

Eunomia: Preemption-based and QoS-aware Core Allocation in Oversubscribed Cloud

Yunda Guo^{1†}, Puqing Wu¹, Haoqiong Bian^{1*}, Yunpeng Chai^{1*},
Yao Shen², Haoyu Yang², Qing Liu², Zhengbin Huang², Le Yue², Yi Yang²

¹School of Information, Renmin University of China, Beijing, China

²HCS Lab, Huawei Cloud, China

{guoyunda,bianhq,ypchai}@ruc.edu.cn

Abstract—Colocating high- and low-priority VMs under CPU oversubscription is an effective way to improve resource utilization, but it demands careful core allocation to control contention and ensure the QoS. Existing solutions typically rely on Linux cgroup mechanisms such as *cpuset*, *quota*, and *share*. However, our experiments show that these mechanisms have inherent limitations and cannot simultaneously ensure QoS and resource efficiency. Unconditional preemption introduces new opportunities, which is a new kernel feature supported by major cloud vendors. Our experimental analysis reveals that, while unconditional preemption provides strong performance guarantees for high-priority VMs, it also increases the risk of starving low-priority VMs.

We present Eunomia, a CPU core allocator for oversubscribed clouds. Eunomia employs a black-box QoS degradation detection model that leverages transmit packet counts and kernel-level KVM tracepoints to identify performance degradation in high-priority VMs. Guided by this model, Eunomia selectively enables unconditional preemption only for degraded high-priority VMs, ensuring QoS while improving CPU efficiency. Experiments show that Eunomia delivers high-priority performance comparable to isolated execution while improving low-priority throughput by 50–64% over the best-performing *cpuset*-based baseline.

I. INTRODUCTION

Enterprises have long relied on self-managed infrastructures, which lack elasticity and often suffer from low utilization [1]–[3]. Cloud computing offers scalability and cost efficiency, yet regulatory, privacy, and security constraints prevent many organizations from fully adopting public clouds [4], [5]. To address this need, vendors provide on-premises cloud solutions, such as Huawei Cloud Stack [6], Alibaba Apsara Stack [7], and AWS Outposts [8]. These platforms commonly colocate high-priority latency-critical (LC) VMs with low-priority compute-intensive VMs and employ CPU oversubscription (i.e., one physical core is multiplexed across multiple vCPUs) to improve utilization [9]–[14]. However, oversubscription introduces contention that complicates QoS guarantees for high-priority VMs.

Existing solutions to CPU contention rely on Linux cgroup mechanisms such as *cpuset*, *quota*, and *share* [13], [15]–[17]. Our analysis in oversubscribed settings shows that none of them can simultaneously guarantee the QoS of high-priority VMs and maximize the throughput of low-priority VMs. *Cpuset* enforces core-level isolation by dedicating physical cores to high-priority VMs, which excludes low-priority VMs from those cores and thus eliminates the benefits of oversubscription. *Quota* restricts

low-priority VMs at a finer granularity of CPU time slices, reserving capacity for high-priority VMs to reduce contention. However, static quota configurations fail to accommodate the dynamic workload variations, leading to either wasted resources or QoS degradation. Even with dynamic tuning, finding a configuration that both protects QoS and maximizes throughput is inherently difficult. *Share* allocates CPU resources according to preset weights only during contention and appears promising, but our experiments show that its performance guarantees and actual resource allocations are unreliable.

Recently, cloud vendors have extended cgroup capabilities to support *unconditional preemption (UP)*. With UP, high-priority VMs can preempt low-priority ones as soon as they become runnable, without being constrained by the minimum time slice of the Completely Fair Scheduler (CFS) [15]. Representative implementations include TencentOS Server [18], Alibaba Cloud Linux [19], and Huawei Cloud EulerOS [20]. Our experiments show that UP is promising in providing strong performance guarantees for high-priority VMs under oversubscription. However, when the number of preemptible high-priority vCPUs increases, low-priority VMs risk starvation, losing access to otherwise idle resources and thereby reducing overall efficiency. Therefore, UP is not suitable for statically configuring all high-priority VMs in oversubscribed environments.

In this paper, we introduce Eunomia, a core allocator designed for oversubscribed CPU environments. Eunomia integrates a lightweight isolation-forest-based QoS degradation detection model that leverages VM transmit packet counts (tx.pkts) and kernel-level KVM tracepoints. Unlike prior approaches that rely on hardware performance monitoring counters or resource metrics [21]–[25], these features more accurately capture performance degradation inside VMs. Guided by this model, Eunomia activates unconditional preemption only when a high-priority VM experiences QoS degradation and uses *cpuset* to isolate it from other high-priority VMs, ensuring preemption affects only low-priority VMs. To maximize efficiency, Eunomia imposes no restrictions on low-priority VMs, allowing them to exploit all available cores. Experimental results across diverse scenarios show that Eunomia achieves high-priority performance comparable to isolated execution while improving low-priority throughput by 50–64% over the strongest *cpuset*-based baseline derived from the state-of-the-art work UFO [13].

