Future Generation Computer Systems 179 (2026) 108355

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Interference modeling and scheduling for compute-intensive batch applications

Chennian Xiong [a], Weiwei Lin [a,b,*], Huikang Huang [a], Jianpeng Lin [a], Keqin Li [c]

[a] Department of Computer Science and Engineering, South China University of Technology, GuangZhou, 510000, China
[b] Pengcheng Laboratory, Shenzhen, 518066, China
[c] Department of Computer Science, State University of New York, New York, 12561, New Paltz, US

ARTICLE INFO

Keywords:
Compute-intensive batch application
Metrics analysis
Temporal data
Performance interference modeling
Interference-aware scheduling.

ABSTRACT

Cloud computing and virtualization technologies have significantly improved resource utilization in data centers. However, performance interference caused by resource contention remains a major challenge, particularly for compute-intensive batch applications, which are vital for large-scale data processing and task scheduling. Addressing performance interference in the modeling and scheduling of such applications still requires improvement. Existing interference models often rely on stereotypical metrics and average values, ignoring the impact of temporal fluctuations, while conventional scheduling algorithms overlook interference dynamics, leading to suboptimal scheduling results. To overcome these limitations, this article investigates the key factors influencing the performance of compute-intensive workloads and introduces a novel performance interference model that incorporates temporal fluctuations. Furthermore, we propose a historical-data-driven scheduling method that accounts for both temporal dynamics and batch application interference characteristics. Experimental results demonstrate that the proposed performance interference model achieves higher accuracy and robustness against overfitting compared to existing models that neglect temporal variations. Additionally, our interference-aware scheduling algorithm significantly outperforms traditional methods in throughput, scheduling efficiency, and server load balancing, providing an effective solution to mitigate performance interference in cloud environments.

1. Background

With the rapid development of cloud computing, its scalability, reliability, and cost-effectiveness have made multi-virtual machine (VM) co-location deployment a common practice for enhancing resource utilization. As of 2025, approximately 94% of enterprises have migrated their workloads to the cloud [1]. However, cloud VMs require access to the host physical machine's resources to execute workloads, resulting in incomplete performance isolation. The performance of VMs depends heavily on the efficient allocation of shared physical resources; yet, resource contention frequently leads to performance degradation, a phenomenon termed performance interference [2]. Performance interference between VMs can significantly diminish application throughput and quality of service. Research by Maji et al. demonstrates that variations in interference intensity can increase the runtime of certain benchmarks, with network-intensive VMs experiencing up to a twofold increase, disk access rising by 4.5 times, and cache access escalating by 5.5 times [3]. Batch applications play a pivotal role in cloud data center scheduling. By leveraging the parallel processing capabilities of multiple physical servers and VMs, the execution efficiency of batch tasks can be improved [4]. These tasks can be executed concurrently across different servers and VMs [5,6]. Nevertheless, when multiple applications compete for the same shared resources, significant performance interference can occur [7]. The primary function of batch programs is to execute computational tasks, hence they typically impose significant demands on CPU resources. However, such programs are not necessarily confined to being CPU-intensive; some also exhibit specific requirements for memory resources. Furthermore, even for batch programs that are purely CPU-consuming, it remains uncertain whether additional resource consumption may arise from system-level factors when they are co-located with other workloads. We refer to such virtual machines hosting these workloads as compute-intensive VMs. The term "compute" is used here to emphasize their core functional attribute [8]. Undoubtedly, addressing performance interference in compute-intensive VMs is essential for enhancing the overall efficiency of cloud systems [9].

Accurate evaluation of performance interference is a prerequisite for effectively tackling this issue. This process generally entails modeling performance interference to quantify or predict the degree of

* Corresponding author.
E-mail addresses: 1119432501@qq.com (C. Xiong), linww@scut.edu.cn (W. Lin), huikanghuang0321@gmail.com (H. Huang), csjianpenglin@mail.scut.edu.cn (J. Lin), lik@newpaltz.edu (K. Li).

https://doi.org/10.1016/j.future.2025.108355
Received 24 December 2024; Received in revised form 22 December 2025; Accepted 24 December 2025
Available online 27 December 2025

Table 1
Summary of related works about interference modeling.

Reference	Method	Quantitative metrics	Datasets
[12]	Analytical/Statistical Modeling	latency, throughput	Public cloud workload traces
[13]	Analytical/Empirical Modeling	execution time, throughput	Zephyr OS workloads
[14]	Regression/Classification	latency, throughput	Multi-tenancy workload data
[15]	Deep Learning Models	memory bandwidth usage, application performance	Memory access patterns
[16]	Machine Learning Models	application performance	Cloud-native workload data
[17]	Incremental Learning	workload performance under partial interference	Serverless workload traces
[18]	Machine Learning Models	workload performance in heterogeneous settings	Data-analytics workloads
[19]	Analytical/Diagnostic Modeling	task execution time, resource utilization	Apache Spark workloads

interference arising from contention for computational resources, followed by the development of interference-aware scheduling algorithms to mitigate the problem. Constructing such models necessitates in-depth research into the causes of interference and the precise identification of metrics that significantly affect VM performance. Moreover, appropriate functional relationships must be selected to reflect the quantitative interplay between these metrics and performance interference [10]. However, current modeling approaches for performance interference, as summarized, predominantly rely on static average values of specific indicators–such as LLC miss rate and CPU utilization–while failing to incorporate a broader array of related indicators. This limitation significantly constrains the evaluation models. Some studies even depend on a single hardware counter, such as instructions per cycle (IPC), to assess performance interference [11]. Beyond metric selection, existing research exclusively employs static average values for modeling, neglecting the dynamic variations of indicators in real-world environments. As analyzed in Section 3.2.2, quantifying dynamic changes cannot be equated with computations based on average values. For certain workloads operating on this kernel, branch misprediction rates exhibit considerable variability, translation lookaside buffer (TLB) flushes occur frequently, and limitations in cache optimization techniques result in performance fluctuations due to LLC contention that may reach up to 40%. Thus, under this kernel, incorporating indicator fluctuations into the model is imperative. In contrast, time-series data can more effectively capture dynamic changes in resource usage and performance interference, providing a more accurate depiction of complex behaviors evolving over time and thereby facilitating the development of more precise performance interference models.

In interference-aware scheduling algorithms, interference is managed by either assigning appropriate workloads to VMs or migrating VMs to ensure the operational efficiency of cloud systems, keeping performance interference within acceptable limits [20]. Current related research reveals areas for improvement. For instance, many studies treat interference metrics as thresholds or aim to ensure deadlines are not exceeded, rather than focusing on enhancing throughput. Additionally, few studies integrate time-series data considerations into the scheduling approaches. Considering the improvement achieved by incorporating time-series data metrics into interference modeling, we believe it is worthwhile to incorporate temporal data considerations into scheduling to enable the determination and scheduling of appropriate loads.

In this article, we focus on the performance interference issues of compute-intensive workloads, considering additional factors and metrics related to the performance degradation of compute-intensive applications, and incorporating time-series data variability into the performance interference modeling and interference-aware scheduling for batch applications. The contributions of our research are as follows:

(1) To address the limitations of traditional interference quantification methods that primarily rely on stereotypical metrics and average values, ignoring the impact of temporal fluctuations, we conducted extensive experiments using compute-intensive benchmark programs under various co-location scenarios. Through experimental analysis, we identified several key performance metrics that accurately capture the performance interference of batch applications and incorporated time-series variability to develop a novel and more precise interference quantification model.
(2) To overcome the shortcomings of conventional scheduling algorithms that neglect temporal fluctuations and application-specific interference characteristics, we proposed an innovative scheduling algorithm for batch applications. This algorithm leverages historical data to account for temporal variations and interference differences in co-located application combinations, resulting in improved throughput, scheduling efficiency, and server load balancing.
(3) Through analysis using QQ(Quantile-Quantile) plots and quantification accuracy, we demonstrate that the fitting degree and reliability of our performance interference quantification model outperform existing approaches. Furthermore, comparisons of results in real-world environments shows that our proposed scheduling algorithm outperforms those that do not account for time-series data in terms of throughput, scheduling results, and server load balancing.

The structure of this article is as follows: Section 2 reviews related work pertinent to our study. Section 3 presents our research approach to compute-intensive load performance interference modeling. Section 4 introduces the scheduling strategy we propose for batch applications. Section 5 discusses comparisons and analyses of the performance interference model and scheduling strategy. Section 6 concludes the article.

2. Related work

Several review articles have summarized the current state of research on performance interference in VMs. For instance, Ghorbian et al. [26,27] reviewed scheduling algorithms in serverless computing, examining resource conflicts, such as CPU time slice competition, and the resulting interference. They highlighted that the consequences of interference can be quantified through metrics like latency and Service Level Agreement (SLA) violations, and noted that current studies employ resource-aware and data-aware approaches to optimize interference-aware scheduling. Similarly, Tari et al. [28] categorized research on serverless auto-scaling mechanisms, identifying performance interference as a key challenge in this domain and citing CPU isolation and memory grading as typical solutions. Strati et al. [29] also provided a summary of recent research in the increasingly prominent field of GPU performance interference. In the subsequent parts of this section, we synthesize research relevant to this article, presenting some of the representative studies on interference modeling and interference-aware scheduling in Tables 1 and 2, respectively.

2.1. Performance interference modeling

The performance interference model should first identify the shared resource responsible for interference. Subsequently, appropriate metrics should be selected to construct a systematic quantitative model

Table 2
Summary of related works about interference-aware scheduling.

