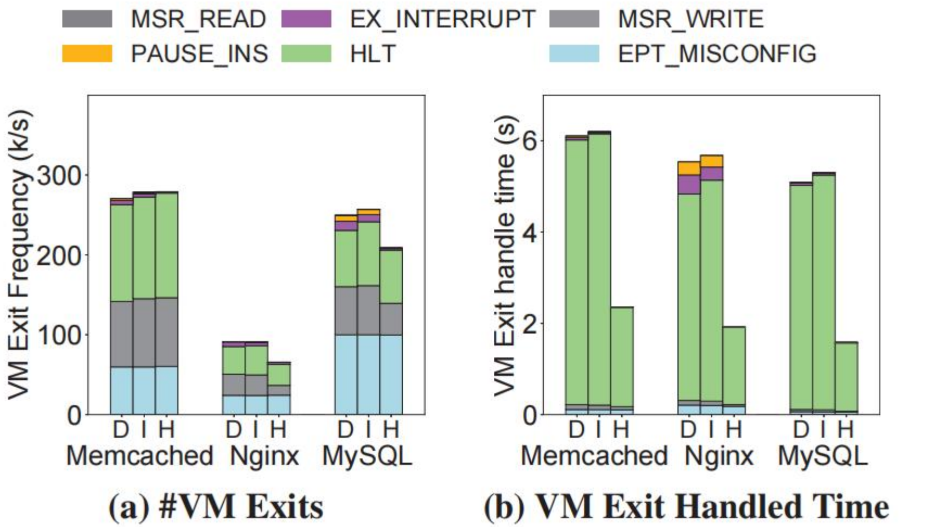


Figure 16: VM exit frequency and VM exit handled time under default (D), isolation (I), and host-aware isolation (H), decomposed by VM exit reason.

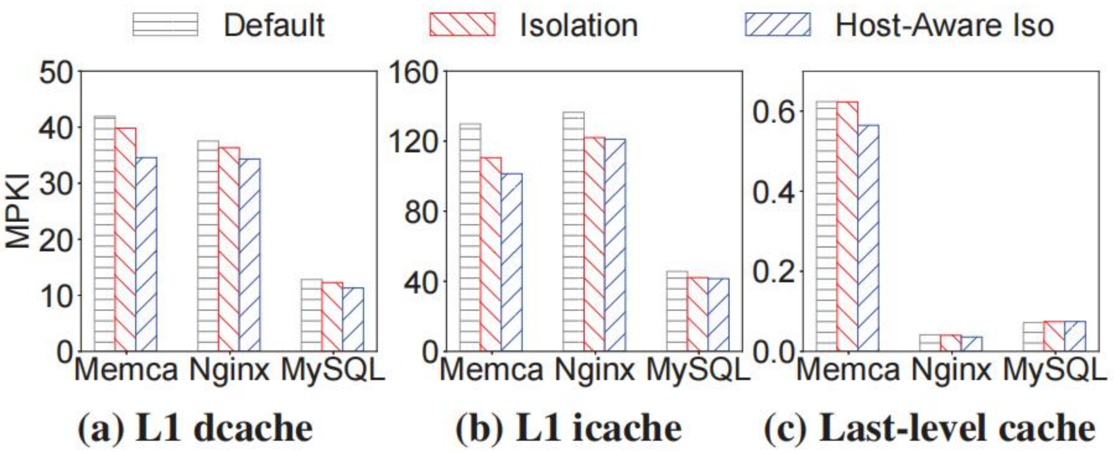


Figure 17: Cache misses-per-kilo-instructions (MPKI) under three core managers.

- VM exits are handled 2x faster on the host under host-aware isolation.
- Compared with Default, Isolation reduces L1D and L1I MPKI by up to 5% and 15% (average of 4.1% and 11%), respectively.

Appendix 2: Comparison with related work

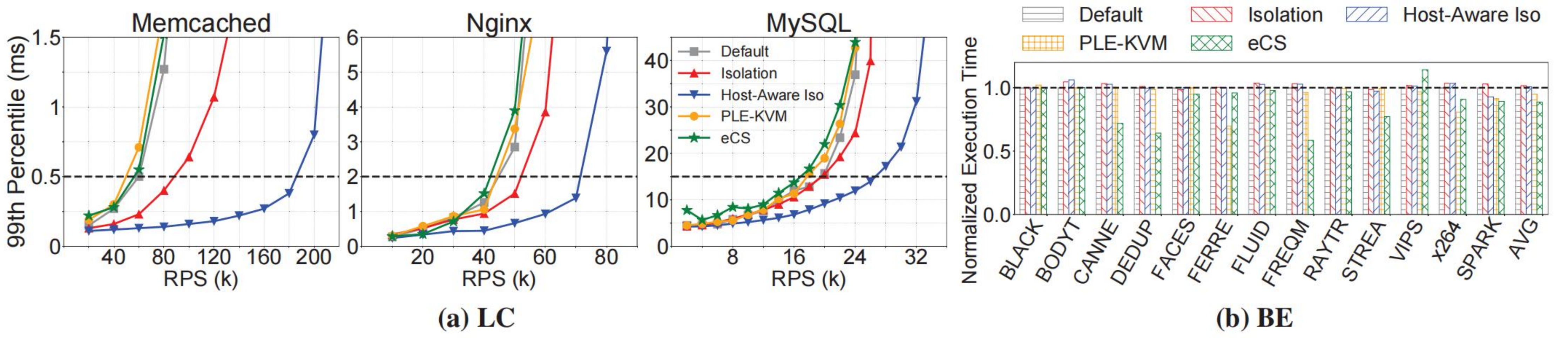


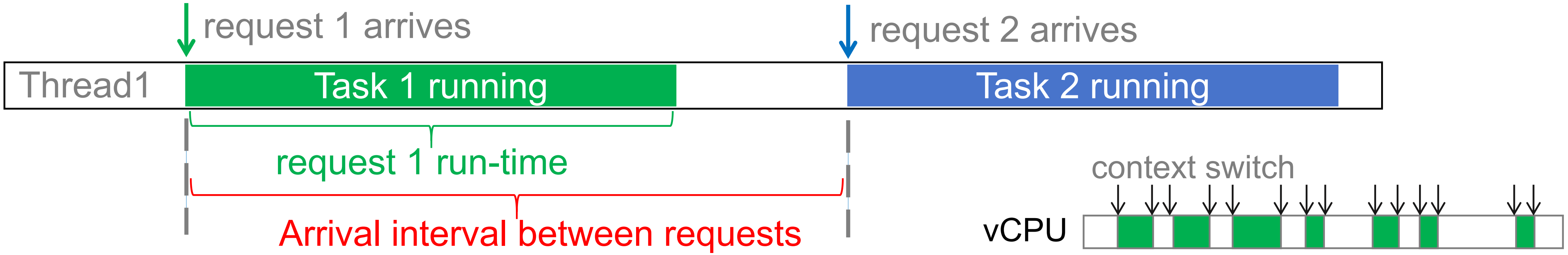
Figure 2: Performance under five core allocation mechanisms. For LC applications, we show the 99^{th} percentile tail latency with increasing input load (RPS). Horizontal dotted lines represent applications' QoS targets. For BE applications, we show the execution time of each benchmark normalized to that under the *Default* manager. Lower is better.

[1] PLE-KVM: Mitigating excessive vcpu spinning in vm-agnostic kvm. (VEE'21)

[2] eCS: Scaling guest {OS} critical sections with ecs. (USENIX ATC'18)

Appendix 3: High input load cause scheduling frequency decrease

Low input load: request inter-arrival time > request processing time



High input load: request inter-arrival time ≤ request processing time