The main contributions of this paper are as follows:

[†]Work done during internship at Huawei Cloud.

*Haoqiong Bian and Yunpeng Chai are the corresponding authors.

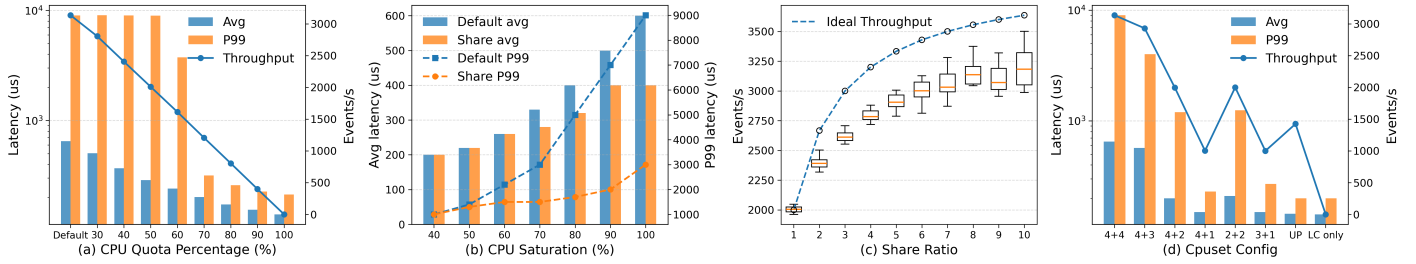


Fig. 1. Performance of MongoDB and Sysbench under different Core allocation mechanisms: (a) different quota percentage for MongoDB by limiting the CPU time available to Sysbench; (b) share with a 10:1 (high-priority:low-priority) ratio under different CPU saturation levels; (c) both high- and low-priority VMs running the same application (Sysbench) under different share ratios; (d) different cpuset configurations and unconditional preemption.

- We evaluate core allocation mechanisms under oversubscription in depth, highlighting their inability to balance high-priority VM QoS and low-priority throughput.
- We propose a novel VM QoS degradation detection model that leverages VM transmit packet counts and kernel-level KVM tracepoints, achieving high accuracy and recall.
- We design a preemption-based CPU core allocator that efficiently guarantees the QoS of high-priority VMs while maximizing the throughput of low-priority VMs.

II. BACKGROUND

We review current core allocation mechanisms supported by Linux `cgroup`, which is widely used in container and VM management [1], [13], [17]. The main mechanisms are `cpuset`, `quota`, and `share` [16]. `Cpuset` controls which cores a VM can run on, eliminating contention by granting exclusive access. `Quota` imposes a hard limit on the CPU time a VM can use within a fixed period (typically 100 ms); for example, a quota of 150 ms under the default period caps utilization at 150%. `Share` allocates CPU proportionally based on assigned weights during contention; for instance, with equal shares among three VMs, each receives one-third of the CPU when all are runnable.

A. Experimental Analysis of Core Allocation Mechanisms

We launch two VMs with a typical configuration of 4 vCPUs and 8 GB of memory [26], [27]. Both VMs are pinned to the same four physical cores, simulating an oversubscription ratio of 2 [12]. One VM runs MongoDB as a latency-critical high-priority workload, and the other runs Sysbench as a compute-intensive low-priority workload. The MongoDB load is generated by a four-thread YCSB [28] client on a separate machine. Sysbench also uses four threads, ensuring that every vCPU in both VMs is active. Figure 1 presents results for different core allocation mechanisms. Each experiment is repeated five times, and the figure reports average values.

Observation 1: Quota enforces strict CPU limits on low-priority VMs, reducing interference on high-priority VMs. Figure 1(a) shows that under the OS’s default scheduling, Sysbench dominates CPU resources, leaving MongoDB with less than 30% and causing a 5 \times increase in average latency and a 42 \times increase in P99 latency. Increasing the quota percentage for MongoDB reduces Sysbench throughput nearly linearly while gradually improving MongoDB’s performance. However,

in practice, selecting an optimal quota percentage that both mitigates interference and maximizes utilization is challenging, and static configurations cannot adapt to dynamic loads.

Observation 2: Share provides limited performance protection for high-priority VM. Figure 1(b) compares share-10:1 (high:low-priority ratio) with the default (1:1). When running alone, MongoDB consumes about 40% of CPU resources. As Sysbench load increases, overall utilization rises toward saturation. Below 60% saturation, share and default show little difference, indicating no contention. Beyond 70%, share begins to take effect, but at full saturation MongoDB’s average and P99 latencies still rise by 2 \times and 3 \times , respectively. This demonstrates that the performance protection offered by share is limited.

To investigate the performance protection limitations of share, we run the same application in both VMs to remove application-specific contention effects. We use Sysbench because its throughput strongly correlates with the CPU resources it receives (Figure 1(a)). Under different share ratios, we run Sysbench for 10 seconds across 10 trials and record the throughput of the high-share VM. Figure 1(c) shows the resulting box plots alongside theoretical values assuming strict proportional allocation.