Strategy	Method	Scheduling decision criteria metrics	Scheduling Improvement	Dataset
[20]	Heuristic algorithm	predicted interference, predicted resource requirements	task assurance rate, priority assurance rate, resource utilization	Google cluster data
[21]	Heuristic algorithm	predicted performance interference	batch task throughput, QoS assurance	Alibaba production cluster 24-hour trace
[22]	Reinforcement learning	system state including resource utilization and energy consumption	energy efficiency, service violation rate	Real workload traces from 800 physical machines
[23]	Traditional algorithm	priority of tasks	response time, throughput, resource utilization	not specified
[24]	Traditional algorithmd (Queueing theory)	priority policy	task completion time, system throughput	not specified
[11]	Heuristic algorithm	task queues, deadlines, server capability, predicted execution time	processing time, cost, SLA violation rate, delay	CloudSimPlus simulated data
[25]	Heuristic algorithm	CPU resource reservation, differentiated shares, node scoring	CPU utilization control, minimal interference with high-priority jobs	Not specified

[30]. Existing modeling approaches typically depend on hardware performance counters to capture CPU, memory, network, and I/O usage metrics. These metrics are then used to build performance interference models, often using the average values of key metrics. For example, Koh et al. modeled interference for diverse VM types, such as memory-intensive, disk-intensive, or hybrid, based on ten performance events [31]. However, interference characteristics vary significantly across workload types. Therefore, it is essential to model performance interference resulting from contention for specific resources. For instance, Lu et al. [32] and Tseng et al. [33] studied interference caused by I/O-intensive operations, while Tzenetopoulos et al. focused on interference from co-locating compute-intensive workloads [18]. Beyond examining different resource demand patterns, some researchers categorize workloads by allocation strategies–such as priority or QoS–into types like best-effort, batch, latency-critical, and HPC. They then developed category-specific models [34]. For example, Shah et al. analyzed performance interference in identifying HPC workloads on Apache Spark [19], Bu et al. proposed a linear interference prediction model for MapReduce applications [35], and Pons et al. developed a regression model quantifying QoS degradation due to interference for latency-critical VMs [12].

To quantify or predict the degree of performance interference for different workload types, researchers select various indicators for modeling. Metrics related to inadequately isolated shared resources are crucial. For example, the Last-Level Cache (LLC) miss rate often creates performance bottlenecks and unpredictable interference, whether viewed from a VM or data center perspective [13,36]. Additionally, contention for shared resources such as memory bandwidth, network bandwidth, and disk bandwidth can also lead to performance degradation [14]. Metrics like these can be selected to quantify interference. For example, Yao et al. utilized deep learning methods to select memory bandwidth and similar metrics for interference quantification [15]. In addition to shared resource contention, workload-specific characteristics can also inform the selection of metrics. Baluta et al. incorporated the resource demand intensity of workloads into a layered queuing model to quantify interference [16]. Zhao et al. used tail latency for serverless workloads, job completion times for short-term computing tasks, and Instructions Per Cycle (IPC) for latency-sensitive workloads to quantify localized interference [17]. Xavier et al. developed an interference quantification model across-applications based on metrics such as average service time, arrival rate, execution time, and clock interrupt rate [37].

2.2. Interference-aware scheduling algorithm for batch applications

Current research on batch applications scheduling in cloud data centers primarily focuses on resource utilization [38], priority [23,24], and load balancing [39,40]. In the domain of interference-aware scheduling, current research typically identifies and quantifies resource contention and performance impacts between applications through performance benchmarking, prior knowledge, or historical operational data, and then formulates scheduling strategies based on different optimization approaches. However, existing interference-aware scheduling solutions for batch applications generally follow three main optimization directions and rarely account for temporal fluctuations.

The one is to select the most optimal or relatively optimal allocation scenario to alleviate performance interference. This approach ensures that the chosen allocation scheme exhibits better interference performance than other alternatives. For example, Verbowen et al. used an application performance interference prediction model combined with energy efficiency optimization to allocate applications to scenarios with relatively lower interference [22]. Similarly, Anu et al. proposed a performance interference model for VM migration, selecting the VM to be migrated and the destination physical host based on the principle of minimum interference [41]. The models used in these studies are primarily based on the mean-value models [11], and do not consider scenarios with significant resource utilization fluctuations.

The second approach requires that the performance interference of the selected scheduling scheme does not exceed a predefined threshold [20]. For instance, in Jersak et al.'s scheduling method, a server is selected as the final allocation scheme only if the performance interference of the VM to be scheduled, after allocation, does not exceed the threshold [42]. Nabavinejad et al. introduced an interference-aware optimization strategy to mitigate data skew in Hadoop, designing a dynamic interference threshold [43]. Wang et al. proposed deciding whether to migrate the victim VM or the culprit VM based on the number of VMs exceeding the interference threshold [44].

The third is to ensure that batch applications meet deadlines [21,45]. This strategy is often employed when compute-intensive batch applications are co-located with other applications. When applications requiring real-time response or interaction applications are co-located with non-interactive batch applications, the former, being delay-sensitive, may experience severe degradation or failure if not executed promptly, leading to SLA violations. To tackle this, a common strategy prioritizes the execution of latency-sensitive applications and schedules batch

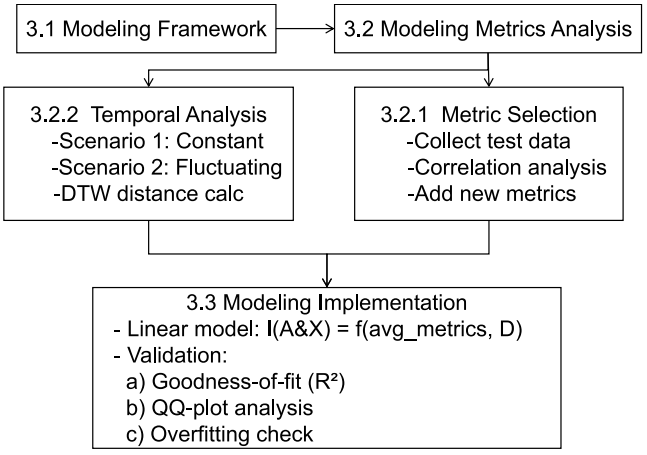


Fig. 1. The framework diagram of Section 3.

applications according to the minimum standard: meeting the SLA by completing execution before the deadline [22,25].

3. Performance interference modeling of compute-intensive loads

3.1. Modeling framework

In this section, we introduce our modeling approach and analyze experimental results to derive the final model. The framework diagram for this section is shown in Fig. 1. As summarized earlier, existing studies tend to use fixed performance metrics. Therefore, we investigate whether additional performance metrics are associated with performance interference. Furthermore, we analyze the temporal variability factor, which has been rarely considered in prior research. A quantitative model typically consists of three key components: the model output, the model input, and the functional form.

The model output represents the target of the modeling process. In a performance interference quantification model, the output is the interference measure. In this study, we define performance interference as the relative performance loss, calculated as the difference between the observed performance in a real environment and the ideal performance under isolated conditions, divided by the ideal performance. For example, consider a physical machine (PM) hosting two VMs: one running workload A, and the other running workload bg. When workload A runs alone, its execution time is $\boldsymbol{T_A}$. When both VMs run concurrently, the execution time of workload A becomes $\boldsymbol{T_{A\&bg}}$. The performance interference experienced by workload A due to colocation can then be expressed as,

$$I(A\&bg) = \frac{T_{A\&bg} - T_A}{T_A}, \tag{1}$$

where $\boldsymbol{I(A\&bg)}$ represents the performance interference loss experienced by workload A due to co-location with the background benchmark bg. The input of the model is the data that can affect the interference measurement, and this part requires experimental analysis and validation to ensure the critical factors affecting performance interference are captured accurately. Lastly, determining the functional form of the model involves exploring various mathematical relationships between inputs and the interference metric. This step evaluates the trade-offs between model accuracy and the ability to avoid overfitting, ensuring the model generalizes well across different scenarios.

3.2. Modeling metrics analysis

In this subsection, we discuss the model inputs based on experimental analysis. To identify which data can affect the interference measure, we first need to determine which performance metrics are correlated with interference. Additionally, we explore whether there are other performance metrics beyond traditional ones. Once the relevant performance metrics are selected, we also investigate whether the time-series variations (not just the average values over a period) of these metrics affect the interference measure. The experiments in this section were conducted on a host machine running Linux 6.1, with hyper-threading explicitly disabled. The host is equipped with an Intel(R) Xeon(R) CPU E5-2620 v2, with a base frequency of 2.10 GHz and 6 cores per physical CPU. Its cache hierarchy comprises 32 KB L1 data cache (L1d) and L1 instruction cache (L1i) per core, 256 KB L2 cache, and 15360 KB L3 cache. Two virtual machines (VMs) were configured on this host: one running a workload benchmark (referred to as the test benchmark), and the other executing a background benchmark. Both VMs share identical configurations, with cache sizes as follows: L1d: 32 KB, L1i: 32 KB, L2: 4096 KB, L3: 16384 KB.

3.2.1. Selection of performance metrics

Traditional performance metrics. CPU utilization [35] and LLC miss rate [13,44] are traditional metrics used in modeling performance interference for compute-intensive workloads. CPU utilization reflects the extent to which a VM utilizes the CPU resources of the host. When a VM with high CPU utilization runs co-located with other VMs, contention for limited host CPU resources becomes more intense, leading to performance degradation of co-located VMs. LLC miss rate refers to the probability that data required by a VM is not found in the last level cache of a processor. When VMs share computational resources, they contend with other VMs for limited cache resources, resulting in a decrease in cache hit rates and ultimately reducing their performance.

Other performance metrics . To analyze whether other performance metrics correlate with performance interference measurements, we conduct co-location tests where one VM runs the benchmark under test (from SPEC CPU2017) and the other runs a background benchmark, which could be either from SPEC CPU2017 or sysbench. For example, to investigate the performance interference characteristics of application A, we run it as the benchmark under test, co-located with different background benchmarks B, C, and D. Before the co-location tests, we run application A alone to collect its related performance metrics and execution time T_A, and we also run benchmarks B, C, and D individually to gather their respective performance metrics. Next, we perform co-location experiments by running application A with benchmarks B, C, and D separately. We record the execution time of application A in these co-location scenarios: $T_{A\&B}$, $T_{A\&C}$, $T_{A\&D}$. Finally, the performance interference loss degree of program A after co-located with the background benchmark, as well as the relevant performance metrics collected when co-located or running separately, are listed line by line. For each pair, we compute the Pearson correlation coefficient to identify which performance metrics are strongly correlated with the interference measure. Three performance metrics are found to have a high correlation with the interference measure, and we summarize the Pearson correlation coefficients between the different test benchmarks and their background benchmarks in Table 3.