Observation 3: Share does not enforce strict proportional allocation. Figure 1(c) shows that, except for the 1:1 case, actual CPU allocation is consistently below the theoretical expectation, with deviations increasing as the share ratio grows. This indicates that share cannot accurately and reliably guarantee resource allocation and performance.

B. Opportunity: Unconditional Preemption

Unconditional Preemption (UP) has been introduced as a kernel feature by major cloud vendors, including TencentOS Server [18], Alibaba Cloud Linux [19], and Huawei Cloud EulerOS [20]. Their implementations follow a similar design, adding a new scheduling class to the kernel that allows high-priority tasks to preempt low-priority tasks in the same way runnable tasks preempt the idle thread. Therefore, we use EulerOS in our experiments as a representative implementation.

Figure 1(d) compares UP with different `cpuset` configurations. Here, $A + B$ denotes the number of cores bound to the high- and low-priority VMs. When $A + B = 4$, there are no overlapping shared cores. Since the standalone high-priority VM requires about 160% CPU, we ensure $A \geq 2$. Under UP, both VMs share four cores, with the high-priority VM able to

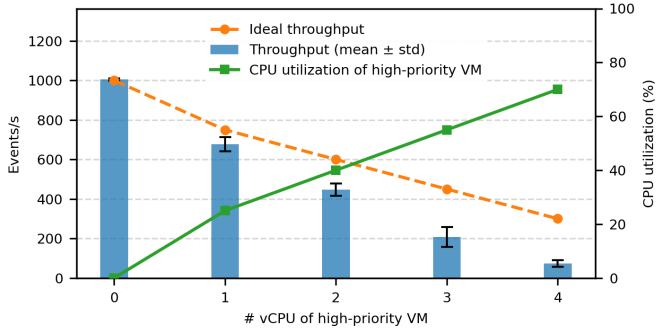


Fig. 2. Throughput of Sysbench when running with different numbers of high-priority vCPUs with unconditional preemption.

preempt the low-priority one. *LC only* serves as the ideal case, which runs only the latency-critical high-priority VM.

Observation 4: Unconditional Preemption provides strong performance guarantees for high-priority VMs. With UP, MongoDB achieves performance nearly identical to isolated execution (LC only), showing no interference from the CPU-intensive Sysbench. In contrast, coarse-grained *cpuset* imposes trade-offs: either increased MongoDB latency (e.g., “2+2”) or reduced Sysbench throughput (e.g., “3+1”). Compared with the best *cpuset* case for MongoDB (“4+1”), UP still reduces P99 latency by 13% while increasing Sysbench throughput by 43%.

One might assume that allowing all high-priority VMs to preempt low-priority VMs would suffice. However, we demonstrate the limitations of this static configuration through an experiment. Using *virsh*, we pin both high- and low-priority VMs to the same physical core. We gradually increase the number of MongoDB instances (each with one worker thread) to emulate multiple vCPUs of high-priority VMs, while the low-priority VM runs a single-threaded Sysbench workload. Figure 2 presents the results of ten runs, reporting Sysbench throughput under varying numbers of high-priority vCPUs

Observation 5: Unconditional Preemption increases starvation risk for low-priority VMs when high-priority vCPUs are numerous. Figure 2 shows that the gap between the actual and ideal throughput of Sysbench widens as the number of high-priority vCPUs increases. The ideal throughput represents the maximum Sysbench could achieve if it fully utilized the CPU cycles left by high-priority vCPUs. The widening gap indicates that Sysbench loses scheduling opportunities due to preemption. Moreover, we observe that as the number of high-priority vCPUs further increases (the aggregate CPU utilization exceeds 70%), the low-priority VM may receive no CPU time at all, leading to CPU stalls and kernel soft lockups.

III. DESIGN

Since unconditional preemption effectively mitigates interference, Eunomia uses it to guarantee high-priority VM QoS. However, to maximize low-priority VM throughput, it is crucial to carefully determine when to enable preemption and how to allocate cores.

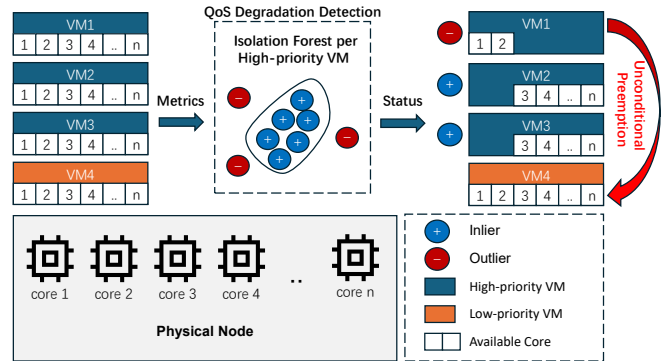


Fig. 3. Workflow of Eunomia.

A. Principles

Unconditional preemption is enabled only when QoS degradation occurs in high-priority VMs. Enabling unconditional preemption at all times negatively impacts the throughput of low-priority VMs (Observation 5). Moreover, when no QoS degradation occurs, preempting low-priority VMs is unnecessary and reduces resource efficiency.

The QoS degradation detection model must be timely, sensitive, and lightweight. Since most CPU cores on a physical node are allocated to customer VMs to maximize cost efficiency, the resources available for monitoring and management are limited. As a result, approaches such as deep learning [25], which rely on monitoring a large number of metrics and excessive compute resources, are not feasible.

No explicit CPU restrictions are imposed on low-priority VMs. To maximize CPU utilization, low-priority VMs always have access to all cores. Except for necessary preemption to protect high-priority QoS, Eunomia applies no CPU constraints (e.g., *cpuset*, quota, or share) to low-priority VMs.

B. Workflow of Eunomia

Eunomia focuses on intra-node core management. Figure 3 illustrates its workflow. For each high-priority VM, Eunomia trains an isolation forest model to detect QoS degradation (see Section III-C for details). During normal operation without degradation, high- and low-priority VMs share all available cores on the node. Meanwhile, Eunomia continuously monitors the load and KVM metrics (Table I) of each high-priority VM and performs outlier detection.

When QoS degradation is detected, as in the case of VM1 in Figure 3, Eunomia grants it unconditional preemption. To prevent VM1 from degrading the QoS of other high-priority VMs, Eunomia uses *cpuset* to isolate VM1. The number of physical cores bound to VM1 equals its vCPU count, ensuring that only one high-priority vCPU thread runs on each assigned core. This design has two benefits: it eliminates the double scheduling [13] and reduces the risk of starving low-priority VMs on those cores (Observation 5). We assume the external scheduler ensures the total number of vCPUs requested by high-priority VMs does not exceed the available physical cores, guaranteeing that each high-priority VM can exclusively occupy the cores it requests, even in extreme cases.

TABLE I
KVM METRICS USED IN VM QoS DETECTION

Metric	Description
kvm_entry	Number of times the virtual CPU switches from the host to the guest.
kvm_apic_accept_irq	Number of times the virtual machine accepts interrupt requests (IRQ).
kvm_vcpu_wakeup	Number of times the vCPU in the virtual machine is woken up.
kvm_halt_poll_ns	Duration of polling when the virtual machine is in an idle state (in nanoseconds).

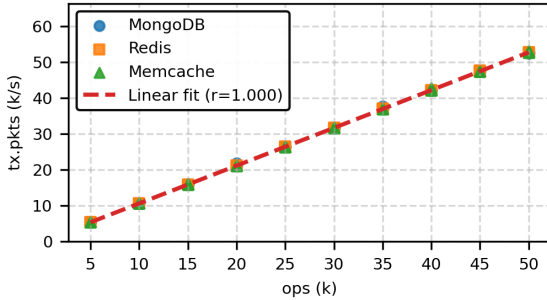


Fig. 4. Correlation between ops and tx.pkts with Linear Regression.

C. QoS Degradation Detection

Applications inside a VM are often opaque, making application-level performance metrics inaccessible. Unlike prior work that relies on low-level hardware performance monitoring counters (PMCs) or resource metrics [21]–[25], we employ kernel-level KVM tracepoints, which are closer to the application and more directly capture the VM’s internal state, including running, idling, and wake-up behavior.

KVM metrics. Using `perf`, we collected all KVM-related metrics across diverse workloads in Table II. After discarding constant metrics and applying correlation-based filtering, we identified four metrics most indicative of application performance. Table I summarizes their interpretations.

VM transmit packets as load indicator. Since the response of LC applications to user requests involves transmitting at least one network packet, we use the number of transmitted packets per second (`tx.pkts`) to represent the load state of high-priority VMs. As shown in Figure 4, we sampled transmit packets five times for three applications under varying operations per second (`ops`) and plotted both scatter and linear fit results. The results demonstrate that `tx.pkts` exhibits a nearly perfect linear correlation with the actual load.

Isolation forest model. Although application performance inside a VM is not directly observable, we can construct a core-interference-free environment using `cpuset`. The KVM metrics and `tx.pkts` collected in a core-isolated environment represent non-degraded performance. For each high-priority VM, we train an isolation forest model as the degradation detection classifier because of its suitability for one-class data (non-degraded only), lightweight nature, and independence from GPU [29].

Training data collection. To enable rapid response to QoS degradation, we set the monitoring granularity to one second. However, continuous metric collection may generate large

volumes of redundant or irrelevant data, thereby increasing storage and training overhead. To mitigate this, we enable core isolation and collect training data only when transmit packet activity is detected. Furthermore, for traffic scenarios of similar magnitude, we record only one minute of data. Since cloud applications typically exhibit diurnal load patterns [30], [31], we collect data over a full day for model training. After training, we remove the core isolation configuration and continuously monitor the metrics online to detect potential degradation.

Model update. When online metrics deviate significantly from those observed during training, we trigger model retraining. For instance, a noticeable shift in the distribution of the VM’s CPU utilization or transmitted packets may indicate a substantial change in the VM’s workload.