For the cycles metric, it counts the number of processor clock cycles. When executing compute-intensive applications, the processor needs to execute a large number of instructions and operations, leading to a significant amount of time spent on execution. As a result, the number of processor clock cycles increases, and thus, the cycles counter value is higher. This metric can serve as an indicator of the time complexity and resource demand of an application.

For the branch misprediction rate metric, when an application experiences a branch misprediction, the processor must flush its instruction pipeline and re-execute the branch instruction. If another application is running concurrently, it may need to wait for the execution of the first application to complete before proceeding, a phenomenon known as "branch prediction contention." During co-located execution, two applications may contend for the processor's branch prediction cache,

Table 3
Three metrics strongly correlated with interference measurement.

	lbm	nab	exchange2	deepsjeng	roms	leela	omnetpp	mcf	cactuBSSN
cycles	0.86	0.57	0.45	0.78	0.67	0.68	0.85	0.83	0.58
branch-misses-rates	0.5	0.54	0.43	0.41	0.5	0.51	0.47	0.5	0.59
TLB-loads-misses	0.68	0.55	0.44	0.49	0.68	0.79	0.74	0.5	0.51

Table 4
Changes in LLC misses and TLB misses for cactuBSSN under different co-location scenarios.

	alone	1 cactuBSSN	2 cactuBSSN	deepsjeng	imagick	sysbench CPU (threads = 8)
LLC misses (%)	0.191399	0.217419	0.26062	0.241375	0.208661	0.203825
TLB misses$\times 10^{-3}$ (%)	0.0377	0.076	0.16	0.159	0.0483	0.0567

increasing the likelihood of branch mispredictions. This is especially evident when both applications have a high branch misprediction rate, further negatively impacting performance.

For the TLB (Translation Lookaside Buffer) load miss count metric, the CPU needs to translate virtual addresses to physical addresses to access data from main memory. If the TLB has already cached the mapping between virtual and physical addresses, the CPU can directly use the physical address, avoiding expensive memory access operations. However, if the mapping is not in the TLB, the CPU must access the main memory, which can be time-consuming and lead to a decrease in CPU performance. When running co-located, two applications may contend for the same TLB cache, increasing the probability of TLB misses. If the TLB miss rates of two applications are both high, they may interfere with each other, causing further performance degradation.

Correlation analysis of LLC misses and TLB misses with interference. Unlike the behavior in isolation, combined with the analysis in the preceding two parts of Section 3.2.1, LLC misses and TLB misses correlate with performance interference for compute-intensive workloads. We attribute this to two factors, based on our analysis of SPEC CPU2017 and supplementary experiments (as shown in Table 4, averaged over 15 trials): firstly, some benchmarks are compute-intensive while also being memory-sensitive; secondly, the pollution and superposition effects during co-location.

First, we analyze the first reason. In SPEC CPU2017, `mcf` performs extensive linked list traversals and random memory accesses, making it cache-unfriendly. `Imagick` has large working sets and mixed access patterns, which also challenge the cache. Meanwhile, applications like `fotonik3d` and `roms` handle large multidimensional arrays with non-contiguous or strided accesses, increasing TLB pressure. Although the benchmarks in SPEC CPU2017 are all compute-intensive and minimize non-CPU resource use, the proportion of memory-sensitive applications remains considerable, reflecting real-world scenarios. Our experimental design must account for these factors; otherwise, the accuracy and practicality of our model for real-world cloud workloads would be compromised.

Next, we analyze the second reason. Benchmarks like `cactuBSSN` are very close to "pure" CPU resource consumption, with low memory intensity. As shown in Table 4, experiments found that when co-located with itself (1 or 2 instances) or other applications (regardless of memory sensitivity), both LLC and TLB misses increase. Even with low memory intensity, such increases alter actual execution time: a lower LLC hit rate raises memory access counts, prolonging execution; TLB misses force the CPU to query in-memory multi-level page tables for address translation, a process requiring multiple memory accesses. We attribute this performance interference to two main factors. First, the pollution effect: even applications that are primarily CPU-bound with small working sets may experience frequent eviction of critical data from the cache (cache thrashing) due to other threads or loads from another VM. This leads to “induced cache misses,” which do not occur in isolation but emerge under interference. Second, the superposition effect plays a role:

Table 5
Changes in system-level behavior and its corresponding impact on LLC and TLB misses for cactuBSSN under different scenarios.

	1 VM	2 VMs	3 VMs	4 VMs	5 VMs
LLC misses (%)	0.195	0.219	0.259	0.308	0.351
TLB misses$\times 10^{-3}$ (%)	0.038	0.077	0.16	0.183	0.267
context-switches (times/second)	8650	12102	13750	14738	15351
cpu-migrations (times/second)	59	67	162	165	150

the default configuration of SPEC CPU2017 6.X benchmarks employs multi-threaded execution. When multiple vCPUs from one VM run concurrently with those from another VM, contention for TLB entries intensifies. I/O operations from any application may cause TLB flushes or misses. Even if a single application exhibits a low TLB miss rate, the collective system pressure can still degrade performance. Moreover, in subsequent scheduling studies presented in later chapters, as the number of co-located applications and VMs increases, the impact of these two metrics on actual execution time becomes even more significant.

The relationship between system-level behavior and the increase in LLC and TLB misses. To further investigate the significant increase in LLC/TLB misses of compute-intensive, low memory-intensity applications in co-located environments, we conducted quantitative experiments focusing on system-level scheduling behaviors, specifically context switches and CPU migrations. We first executed the `cactuBSSN` workload on an isolated VM and collected the baseline metrics presented in Table 5. We then incrementally co-located this VM with additional instances running the same workload, gathering corresponding data for each configuration. Each scenario was repeated 15 times, with the average values recorded in Table 5. It can be observed that as the number of co-located VMs increases, the number of context switches rises from approximately 8650 per second to about 15,351 per second. Each context switch leads to partial or complete replacement of the working set of the current process in the CPU cache, directly causing LLC and TLB misses. This mechanism explains why, even for compute-intensive applications with relatively low memory access intensity (such as `cactuBSSN`), the cache miss rate increases significantly with higher co-location intensity. Regarding CPU migrations, a notable trend emerges: while the migration count increases markedly from 1 to 3 co-located VMs, its growth rate diminishes sharply at 4 VMs, and the count even slightly decreases at 5 VMs. We attribute this to the scheduler’s adaptive behavior under extreme resource contention. When CPU saturation is high, the performance gain from migration may be outweighed by the overhead of cache and pipeline flushes. Consequently, the scheduler reduces such counterproductive migrations. Collectively, the above findings provide empirical support for the previously proposed "pollution and superposition effect".

Next, we examined VM memory usage under various co-location scenarios. During the aforementioned experimental period, we monitored the %MEM readings of the qemu-kvm processes corresponding to each

VM via the top command on the host. These readings exhibited no correlation with the number of co-located VMs. This observation remained true whether a VM was idle or executing the `cactuBSSN` workload; in some instances, the %MEM of a loaded VM was even lower than when idle. This indicates that under low memory-intensity workloads, the proportion of physical memory (RSS) allocated by the host to each VM's qemu-kvm process remains relatively stable. Subsequently, we measured memory usage from within the subject VM's perspective using the free -h command. We first ran `cactuBSSN` on a single VM and recorded its internal memory statistics. We then incrementally co-located this VM with 1 to 4 other VMs running the same workload, collecting internal memory data at each step. The results showed that, from the viewpoint of the guest OS, the available virtual memory capacity and free memory statistics remained stable. Furthermore, analysis of data collected via the vmstat command revealed that the minimum and final free memory readings were almost identical across configurations from one to five co-located VMs. These findings collectively rule out significant memory pressure, large-scale page swapping, or other compulsory memory management operations due to memory overcommitment. In conclusion, we substantiate that the observed increase in LLC and TLB misses under such co-location scenarios is unrelated to non-trivial memory usage.

3.2.2. Time series differences in performance metrics

Traditional modeling methods typically construct models based solely on the mean values of performance metrics, without considering the temporal variations in these metrics for co-located VMs. However, relying solely on the mean may not accurately predict the performance interference experienced by co-located VMs. For example, when VM A and VM B are co-located, the average CPU utilization of VM B is 75%. However, if the temporal pattern of CPU utilization differs, the resulting performance interference on VM A may also differ. To illustrate this, we set up two scenarios for VM B:

Scenario 1: Virtual machine B maintains a constant CPU utilization of 60%.

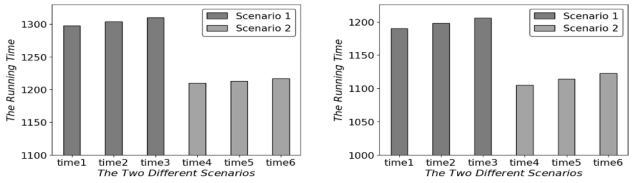
Scenario 2: The CPU utilization of virtual machine B is 90% for half of the time and 30% for the other half.

We calculate the degree of interference that VM B causes to VM A in the two different scenarios. In the first scenario, where VM B maintains a constant CPU utilization of 60%, the interference in the two given period are denoted as $I^1_{0.6}$ and $I^2_{0.6}$, respectively. In the second scenario, where VM B fluctuates between 90% and 30% CPU utilization, the interference during the first and second periods are denoted as $I^1_{0.9}$ and $I^2_{0.3}$, respectively. It is clear that the interference values satisfy : $I^1_{0.9} > I^1_{0.6} = I^2_{0.6} > I^2_{0.3}$. However, the sum of $I^1_{0.6}$ and $I^2_{0.6}$ may not be equal to the sum of $I^1_{0.9}$ and $I^2_{0.3}$. This is because the performance impact on VM A when co-located with a VM that has 90% CPU utilization is significantly greater than when co-located with a VM at 60%. Additionally, the benefit of co-location with a VM at 30% utilization may not be sufficient to offset the performance loss incurred during the high-utilization (90%) period. In fact, co-location with a VM at 30% utilization may have minimal impact on the performance of VM A, as CPU resources are likely sufficiently available.