IV. EVALUATION

Table II presents the platform configuration and workloads used in our experiments. Since most VM specifications are much smaller than the core count of a single NUMA node and require NUMA affinity [12], [27], [32], we conduct VM colocation experiments within a single NUMA node. As high-priority VMs, we use Redis, Memcached, and MongoDB, which represent common cloud workloads and are widely used in prior literature [14], [26], [33]. Their workloads are generated using YCSB [28] with one million records under a read-only query pattern. YCSB runs on a separate node. For low-priority VMs, we employ Sysbench [34], as its throughput is strongly correlated with resource allocation (see Figure 1(a)), making it a good proxy for resource utilization.

A. Baselines

Default. VMs are scheduled according to OS’s default scheduling policy, with all VMs sharing the available cores.

Share. We evaluate three configurations of the high-to-low priority VM share ratio: 2:1, 100:1, and max (1048576:2).

Quota. We limit the maximum CPU resources available to low-priority VMs using quota mechanism. Ratios of 1:1, 3:1, and 9:1 correspond to low-priority VMs being restricted to 50%, 25%, and 10% of total CPU resources, respectively.

Cpuset. We evaluate two Cpuset modes. In Cpuset-partial, each high-priority VM exclusively occupies a number of physical cores equal to half of its vCPU count, while sharing the remaining cores with other VMs [35]. In Cpuset-full, each high-priority VM exclusively occupies a number of physical cores equal to its vCPU count, with no sharing across VMs [13].

LC only. Only latency-critical (LC) high-priority VMs are deployed, serving as an upper-bound performance baseline.

B. Diverse Resource Saturation Scenarios

After reserving cores for VM management, 40 cores remain available on the test node. We launch a 32-vCPU high-priority VM running MongoDB with a 32-thread YCSB workload. Additionally, we start a 40-vCPU low-priority VM running Sysbench (40 threads). This setup simulates an oversubscription ratio of $(32 + 40)/40 = 1.8$.

When running alone, MongoDB consumes approximately 25% of total CPU resources. We gradually increase Sysbench

TABLE II
PLATFORM SPECIFICATION AND WORKLOADS

Platform Specification	
Hardware	CPU Intel(R) Xeon(R) Gold 5220R, 2 sockets, 24 cores per socket 2 threads per core, 692GB memory, 2 NUMA nodes
Software	OS: Huawei Cloud EulerOS 2.0 (x86_64) with kernel 5.10.0
Workloads	
Redis [36]	An in-memory data store (single-threaded), high priority
Memcached [37]	An in-memory key-value store (multi-threaded), high priority
MongoDB [38]	A document-oriented NoSQL database, high priority
Sysbench [34]	A compute-intensive cpu test, low priority

load to saturate the node’s CPU, evaluating scenarios with 40%, 60%, 80%, and 100% saturation levels. Figure 5 presents the results. For each scenario, we collect stable data over a 10-second interval. P99 latency is reported as the mean and standard deviation of ten 1-second samples, while Sysbench throughput is the average over the 10-second period.

In *default*, MongoDB’s P99 latency rises sharply as node saturation increases. *Share-2:1* provides little improvement over default and offers only minor stability at 100% saturation. As the share ratio increases to 100:1, the effect becomes clearer and P99 latency drops noticeably (note the exponential scale of the latency axis). Averaged across the four saturation levels, *share-100:1* reduces P99 latency by 38% relative to default. *Share-max* shows no meaningful benefit over 100:1. Notably, under the share policy, Sysbench throughput remains close to default, even at full saturation. This indicates that Sysbench can still consume any idle CPU cycles left unused by MongoDB.

By contrast, *quota* strictly caps the CPU available to Sysbench. From 1:1 to 3:1 to 9:1, Sysbench receives a smaller CPU fraction and its throughput decreases roughly linearly. MongoDB’s P99 latency improves as more CPU is reserved for it. Due to its strict resource constraints, quota provides significantly more stable performance than share, particularly under high saturation. However, even with *quota-9:1*, Sysbench still increases MongoDB’s P99 latency by about 60% compared with running alone.

Cpuset-partial improves MongoDB latency over default by granting it exclusive cores, but Sysbench throughput declines by 33% at 100% saturation due to reduced core availability. *Cpuset-full* eliminates core interference and achieves near-isolated MongoDB performance, but further reduces Sysbench throughput by 64%. *Eunomia* also attains near-isolated MongoDB performance through unconditional preemption. However, unlike cpuset, it imposes no core restrictions on Sysbench, resulting in a substantial throughput gain. Compared with cpuset-full, Sysbench throughput improves by 53% on average.