We conducted an experiment to compare the two scenarios described above. As shown in Table 6, one VM ran the SPEC CPU2017 benchmarks [46], while the other VM ran the sysbench [47] CPU testing function. The SPEC VM executed the 607.cactuBSSN_s and 638.imagick_s reseparately, while the sysbench VM maintained approximately 60% CPU utilization throughout in Scenario 1. In Scenario 2, the sysbench VM maintained approximately 90% CPU utilization during the first half of execution and approximately 30% during the second half. The sysbench parameters are detailed in Table 6. The execution times for 607.cactuBSSN_s and 638.imagick_s under co-location in both scenarios are presented in Fig. 2. Each scenario was executed three times. The execution time of the two benchmarks in Scenario 1 is lower than that in Scenario 2. This finding demonstrates that the sum of $I^1_{0.6}$ and $I^2_{0.6}$ in Scenario 1 is less than that of $I^1_{0.9}$ and $I^2_{0.3}$ in Scenario 2. It also highlights that relying solely on the mean values of performance metrics cannot accurately measure performance interference.

Table 6
The settings for sysbench in two scenarios.

	cpu-max-prime	Scenario 1 threads time	Scenario 2 threads1 time1 threads2 time2
607.cactuBSSN_s	9999999	8 1200	18 600 4 600
638.imagick_s	9999999	8 1200	18 600 4 600



(a) 607.cactuBSSN_s (b) 638.imagick_s

Fig. 2. The execution time of 607.cactuBSSN_s and 638.imagick_s in two co-location scenarios respectively.

Therefore, we propose incorporating the temporal variation of the same performance metric from co-located VMs as an independent variable in the model. For instance, if the workload on VM A runs standalone for 900 seconds, and the workload on VM B runs standalone for 2000

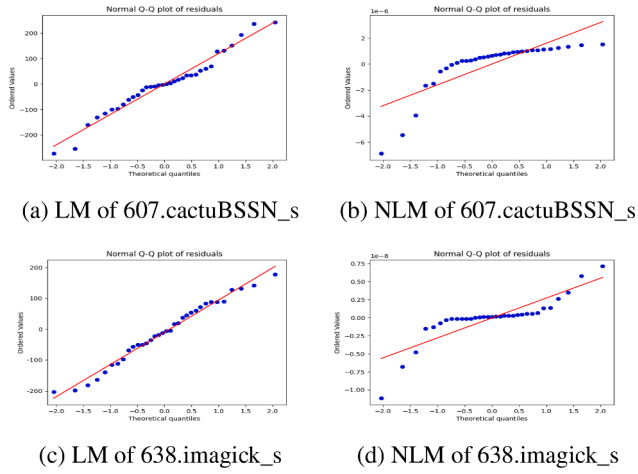
seconds, we calculate the similarity/difference between the temporal patterns of the relevant performance metrics from VM A during its 900 seconds of standalone execution and from VM B during its first 900 seconds of standalone execution. This measure of similarity/difference then included as an independent variable in the model. If VM B has already been running for 1100 seconds before VM A starts execution, the similarity/difference is calculated between the temporal patterns of the relevant performance metrics from VM A during its standalone execution and from VM B during its final 900 seconds of standalone execution.

3.3. Modeling implementation

Based on the experimental analysis in Section 3.2, the average values of the five performance metrics and the temporal differences are added to the model input. The output of the model is shown in Eq. (1). This section discusses the function relationship f, and the model form is shown in Eq. (2). Before introducing the equation, we need to introduce the concept of handling temporal differences: Dynamic Time Warping (DTW) [48]. DTW is an algorithm employed to measure the similarity between two time series. It performs elastic alignment along the temporal dimension to accommodate sequences of differing lengths or out-of-phase temporal progression. In this study, we apply DTW to compute the distance between the performance metric time series of two individual VMs during standalone operation, thereby capturing and quantifying the temporal fluctuation differences between the metrics. The form of Eq. (2) is as follows,

$$I(A\&X) = f(x_1, x_2, \dots, x_5, D(a_1\&x_1), \dots, D(a_5\&x_5)), \tag{2}$$

where A and X represent the two co-located VMs. The variables x_1 to x_5 denote the average values of CPU utilization, LLC miss rate, cycles, branch misprediction rate, and TLB load miss rate for the standalone workload on VM X. Similarly, the variables a_1 to a_5 represent the corresponding average values of the five metrics for VM A when running standalone. The terms $D(a_1\&x_1)$ to $D(a_5\&x_5)$ represent the temporal DTW (Dynamic Time Warping) distances between the respective performance metrics of VM A and X during standalone execution.The nature of the function f, whether it exhibits a linear or nonlinear relationship, warrants further discussion. We need to comprehensively consider the accuracy of different model forms (e.g., the degree of fitting) as well as



(a) LM of 607.cactuBSSN_s (b) NLM of 607.cactuBSSN_s

(c) LM of 638.imagick_s (d) NLM of 638.imagick_s

Fig. 3. Figure 3(a) and 3(b) show the QQ plots for the linear **(LM)** and nonlinear **(NLM)** models of performance interference prediction for the VM running the 607.cactuBSSN_s workload, respectively. Figure 3(c) and 3(d) show those for the VM running the 638.imagick_s workload, respectively.

their reliability (e.g., whether overfitting occurs) in the following sections.

We use Goodness of Fit [49] to evaluate the goodness of fit of the model to the data. In this study, the Goodness of Fit is measured using the coefficient of determination R^2, which indicates the percentage of variance in the data explained by the model. The equations of R^2 are shown in Eq. (3)–(5):

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}, \tag{3}$$

$$SS_{\text{res}} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{4}$$

$$SS_{\text{tot}} = \sum_{i=1}^{n}(y_i - \bar{y})^2, \tag{5}$$

where y_i is the actual observed value of the dependent variable, $\hat{y}_i$ is the predicted value of the model, $\bar{y}$ is the mean of the dependent variable, and n is the sample size. Specifically, representing the proportion of total variance accounted for by the model and ranges from 0 to 1. Higher values, particularly those above 0.75, are generally considered acceptable for indicating a good model fit. However, fitting the data well does not necessarily mean that the model is optimal. Issues such as overfitting may lead to excellent fitting of the training data but poor predictive performance on new data. Therefore, in regression analysis, in addition to examining the goodness-of-fit, residual analysis should be conducted to assess issues related to the model's reliability. Residual analysis in this study includes the use of QQ plots (quantile-quantile plots [50]). A QQ plot compares the residuals to theoretical quantiles, typically assuming a normal distribution. If the residual points are evenly distributed around the theoretical quantiles, this suggests normality. Conversely, significant deviations indicate a violation of the normality assumption.

We set the loads of the VMs under test to 607.cactuBSSN_s and 638.imagick_s, respectively, and performed co-location experiments with the VM running different workloads. Both nonlinear and linear models were used to predict the performance interference. It was found that the nonlinear model with an exponent of 4 achieved the best fit, with R^2 values of 1 and 0.99, respectively. However, as shown in Fig. 3(b) and (d), the QQ plot trends indicate that the nonlinear model suffers from overfitting. In contrast, while the linear model produced R^2 values of 0.85 and 0.88, the QQ plot trends showed that the residuals of the model approximately followed a normal distribution, with no signs of overfitting. Furthermore, with R^2 values above 0.75, the fit is considered acceptable. The QQ plots for the linear and nonlinear performance interference models for both workloads are shown in Fig. 3. After considering both the goodness of fit and the model's accuracy and reliability, the linear function form is adopted for the model in Eq. (2).

4. Interference-aware scheduling algorithm for compute-intensive batch applications

4.1. Performance interference in batch applications scheduling

As mentioned earlier, it is a common practice to place multiple batch applications on the same VM. These applications typically require high computational resource, involving extensive data processing and scientific computing. The interference in co-located execution arises from competition for computational resources both between applications running on the same VM and between VMs hosted on the same physical host. Consider application A as a batch application being scheduled. In the remainder of this section, we define its performance interference at both the VM level and the physical machine level.

Consider a scheduling scenario where a VM runs multiple instances of application A alongside other co-located applications. The performance of application A on this VM may deviate from its ideal state due to resource contention. For a single instance of A, we measure interference as the difference between its ideal execution time and its actual execution time. When multiple instances of A run on the same VM, their overall performance interference level is the average interference across all instances. We propose a formula to define the level of interference experienced by batch applications within a VM, as shown in Eq. (6):

$$I(A|VM) = \frac{1}{n} \cdot \sum_{i=1}^{n} \frac{T(A_i|VM)}{T(A)}, \tag{6}$$

where $I(A|VM)$ is the performance interference of A in the virtual machine VM, n is the number of A instances in the virtual machine VM, $T(A)$ is the ideal execution time of A, and $T(A_i|VM)$ is the actual execution time of the i-th instance of application A in virtual machine VM.

How to evaluate the performance interference suffered by application A when multiple applications A are running on a server is a challenging topic. Scheduling A to a server hosting more applications may not necessarily yield worse interference than scheduling it to one with fewer applications. This occurs because a server with a higher total application count might still have VMs sparsely loaded, while allocating A to a server with fewer total applications could place it on a VM with high A-instance density, potentially worsening interference. Our tests (Table 7) demonstrate this: Server 1 (15 "perlbench" instances) achieved better performance for some instances than Server 2 (14 "perlbench" instances) under identical VM configurations.

Based on the analysis, we conclude that the performance of application A is not only related to the total number of applications on the server, but also to the co-location scenarios of each VM on the server and the number of applications running on the VM where application A is located. We propose the concept of overall performance interference sensitivity for an application on a server, which is the weighted sum of the performance interference of each VM running application A. Since this part is used to evaluate the potential performance interference an application A may face during scheduling, we emphasize the difficulty faced by VMs running more applications by assigning them higher weights, while giving lower weights to VMs running fewer applications. The denominator for the weight of a VM is the total number of applications across all VMs running application A, and the numerator is the total number of applications running on that specific VM. The formula is as follows,

$$I(A|Server) = \sum_{i=1}^{n}(\frac{N_i}{N} \cdot I(A|VM_i)), \tag{7}$$

where $I(A|Server)$ represents the overall performance interference sensitivity of application A on server Server, VM_i is the i-th VM that is

Table 7
The completion time of each application in different co-location scenarios.

	co-located scenarios	VM1	VM2	VM3	VM4	VM5
Server1 (14 apps of perlbench)	vm1: 3, vm2: 3, vm3: 3, vm4: 3, vm5: 2	3054–3058	3061–3064	3063–3065	3048–3052	2208–2210
Server2 (15 apps of perlbench)	vm1: 5, vm2: 4, vm3: 3, vm4: 2, vm5: 1	3187–3192	3059–3063	2600–2602	1978–1981	1151

running application A on Server, N_i is the number of applications running on VM VM_i and N is the total number of applications across all VMs on the server Server.