C. Multiple High-Priority VMs Colocation Scenario

We evaluate a colocation scenario with three high-priority VMs: a 16-vCPU VM running MongoDB, an 8-vCPU VM running Memcached, and a 4-vCPU VM running Redis. The low-priority VM is a 40-vCPU instance running Sysbench (no load limit). This setup simulates an oversubscription ratio of $(16 + 8 + 4 + 40)/40 = 1.7$. For each high-priority VM, the number of YCSB threads equals its vCPU count. Collectively, the high-priority workloads consume about one-third of the

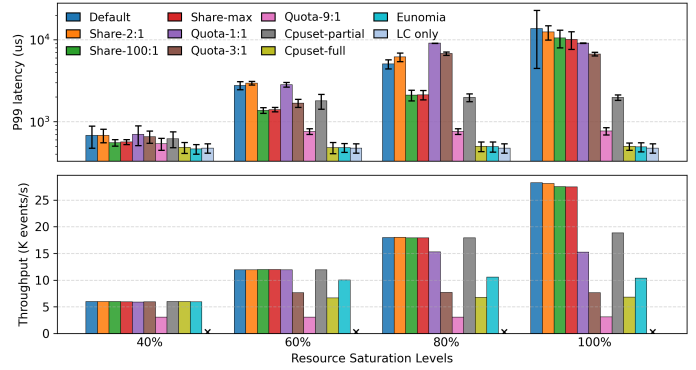


Fig. 5. The comparison of MongoDB P99 latency and Sysbench throughput for different approaches with various CPU saturation levels.

total CPU resources. Figure 6 presents the performance of all four applications under different resource allocation methods.

Share fails to guarantee the performance of the three high-priority applications, as Sysbench can utilize idle node resources, keeping the system in a highly saturated state. *Quota* improves performance by limiting low-priority VMs and thereby creating a more resource-abundant environment for high-priority VMs. However, under the *quota-9:1* case, the P99 latency of the three high-priority applications still increases by an average of 47% compared with the *LC only* scenario, indicating interference from low-priority vCPUs.

Cpuset-partial alleviates interference by granting high-priority VMs exclusive access to part of the cores. Yet, contention on the shared cores becomes more severe, as the low-priority VM can only run there. Consequently, compared with *cpuset-full*, *cpuset-partial* causes performance fluctuations (Redis and Memcached) or degradation (MongoDB). *Cpuset-full* achieves performance comparable to *LC only* but restricts Sysbench throughput due to limited cores. Leveraging unconditional preemption, *Eunomia* preserves high-priority VM performance while improving Sysbench throughput by 64% compared with *cpuset-full*.

D. QoS Degradation Detection Model

We evaluate Eunomia’s QoS degradation detection model using three LC applications under stepwise increasing loads [13]. The VM is configured with 4 vCPUs, a commonly used specification [26], [27], [32]. We compare Eunomia with several representative approaches from both industry and academia.

ST% (Huawei Cloud) [21]. This method computes the effective CPU utilization as $\text{cpu_util\%}/(\text{cpu_util\%} + \text{steal\%})$, where cpu_util\% is the CPU time actually consumed by the VM and steal\% is the fraction of time the VM waits for CPU allocation. Values below the threshold (minimum in the cpuset phase), indicating excessive steal time, are classified as QoS degradation.

CPI2 (Google) [22]. It uses CPI (cycles per instruction) to detect performance anomalies. During the cpuset phase, we collect the distribution of CPI values and set the threshold to the mean plus two standard deviations as the paper described. CPI values exceeding this threshold indicate QoS degradation.

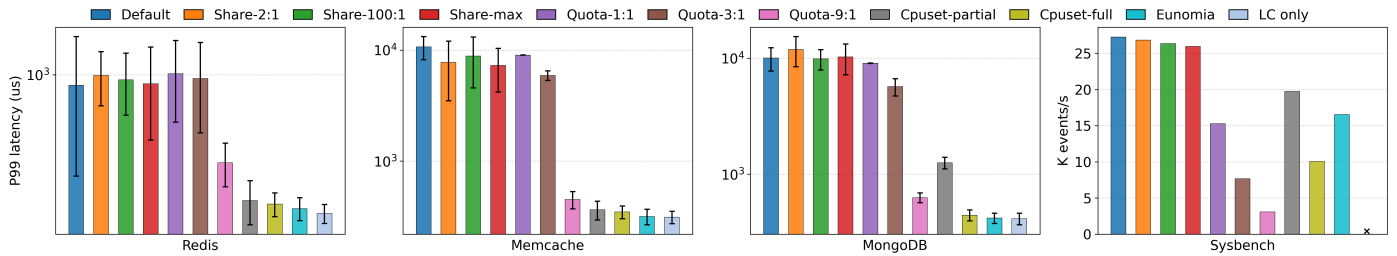


Fig. 6. Performance of multiple colocated applications under different resource allocation methods.

TABLE III
COMPARISON OF QoS DEGRADATION DETECTION METHODS

Method	Memcached		Redis		MongoDB		Avg	
	Accuracy	Recall	Accuracy	Recall	Accuracy	Recall	Accuracy	Recall
ST% [21]	74%	68%	83%	79%	83%	78%	80%	75%
CPI2 [22]	73%	68%	70%	45%	42%	26%	62%	46%
Holmes [23]	66%	60%	73%	50%	44%	28%	61%	46%
CloudWhite [24]	75%	88%	57%	92%	78%	95%	70%	92%
Eunomia	93%	100%	81%	95%	90%	100%	88%	98%

Holmes [23]. It uses $\text{stalls_mem_any}/(\text{Num_load} + \text{Num_store})$, where stalls_mem_any is the number of cycles stalled waiting for memory subsystem responses. Values above the threshold (maximum in the cpuset phase) are classified as QoS degradation.