4.2. Interference-aware scheduling based on historical information and temporal data DTW distance

The proposed server selection algorithm and the VM selection algorithm are presented in Algorithm 1 and Algorithm 2. Our scheduling scheme first needs to collect prior historical information, which includes the actual completion time of each application in different co-location combinations and the time-series data of various performance metrics of the server throughout the entire running phase. We set up a scenario where five VMs on a server run three different applications: POV-Ray, NAB, and Perlbench. To further simulate performance fluctuations, we introduce a periodic load wave, which causes a 30% fluctuation in the host CPU utilization. We collected performance data for these three applications across 1500 different co-location combinations. A brief introduction to the three applications is provided below:

- POV-Ray [51]: A ray tracing program used for image rendering, which is computation-intensive.
- NAB [52]: A computational application used to simulate numerical models for biological molecules, which is also computation-intensive.
- Perlbench [53]: Primarily used to evaluate the performance of scripting languages. However, in the SPEC benchmark, it is executed as a long-running batch job.

Based on prior historical information, our scheduling algorithm is divided into two parts: server selection algorithm and VM selection algorithm. The framework diagram of the scheduling algorithm is shown in Fig. 4. The server selection algorithm evaluates the interference sensitivity of the application to be scheduled on a server by analyzing the time-series data fluctuations of the server during a specific period. Based on the expected performance of the application on different servers, the algorithm decides which server is the most appropriate for scheduling the application. Once the server is chosen, the VM selection algorithm determines which VM within the selected server should host the applications. This decision is made based on the analysis of the number and type of applications running on each VM, aiming to allocate the application to a VM where it will experience the least interference, taking into account the resource demands and potential contention with other co-located applications.

When the application to be scheduled arrives, for each server, calculate the DTW distance of the time series data collected from that server and the time series data in the correlated co-occurrence combination with the same number of applications in the prior collection data, and normalize the calculated DTW distance of each performance counter. Then, for the specific application under consideration, removing the co-located combinations without the application, we found that the overall performance interference sensitivity of the application corresponding to the prior data with the shortest distance was not significantly different from the performance interference sensitivity of the application on the current server. Therefore, the algorithm periodically calculates the DTW distance between the performance counters of each server and the relevant prior data with the same number of applications, normalizes these values, compares the distances when the application to be scheduled appears, selects the set of prior data closest to the current server, considers the overall performance interference degree of the application in

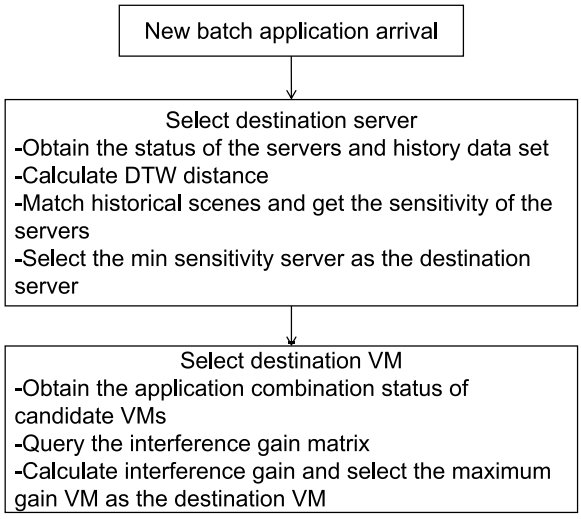


Fig. 4. The framework diagram of Section 4.

this prior scenario as the that of the application on the current server, compares the expected overall performance interference sensitivity of the applications to be scheduled on each current server, and selects the server with the lowest overall interference sensitivity as the destination server.

After selecting the destination server, choose the destination VM. There are multiple VMs on candidate servers. The VM with the fewest applications is prioritized as the candidate VM. If multiple candidate VMs exist, an analysis of the application co-location combinations on each VM is performed. In our example scenario, we need to determine how the three specific applications are suitable for co-location. We defined the following concepts and conducted an analysis based on these concepts. First, we assumed the scenario where a VM runs 5 application instances as the test load scenario, as shown in Table 8. Subsequently, we define the homogeneous interference attenuation coefficient to quantify the performance degradation of identical applications when co-located under the maximum load scenario:

$$\alpha_i = \frac{T^i_{\text{isolated}}}{T^i_{\text{5-instances}}}, \tag{8}$$

where T^i_{isolated} is the completion time of a single instance of application i in an isolated environment, and $T^i_{\text{5-instances}}$ is the total completion time of application i instances under the test load scenario. Furthermore, under the maximum load scenario, the theoretical completion time of application i when co-located with a different application j, inferred based on the performance when all co-located applications are instances of i, is:

$$T^{\text{co-loc}}_{\text{theory}}(n_i \cdot i \mid n_j \cdot j) = \frac{T^i_{\text{isolated}}}{\alpha_i \cdot \left(\frac{n}{n_i}\right)}, \tag{9}$$

where n_i is the number of instances of application i on the VM, n_j is the number of instances of application j on the VM, and $n = n_i + n_j$. We quantify the gain for this specific co-location combination by combining the theoretical performance with the actual performance observed in historical data:

$$G(n_i \cdot i \mid n_j \cdot j) = \frac{T^{\text{co-loc}}_{\text{theory}}(n_i \cdot i \mid n_j \cdot j)}{T^{\text{co-loc}}_{\text{hist}}(n_i \cdot i \mid n_j \cdot j)} - 1, \tag{10}$$

Table 8
The completion time of each application in different co-location scenarios.

Application	Run alone	Maximum load scenario same applications	3nab + 2povray	3nab + 2perlbench	3perlbench + 2povray
Nab	549	4800	4746	4698	\
Perlbench	524	5802	\	5046	5832
Povray	551	6298	5205	\	5989

where $T(n_i.i|n_j.j)^{\text{co-loc}}_{\text{hist}}$ is the actual performance observed in historical data for this combination. A G value greater than 0 indicates that, for the same total number of applications, the interference experienced by application i when co-located with application set J (here, primarily j) is less than the scenario where all co-located applications are i. Conversely, a G value less than 0 indicates greater interference. Based on our analysis of historical data, we determined that the interference effect of application co-location is fully manifested when $n_i = 3$ and $n_j = 2$. The G value under this specific configuration ($G(3_i.i|2_j.j)$) most intuitively reflects the gain (or loss) of the scenario where application i and application j are co-located compared to the scenario where all co-located applications are i. Therefore, we define $F(i|j)$ as $G(3_i.i|2_j.j)$. A positive value of $F(i|j)$ indicates that application i exhibits better performance when co-located with application j than when co-located with homogeneous instances. This implies that application i is suitable for co-location with application j. The F metric enables determination of compatible co-location partners for a given application based on mutual interference characteristics. Furthermore, by evaluating the values from the perspective of other applications, F facilitates assessment of an application's co-location friendliness toward others. This framework guides destination VM selection. For instance, when a `perlbench` instance arrives, consider scheduling it to a candidate VM hosting two `nab` instances and one `povray` instance. The resulting gain is calculated as:

$$\text{Gain} = 2F(\text{perlbench} \mid \text{nab}) + F(\text{perlbench} \mid \text{povray}). \tag{11}$$

For subsequent candidate VMs, the *Gain* is similarly computed. The VM exhibiting the maximum *Gain* value is ultimately selected as the destination VM.

4.3. Real-world case study

The scenario proposed in this article concerns batch task scheduling under the constraints of limited physical machines and VMs. To further illustrate the universality of this scenario, some real-world cases will be presented. Firstly, the scheduling of multiple MapReduce tasks in a classic Hadoop cluster may be similar to this scenario, where these tasks need to run on a limited number of physical servers and VMs. The same applies to Spark clusters. Another example is provided: a financial services company utilizes a private cloud platform with a fixed VM pool to handle high-frequency trading [54]. We assume that this platform comprises 10 servers, each fixedly hosting 5 VMs, with each VM configured with multiple CPU cores. The task scheduling involves various types of compute-intensive batch tasks, such as risk analysis, portfolio optimization, and market simulation. These tasks require high throughput to support massive trading decisions. The Performance metrics shown in Eq. (2) for different task categories exhibit variations. Therefore, interference models are established for the VM running these different types of tasks, and server selection and VM selection are carried out in accordance with the scheduling algorithm in this section. The platform executes batch tasks (e.g., risk analysis and market simulation) multiple times a day, with approximately 200 tasks executed each time. The interference-aware algorithm is conducive to reducing the total task completion time and increasing system throughput.

Algorithm 1 Interference-aware destination server selection algorithm.

Symbol:
S: Set of servers $\{s_1, s_2, \ldots, s_d\}$
H: Historical dataset $\{(X_i, I_i)\}$ where
  X_i = Time-series matrix of performance metrics
  I_i = Interference sensitivity vector
X_{new}: Performance metrics time-series of new app
M: Set of performance metrics $\{m_1, m_2, \ldots, m_D\}$ (e.g., CPU, LLC, Branch, TLB)
Input:
S, H, X_{new}
Output:
s_d: Selected target server

function SELECT_SERVER(S, H, X_{new})
  min_sensitivity $\leftarrow \infty$
  $s_d \leftarrow$ null
  for each server s **in** S **do**
    $H_s \leftarrow \{(X, I) \in H \mid |\text{VM}_s| = |\text{VM}_{\text{hist}}|\}$ ▷ Extract same-scale historical data
    $\text{dist}_{\min} \leftarrow \infty$
    for each (X, I) **in** H_s **do**
      $d \leftarrow 0$
      for each metric m **in** M **do**
        $X_{\text{hist}} \leftarrow X[:, \text{metric\_index}(m)]$
        $X_{\text{app}} \leftarrow X_{\text{new}}[:, \text{metric\_index}(m)]$
        $d \leftarrow d + \text{DTW}(X_{\text{app}}, X_{\text{hist}})$
      end for
      if $d < \text{dist}_{\min}$ **then**
        $\text{dist}_{\min} \leftarrow d$
        $I_{\text{ref}} \leftarrow I$ ▷ Record sensitivity of closest match
      end if
    end for
    $I_s \leftarrow I_{\text{ref}}$ ▷ Set current sensitivity
    sensitivity $\leftarrow \|I_s\|_1$ ▷ L1 norm summation
    if sensitivity < min_sensitivity **then**
      min_sensitivity $\leftarrow$ sensitivity
      $s_d \leftarrow s$
    end if
  end for
  return s_d
end function

5. Experiment

5.1. Experimental setup

In the performance interference model experiment, the hardware setup remains the same as in Section 3.2. In the scheduling algorithm experiment, we established environments to demonstrate the effectiveness of the scheduling algorithm, which include three scenarios with different numbers of servers: scenarios with 6 servers, 7 servers, and 8 servers. Specifically, in the 6-server scenario, there is 1 control node and 5 scheduling nodes; in the 7-server scenario, there is 1 control node and 6 computing nodes; and in the 8-server scenario, there is 1 control node and 7 computing nodes. In each of these scenarios, each

Algorithm 2 Interference-aware destination VM selection algorithm.

Symbol:
V: Set of VMs $\{v_1, v_2, \ldots, v_k\}$
A: Application types $\{a_1, a_2, \ldots, a_t\}$
F: Interference feature matrix ($F[i,j]$ = suppression factor of a_j on a_i)
a_{new}: Type of new application
Input:
V, A, F, a_{new}
Output:
v_d: Selected target VM

```
function SELECT_VM(V, A, F, a_new)
    min_load ← min_{v∈V} |v.apps|                    ▷ Find minimum load
    V_candidate ← {v ∈ V | |v.apps| = min_load}
    if |V_candidate| = 1 then
        return V_candidate[0]
    end if
    max_gain ← −∞
    v_d ← null
    for each v in V_candidate do
        gain ← 0
        for each app a in v.apps do
            gain ← gain + F[a_new, a]                ▷ Accumulate suppression
        end for
        if gain > max_gain then
            max_gain ← gain
            v_d ← v
        end if
    end for
    return v_d
end function
```

scheduling node hosts 5 VMs having the same configuration. The settings for each server and VM remain the same as those in Section 3.2

5.1.1. Validation and comparison of performance interference measurement models

In previous studies, most studies only validate the model's goodness of fit, such as by examining the R^2 value, or evaluate prediction accuracy under specific scenarios without investigating potential overfitting. Our comparison work, as described in Section 3.3, primarily contrasts the models based on their goodness of fit, accuracy, and reliability. In addition to the model we propose in Eq. (2), we also compare a model that omits the metric of the temporal differences of the same performance metrics between co-located VMs, as shown in Eq. (12):

$$I(A\&X) = b_0 + b_1.x_1 + b_2.x_2 + b_3.x_3 + b_4.x_4 + b_5.x_5, \tag{12}$$

where A and X represent two co-located VMs, x_1 to x_5 represent the mean values of CPU utilization, LLC miss rate, cycles, branch misprediction rate, and TLB miss rate for VM X when running in isolation, and b_0 to b_5 are coefficients.

In addition to comparing Eq. (2) and Eq. (12), to highlight the advantages of our model over those proposed in existing research, we also analyze and compare the following models with Eq. (2):

E_s: Liu et al. analyzed the use of LLC miss rate to quantify performance interference and developed a linear regression model based on LLC miss rate [36].

Mvei: Sun et al. proposed a model based on the LLC miss rate of VMs running in isolation, the current CPU utilization of the PM, the vCPU utilization of each VM deployed on the PM, and the number of VMs [55].

LRA: A regression model for quantifying interference based on LLC miss rate and CPU utilization [56].

5.1.2. Scenarios and comparisons of scheduling algorithms

Execution Scenarios of Scheduling Algorithms.. Since our model incorporates performance counters such as LLC miss rate and branch misprediction rate, which are difficult to simulate, it is only in a real-world environment that the effectiveness of the interference-aware scheduling algorithm can truly be demonstrated. We implemented the control end to receive and record the load information of servers and VMs sent by the scheduling end and to make decisions based on the scheduling algorithm. On the scheduling nodes, servers periodically collect performance metrics and calculate the DTW distance against prior data, while VMs detect idle states and log the information of the running applications. The control and scheduling nodes communicate via the HTTP protocol. Before executing the scheduling algorithm, initial applications are evenly distributed across the VMs based on their number and type, and a list of applications to be scheduled is generated. These applications are scheduled at fixed intervals. Specifically, in this scenario, we aim to schedule 252 applications. Initially, each VM runs one application, and a new application arrives every 40 seconds. The algorithm determines which server and which VM the new application should be allocated to. The reason for this setup is that, once the entire allocation scheme is completed, each VM will be loaded with exactly one application to five applications. If further allocation occurs, some VMs will run more than five applications. This setup enables the algorithm to make allocation decisions under varying load conditions and facilitates a comprehensive evaluation of the overall performance.

Comparison of Scheduling Algorithms.. To demonstrate the feasibility of our algorithm, we also conducted experiments in scenarios with 6 and 7 scheduling nodes, while keeping the total number of scheduled applications constant. Although increasing the number of nodes can alleviate congestion during application allocation which might otherwise diminish the value of the analysis, our actual testing revealed that shortening the allocation interval by just 5 seconds significantly increased application distribution congestion. Therefore, we reduced the scheduling intervals to 35 seconds and 30 seconds respectively.

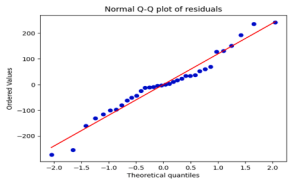
To validate our proposed scheduling algorithm, TS-DTW (Interference-Aware Scheduling based on Time Series Data DTW Distance), we also implemented the following algorithms for comparison:

1. MFF (Interference-Aware Scheduling Algorithm based on Mean Fitting Function): This algorithm determines the allocation plan based on the output of a linear formula fitted with the average values of each indicator, without considering the temporal fluctuations of the indicators or the differences in interference characteristics of different application combinations during co location operation [11].
2. NOL (Scheduling Algorithm based on the Number of Loads): The essence of this algorithm is to balance the load across cloud data center servers, without considering performance interference [45].
3. Round-bin Scheduling Algorithm (Round-bin Scheduling Algorithm): Based on the round in scheduling method, applications are scheduled to run on servers and their VMs in rotation according to certain rules, without considering performance interference [57].
4. RA (Random Algorithm): This algorithm randomly decides the allocation of applications to servers and VMs.

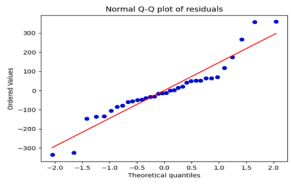
5.2. Experimental results

5.2.1. Validation and comparison of performance interference measurement models

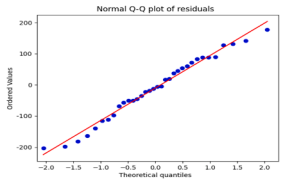
Verify the effectiveness of measuring temporal differences.. In Section 3.1, we proposed a performance interference model based on five performance metrics. Compared to existing research, our model additionally incorporates a metric quantifying the temporal differences of co-located VMs for the same performance indicator. To verify the effect of adding temporal difference measurement, we compared the models in Eq. (2)



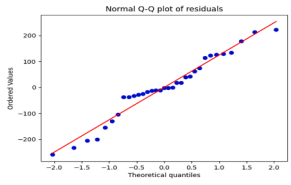
(a) TDM of 607.cactuBSSN_s



(b) NTDM of 607.cactuBSSN_s



(c) TDM of 638.imagick_s



(d) NTDM of 638.imagick_s

Fig. 5. Figure 5(a) and 5(b) are the QQ plots for the performance interference modeling of the VM with the 607.cactuBSSN_s load, using the model which considering temporal differences **(TDM)** in Eq. (2) and which excluding temporal differences **(NTDM)** in Eq. (12), respectively. Figure 5(c) and 5(d) are those of the VM with the 638.imagick_s load, using the model in Eq. (2) and that in Eq. (12), respectively.

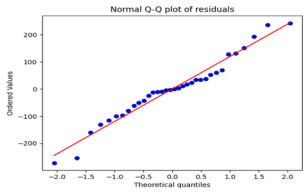
Table 9
R^2 values of the models in measuring performance interference for the VM with the load of 607.cactuBSSN_s or 638. imagick_s.

R^2	Eq. (2)	E_s	Mvei	LRA
607.cactuBSSN_s	0.85	0.53	0.73	0.69
638.imagick_s	0.88	0.55	0.76	0.65

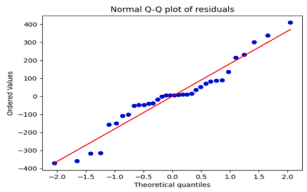
and Eq. (12). We configured the workloads of the VMs under test as 607.cactuBSSN_s and 638.imagick_2 respectively, co-locating with the VM running different workloads. We then used the two models to measure their performance interference. When quantifying the performance interference for the VM running 607.cactuBSSN_s workload, the model fit, measured by R^2, was 0.85 for Eq. (2) and 0.81 for Eq. (12). Similarly, for the VM running 638.imagick_s workload, the R^2 values were 0.88 for Eq. (2) and 0.84 for Eq. (12). The QQ plots, used to assess model accuracy and reliability, are shown in Fig. 5. It can be observed that regardless of whether the workload on the VM was 607.cactuBSSN_s or 638.imagick_s, the model in Eq. (2) demonstrated higher accuracy and reliability than the model in Eq. (12). This result indicates that incorporating the temporal difference metric improves modeling performance.

Compared with existing compute-intensive models.. We compare our model Eq. (2) with E_s, Mvei, and LRA. These models, unlike ours, do not incorporate the time-series difference metric for the same performance indicators. Specifically, E_s measures performance interference based on LLC miss rate. Mvei extends E_s by adding the current host CPU utilization and the sum of CPU utilizations of all co-located VMs; it models the relationship between host CPU utilization and performance interference using an exponential function. LRA, conversely, omits three of the metrics included in our model–metrics proposed in Section 3.1 that are rarely found in other performance interference models for compute-intensive VMs. Through experiments, we compared the goodness-of-fit metric, R^2, of these models in measuring the performance interference of the VM with the load of 607.cactuBSSN_s or 638.imagick_s. The results are shown in Table 9. It is evident that our proposed model achieves a better fit than the other models under comparison.