CloudWhite [24]. This method monitors both GIPS (Giga-Instructions Per Second) and CPU utilization. If the two metrics increase proportionally, the VM is considered normal; otherwise, degradation is inferred. When one VM is identified as normal under increasing load, all other VMs are regarded as potential QoS-degraded instances.

We use *perf*, *virsh*, and *vmtop* to collect the required metrics at 1-second granularity. All methods use data (300–400 samples) from the same scenarios either to train their models or to compute thresholds. In the test phase, we colocate Sysbench (4-vCPUs) on four physical cores, simulating an oversubscription ratio of 2. By varying the Sysbench load, we construct multiple levels of CPU saturation. Each test set contains about 300 entries with a degradation-to-nondegradation ratio of approximately 7:3. We record the application’s P99 latency during cpuset as the baseline; in testing, P99 values above the baseline are labeled degraded, otherwise non-degraded.

Table III reports the results across the three application test sets. We focus on two metrics: (1) classification accuracy, and (2) recall for degraded cases. *Eunomia* consistently achieves the highest recall, reaching 100% for both Memcached and MongoDB. In terms of accuracy, *Eunomia* slightly trails *ST%* in the Redis case but ranks first in the other two. *CloudWhite* achieving high recall is largely due to over-detection: flagging other VMs as degraded when Sysbench’s GIPS and CPU utilization increase proportionally. Additionally, its parameter requires fine-tuning for different VMs. Among the remaining methods, the *ST%* proves more effective than *CPI2* or *Holmes* at capturing application performance under CPU contention.

E. Compare with Static Unconditional Preemption

We compare *Eunomia* with *static unconditional preemption (UP)* to highlight the benefits of QoS degradation detection

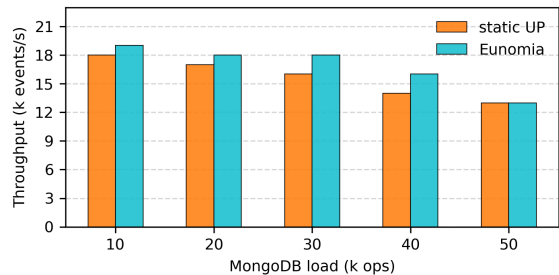


Fig. 7. Sysbench throughput under different MongoDB load levels.

model. We launch a 40-vCPU high-priority VM running MongoDB and a 20-vCPU low-priority VM running Sysbench. Figure 7 shows that, before 50k ops, *Eunomia* correctly detects no QoS degradation, allowing MongoDB to share resources with Sysbench. As the load increases, Sysbench faces more frequent preemption. From 10k to 40k ops, *Eunomia* improves Sysbench throughput by 1.8%, 7.6%, 9.1%, and 11.7% compared to *static UP*. Once the load exceeds 50k ops (around 40% CPU saturation), *Eunomia* correctly identifies QoS degradation and enables unconditional preemption, achieving performance equivalent to *static UP*. *Eunomia*’s advantage lies primarily in low-load periods, during which high-priority VMs do not suffer QoS degradation. Since low-load periods are typically much longer [13], [30], and considering the scale of cloud clusters, the overall improvement in resource utilization is substantial.

V. CONCLUSION

We present *Eunomia*, a preemption-based CPU core allocator for oversubscribed clouds. Our experiments show that unconditional preemption offers stronger QoS guarantees for high-priority VMs than traditional cgroup mechanisms. *Eunomia* augments this capability with a lightweight QoS degradation detection model that activates preemption only when necessary, thereby maximizing resource efficiency. By leveraging kernel-level metrics, it captures VM performance more accurately than prior hardware-based approaches. We expect these insights to inform future research and bring *Eunomia* closer to deployment in real cloud environments.

ACKNOWLEDGMENT

This work is supported by the National Key Research and Development Program of China (No. 2023YFB4503600), National Natural Science Foundation of China (No. U23A20299).