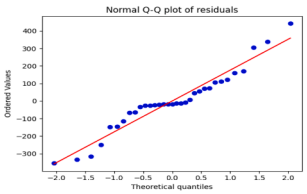
However, a high fit does not guarantee high model reliability or accuracy. To evaluate potential overfitting, we compared the models using QQ plots, as shown in Figs. 6 and 7. The trends in the QQ plots clearly



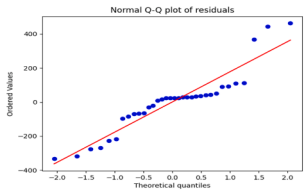
(a) Eq. (2) of 607.cactuBSSN_s



(b) E_s of 607.cactuBSSN_s

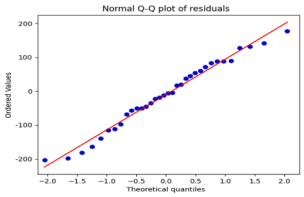


(c) Mvei of 607.cactuBSSN_s

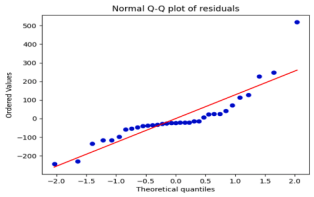


(d) LRA of 607.cactuBSSN_s

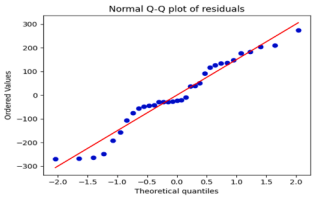
Fig. 6. The QQ plots of different forms of performance interference models for the VM with the load of 607.cactuBSSN_s, corresponding to Eq. (2), E_s, Mvei, and LRA, respectively.



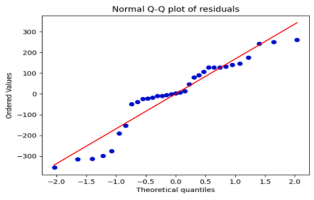
(a) Eq. (2) of 638.imagick_s



(b) E_s of 638.imagick_s



(c) Mvei of 638.imagick_s



(d) LRA of 638.imagick_s

Fig. 7. The QQ plots of different forms of performance interference models for the VM with the load of 638.imagick_s, corresponding to Eq. (2), E_s, Mvei, and LRA, respectively.

indicate that our model (Figs. 6(a) and 7(a)) exhibits significantly better reliability and accuracy than the other compared models. Further analysis reveals that, besides the benefits of considering temporal differences, utilizing multiple metrics substantially enhances the model. If the model relies on a single metric, there is a higher risk of misrepresentation when quantifying interference for different co-located applications with varying interference characteristics. Although Mvei includes more metrics, it fundamentally models based on the CPU utilization metric, so a multi-metric approach is preferred for a more comprehensive capture of interference features.

5.2.2. Comparison of scheduling algorithms

Comparison of throughput.. For each algorithm listed in Section 5.1.2, we executed 15 runs. We recorded the time taken from the placement of the initial applications to the completion of the last application (makespan), and calculated the average makespan per algorithm. Algorithm performance was compared using throughput, defined as the total number of applications executed divided by the makespan (measured in

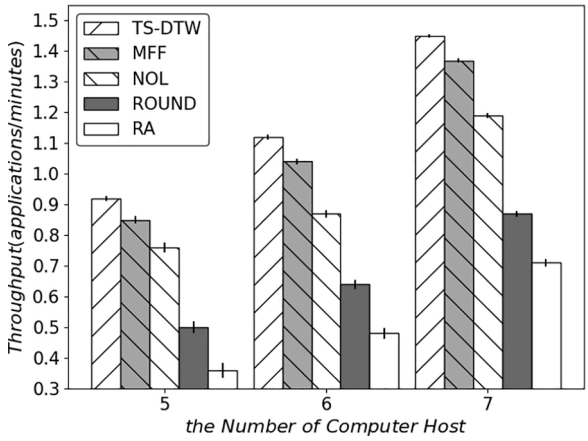


Fig. 8. Throughput comparison of different algorithms under each host counts.

minutes), as shown in Eq. (13):

$$Throughput = \frac{Number\ of\ applications}{Makespan}. \tag{13}$$

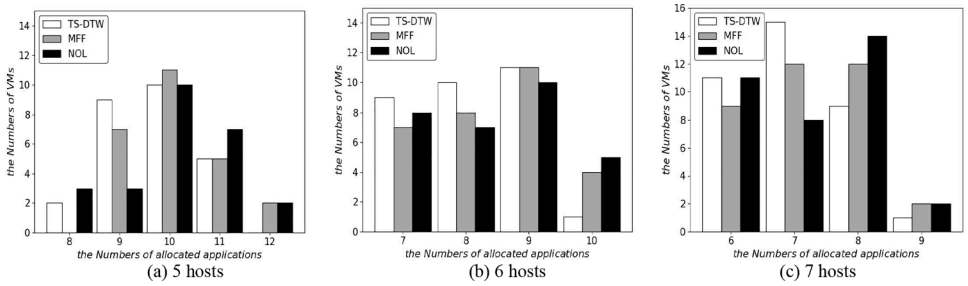
Under the experimental setup described in Section 5.1.2, scheduling 252 applications onto 5 hosts, the throughput comparison is shown in Fig. 8. The height of the bar chart represents the mean after 15 runs, and the vertical line on each bar is used to show the degree of dispersion of these experimental results. We can easily observe that TS-DTW performed the best, followed by MFF in second place, NOL in third place, and round-bin in fourth place. When the number of scheduling hosts increased to 6 or 7, the throughput of all algorithms increased, but their relative ranking remained unchanged. MFF's slightly lower performance is attributed to its use of average metric values to assess the current state. However, dynamic load fluctuations in the runtime environment cause state changes, meaning that each assessment may not always be accurate. Regarding the NOL algorithm, its scheduling decisions primarily rely on the number of applications on servers and VMs, without accounting for the specific interference characteristics of different application co-location combinations. As demonstrated in the third paragraph of Section 4.2, the actual runtime of applications varies significantly under different co-location combinations. Failure to fully consider this factor can result in poor co-location combinations leading to longer execution times, explaining NOL's lower throughput compared to TS-DTW and MFF. As for the round-bin algorithm, it simply allocates applications in a fixed sequence without dynamic adjustment based on current conditions, resulting in the lowest performance among the four algorithms.

Comparison of application allocation in VMs.. After comparing the throughput, we analyzed the application allocation for the other three algorithms shown in Fig. 9, except for the random algorithm and round-bin algorithm, under different numbers of scheduling end hosts. All 15 experiments obtained the same results, proving that the performance of these three algorithms is relatively stable. With five scheduling hosts (totalling 25 VMs), TS-DTW and MFF allocated the largest number of VMs running 9–10 applications. The distinction is that TS-DTW allocated more VMs to 9 applications than MFF did, whereas the reverse was observed for VMs with 10 applications. Additionally, under TS-DW, some VMs were allocated 8 applications, in contrast to MFF, where some hosted 12. Given that TS-DTW outperformed MFF in throughput, the key difference between these algorithms is their consideration (or lack thereof) of time-series data fluctuations. The NOL algorithm, which targets a balanced number of applications per VM, exhibited a different distribution: some VMs received only 8 applications, while VMs with 11 or 12 applications were relatively numerous, and the count of VMs with 9 applications was lower than under the other two algorithms. Therefore, NOL performed worse than TS-DTW and MFF because it prioritizes load balancing while neglecting the performance-interference characteristics of specific co-location combinations; this omission led to substantially increased actual execution times and an accumulation of co-located applications. When the number of scheduling hosts increased to 6 or 7 (30 and 35 VMs, respectively), TS-DTW allocated fewer applications per VM than the other algorithms, while MFF again demonstrated superior performance over NOL. Combining throughput results with allocation patterns indicates that algorithms ignoring specific co-location combinations perform worse than those that account for them, and that algorithms that neglect time-series fluctuations are less effective than those that incorporate them.

Comparison of server overload situations.. Following the analysis of VM application allocation, we analyzed server load conditions. Previous experiments showed that application completion times vary greatly under different co-location combinations. If applications consistently experience long execution times, the real-time count of accumulated applications running on VMs will increase, leading to more severe resource constraints and performance interference on servers, worsening load conditions. To evaluate load under different host counts, we defined a threshold. When the real-time number of applications on a host exceeds this threshold, the server load is considered critical. This threshold was defined as the minimum value among the highest real-time application counts observed on each compute host. Once the real-time application count on a compute host exceeds this threshold, the host is considered saturated. We counted the results obtained from 15 experiments and presented the median in Fig. 10. For the 5-host and 6-host scenarios, based on collected data, we set the threshold at 25. The resulting saturation states are shown in Figs. 10(a) and 10(b), respectively. From the throughput analysis, we know that while the total number of applications is constant, shortening the allocation interval with more hosts alleviates assignment congestion, increasing throughput. Consequently, saturation under 6 hosts was significantly lower than under 5 hosts. However, the round-bin algorithm still reached its peak real-time application count more frequently than NOL, NOL exceeded MFF, and MFF reached higher values than TS-DTW. When the number of hosts increased to 7, the saturation threshold was set to 23 based on collected data, with saturation states shown in Fig. 10(c). The saturation ranking of the algorithms remained unchanged. Combining the first two comparative experiments, it is evident that considering co-location combination differences and timing difference factors influences server load conditions and consequently impacts the final scheduling effectiveness.