REFERENCES

- [1] C. Delimitrou and C. Kozyrakis, “Quasar: Resource-efficient and qos-aware cluster management,” *ACM Sigplan Notices*, vol. 49, no. 4, pp. 127–144, 2014.
- [2] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, “Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces,” in *Proceedings of the international symposium on quality of service*, 2019, pp. 1–10.
- [3] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, “Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 153–167.
- [4] N. Phaphoom, X. Wang, S. Samuel, S. Helmer, and P. Abrahamsson, “A survey study on major technical barriers affecting the decision to adopt cloud services,” *Journal of Systems and Software*, vol. 103, pp. 167–181, 2015.
- [5] N. Soveizi, F. Turkmen, and D. Karastoyanova, “Security and privacy concerns in cloud-based scientific and business workflows: A systematic review,” *Future Generation Computer Systems*, vol. 148, pp. 184–200, 2023.
- [6] “Huawei cloud stack,” <https://www.huaweicloud.com/intl/en-us/product/huaweicloudstack.html>.
- [7] “Alibaba apasara stack,” <https://www.alibabacloud.com/en/product/apasarastack>.
- [8] “Aws outposts family,” <https://aws.amazon.com/outposts>.
- [9] “Configure the cpu overcommit ratio,” <https://www.alibabacloud.com/help/en/dedicated-host/user-guide/configure-the-cpu-overcommit-ratio>.
- [10] “Configuring the cpu overcommitment ratio,” https://support.huawei.com/carrier/docview?nid=DOC1101452143&topicId=aba45f2a#EN-US_TOPIC_0248263254.
- [11] “Byol and oversubscription,” <https://aws.amazon.com/blogs/compute/byol-and-oversubscription>.
- [12] “Guidelines for overcommitting vmware resources,” https://www.heroix.com/download/Guidelines_for_Overcommitting_VMware_Resources.pdf.
- [13] Y. Peng, S. Chen, Y. Zhao, and Z. Yu, “{UFO}: The ultimate {QoS-Aware} core management for virtualized and oversubscribed public clouds,” in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1511–1530.
- [14] B. Reidys, P. Zardoshti, Í. Goiri, C. Irvine, D. S. Berger, H. Ma, K. Arya, E. Cortez, T. Stark, E. Bak *et al.*, “Coach: Exploiting temporal patterns for all-resource oversubscription in cloud platforms,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 164–181.
- [15] L. Liu, H. Wang, A. Wang, M. Xiao, Y. Cheng, and S. Chen, “Mind the gap: Broken promises of cpu reservations in containerized multi-tenant clouds,” in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 243–257.
- [16] P. Turner, B. B. Rao, and N. Rao, “Cpu bandwidth control for cfs,” in *Linux Symposium*, vol. 10. Citeseer, 2010, pp. 245–254.
- [17] C. Carrión, “Kubernetes scheduling: Taxonomy, ongoing issues and challenges,” *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.
- [18] “Tencentos server introduction,” <https://www.tencentcloud.com/document/product/213/40223>.
- [19] “Alibaba cloud linux group identity feature,” <https://www.alibabacloud.com/help/en/alinux/user-guide/group-identity-feature>.
- [20] “Huawei cloud eulers multi-level hybrid scheduling of kernel cpu cgroups,” https://doc.hcs.huawei.com/en-us/usermanual/hce/hce_02_0074.html.
- [21] “Hotspot elimination,” https://support.huawei.com/carrier/docview?nid=DOC1101452143&topicId=4b6b88e3#EN-US_TOPIC_000001987107901.
- [22] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, “Cpi2: Cpu performance isolation for shared compute clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 379–391.
- [23] A. Pi, X. Zhou, and C. Xu, “Holmes: Smt interference diagnosis and cpu scheduling for job co-location,” in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 110–121.
- [24] L. Pons, J. Feliu, J. Sahuquillo, M. E. Gómez, S. Petit, J. Pons, and C. Huang, “Cloud white: Detecting and estimating qos degradation of latency-critical workloads in the public cloud,” *Future Generation Computer Systems*, vol. 138, pp. 13–25, 2023.
- [25] Y. Cheng, X. Huang, Z. Liu, J. Chen, X. Gao, Z. Fang, and Y. Yang, “Fedge: An interference-aware qos prediction framework for black-box scenario in iaas clouds with domain generalization,” in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 128–138.
- [26] T. Shi, Y. Yang, Y. Cheng, X. Gao, Z. Fang, and Y. Yang, “Alioth: A machine learning based interference-aware performance monitor for multi-tenancy applications in public cloud,” *arXiv preprint arXiv:2307.08949*, 2023.
- [27] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich *et al.*, “Protean:{VM} allocation service at scale,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 845–861.
- [28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.
- [29] Y. Zhong, D. S. Berger, C. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury *et al.*, “Managing memory tiers with {CXL} in virtualized environments,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 37–56.
- [30] Y. Guo, J. Ge, P. Guo, Y. Chai, T. Li, M. Shi, Y. Tu, and J. Ouyang, “Pass: Predictive auto-scaling system for large-scale enterprise web applications,” in *Proceedings of the ACM Web Conference 2024*, 2024, pp. 2747–2758.
- [31] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *2018 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2018, pp. 620–629.
- [32] K. Wang, Y. Li, C. Wang, T. Jia, K. Chow, Y. Wen, Y. Dou, G. Xu, C. Hou, J. Yao *et al.*, “Characterizing job microarchitectural profiles at scale: Dataset and analysis,” in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.
- [33] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, “Pond: Cxl-based memory pooling systems for cloud platforms,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.
- [34] “Scriptable database and system performance benchmark,” <https://github.com/akopytov/sysbench>.
- [35] Y. Liu, X. Deng, J. Zhou, M. Chen, and Y. Bao, “Ah-q: Quantifying and handling the interference within a datacenter from a system perspective,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 471–484.
- [36] “Redis,” <https://redis.io/>.
- [37] “Memcached,” <https://memcached.org/>.
- [38] “Mongodb,” <https://www.mongodb.com/>.