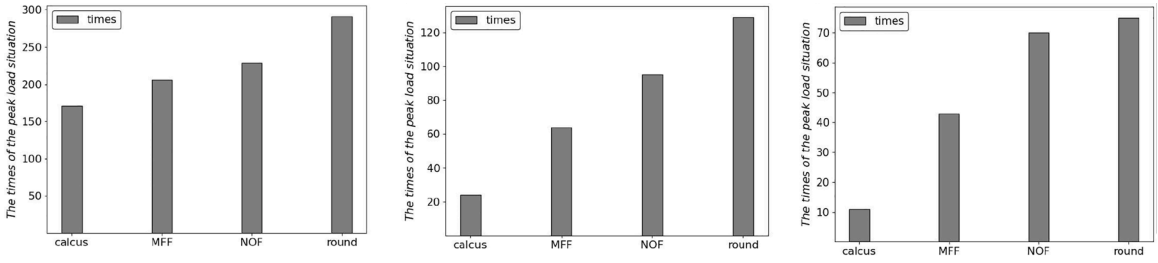
5.3. Algorithm overhead analysis

In this section, we analyze the time complexity of Algorithm 1 and Algorithm 2 to understand their computational overhead and scalability. First, we analyze Algorithm 1. Let n denote the number of servers, m denote the number of historical datasets per server, and T denote the length of the time series. Calculating the Dynamic Time Warping (DTW) distance between two time series of length T has a complexity of $O(T^2)$. For each server, computing the DTW distance against its m historical datasets incurs a cost of $O(m \cdot T^2)$. Consequently, for n servers, the overall time complexity is $O(n \cdot m \cdot T^2)$. In practical scenarios, this overhead can be mitigated through several optimization strategies, such as limiting m to only the most relevant or recent historical data, managing T by sampling or aggregating the time series data and parallelizing the distance computations across servers. Next, we analyze Algorithm 2. Let k denote the number of VMs per server. This algorithm involves simple comparisons and selections based on precomputed application counts, resulting in a time complexity of $O(k)$. In practice, since k is typically small (e.g., 5 in our experiments), this overhead is negligible. The high efficiency of Algorithm 2 makes it suitable for real-time scheduling decisions. Overall, the DTW distance computation in Algorithm 1 constitutes



(a) 5 hosts (b) 6 hosts (c) 7 hosts

Fig. 9. Application allocation on VMs of different scheduling algorithms under each host counts.



(a) 5 hosts (b) 6 hosts (c) 7 hosts

Fig. 10. The times of the servers reaches the peak load situation of different algorithms under each host counts.

the primary computational bottleneck. However, its overhead can be effectively managed by limiting the number of historical datasets, employing approximate DTW algorithms, or precomputing distances. Given that scheduling decisions are made periodically (e.g., every 40 seconds in our experiments), the computational overhead remains acceptable in most cloud environments. This analysis confirms that the proposed solution is not only effective but also practically feasible for real-world deployment.

6. Conclusion

We proposed a performance interference measurement model for compute-intensive VM, which incorporates three additional performance metrics compared to existing studies and takes account into the temporal differences of co-located VMs on the same metrics. We conducted an analysis of the model's fitting accuracy and verified its reliability in terms of overfitting, with results showing that our model outperforms several existing models. Additionally, we proposed a scheduling algorithm that considers for the temporal fluctuations of performance metrics and interference characteristics of different co-location application combinations. The results demonstrate that our algorithm outperforms scheduling methods based solely on load balancing or the average value of performance metric. This study focuses primarily on scenarios involving computational resource consumption, and we believe that our research approach could also be applied to performance interference studies involving other resource types. Furthermore, there are areas for improvement in our research. For instance, more complex models such as neural networks could be considered for performance interference modeling, accompanied by a finer-grained analysis of resource-specific performance (e.g., memory) in co-location scenarios. Especially for compute-intensive VMs that possess their own memory requirements, such an analysis would be particularly valuable. In terms of interference-aware scheduling, we could explore the design of a scheduling algorithm based on time-series data that does not rely on prior knowledge. Moreover, it would be valuable to investigate scheduling algorithms for batch applications with dependencies, in which tasks are not independent.

CRediT authorship contribution statement

Chennian Xiong: Writing – original draft, Methodology; **Weiwei Lin:** Writing – review & editing, Supervision; **Huikang Huang:** Conceptualization; **Jianpeng Lin:** Conceptualization; **Keqin Li:** Supervision.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by Shandong Provincial Natural Science Foundation Project (ZR2024LZH012), Guangdong Provincial Natural Science Foundation Project (2025A15150 10113) and the Major Key Project of PCL, China under Grant PCL2025A08, PCL2025A11.

References

[1] C. Slingerland, 90 + Cloud Computing Statistics: A 2025 Market Snapshot, 2025, Accessed: 2025-07-05, https://www.cloudzero.com/blog/cloud-computing-statistics.
[2] W. Lin, C. Xiong, W. Wu, F. Shi, K. Li, M. Xu, Performance interference of virtual machines: a survey, ACM Comput. Surv. 55 (12) (2023) 1–37.
[3] A.K. Maji, S. Mitra, S. Bagchi, Interference-aware configuration for virtualized web servers, ACM SIGMETRICS Performance Eval. Rev. 42 (1) (2014) 573–574. https://doi.org/10.1145/2663165.2663330
[4] M.M. Alves, L. Teylo, Y. Frota, L.M.A. Drummond, An interference-aware virtual machine placement strategy for high performance computing applications in clouds, in: 2018 Symposium on High Performance Computing Systems (WSCAD), IEEE, 2018, pp. 94–100.
[5] M. Alam, R.A. Haidri, D.K. Yadav, Efficient task scheduling on virtual machine in cloud computing environment, Int. J. Pervas. Comput. Commun. 17 (3) (2021) 271–287.
[6] D.K. Shukla, D. Kumar, D.S. Kushwaha, An efficient tasks scheduling algorithm for batch processing heterogeneous cloud environment, Int. J. Adv. Intell. Parad. 23 (1–2) (2022) 203–216.

[7] E. Bakshi, Performance Interference Detection for Cloud-Native Applications Using Unsupervised Machine Learning Models, Master's thesis, California Polytechnic State University, 2024.
[8] H. Labs, HPC on Cloud — Cost Optimisation & Cloud Economics, 2024, Accessed: 2025-07-05, https://medium.com/hmxlabs/hpc-on-cloud-cost-optimisation-cloud-economics-2c82d259b0d4.
[9] H. Xu, S. Song, Z. Mao, Characterizing the performance of emerging deep learning, graph, and high performance computing workloads under interference, in: 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2024, pp. 468–477.
[10] A. Cascajo, D.E. Singh, J. Carretero, Detecting interference between applications and improving the scheduling using malleable application clones, Int. J. High Perform. Comput. Appl. 38 (2) (2024) 108–133.
[11] T. Zhang, D. Ou, Z. Ge, C. Jiang, C. Cérin, L. Yan, Interference-aware workload scheduling in co-located data centers, in: IFIP International Conference on Network and Parallel Computing, Springer, 2022, pp. 69–82.
[12] L. Pons, J. Feliu, J. Sahuquillo, M.E. Gómez, S. Petit, J. Pons, C. Huang, Cloud white: detecting and estimating qos degradation of latency-critical workloads in the public cloud, Future Gener. Comput. Syst. 138 (2023) 13–25.
[13] M. Ruaro, H. Barral, M. Bertolino, R. Cataldo, R. Medina, M. Karaoui, E. Borde, The last-Level-Cache interference in guest performance: a case-Study with zephyr OS, in: 2023 26Th Euromicro Conference on Digital System Design (DSD), IEEE, 2023, pp. 351–358.
[14] T. Shi, Y. Yang, Y. Cheng, X. Gao, Z. Fang, Y. Yang, Alioth: A Machine Learning Based Interference-Aware Performance Monitor for Multi-Tenancy Applications in Public Cloud, arXiv:2307.08949 (2023).
[15] J. Yao, J. Li, X. Mo, W. Wu, Modeling memory bandwidth interference in cloud data centers via deep learning, in: 2024 9Th International Conference on Computer and Communication Systems (ICCCS), IEEE, 2024, pp. 441–447.
[16] A. Baluta, J. Mukherjee, M. Litoiu, Machine learning based interference modelling in cloud-native applications, in: Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, 2022, pp. 125–132.
[17] L. Zhao, Y. Yang, Y. Li, X. Zhou, K. Li, Understanding, predicting and scheduling serverless workloads under partial interference, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15.
[18] A. Tzenetopoulos, D. Masouros, S. Xydis, D. Soudris, Orchestration extensions for interference-and heterogeneity-Aware placement for data-Analytics, Int. J. Parallel. Program. (2024) 1–26.
[19] S. Shah, Y. Amannejad, D. Krishnamurthy, Diaspore: diagnosing performance interference in apache spark, IEEE Access 9 (2021) 103230–103243.
[20] C.K. Swain, A. Sahu, Interference aware workload scheduling for latency sensitive tasks in cloud environment, Computing 104 (4) (2022) 925–950.
[21] Y. Zhang, B. Tang, J. Luo, J. Zhang, Deadline-aware dynamic task scheduling in edge–cloud collaborative computing, Electr. (Basel) 11 (15) (2022) 2464.
[22] R. Shaw, E. Howley, E. Barrett, An intelligent ensemble learning approach for energy efficient and interference aware dynamic virtual machine consolidation, Simul. Modell. Pract. Theory 102 (2020) 101992.
[23] R. Andreoli, T. Cucinotta, D.B. De Oliveira, Priority-driven differentiated performance for nosql database-as-a-service, IEEE Trans. Cloud Comput. (2023).
[24] S.-H. Huh, N. Ham, J.-H. Kim, J.-J. Kim, Quantitative impact analysis of priority policy applied to BIM-based design validation, Autom. Constr. 154 (2023) 105031.
[25] Y.-G. Yim, H.-J. Jang, H.-W. Jin, QoS For best-effort batch jobs in container-based cloud, Concurr. Computat.: Practice Exper. 35 (15) (2023) e6422.
[26] M. Ghorbian, M. Ghobaei-Arani, L. Esmaeili, A survey on the scheduling mechanisms in serverless computing: a taxonomy, challenges, and trends, Cluster Comput. 27 (5) (2024) 5571–5610.
[27] M. Ghorbian, M. Ghobaei-Arani, Function offloading approaches in serverless computing: a survey, Comput. Electr. Eng. 120 (2024) 109832.
[28] M. Tari, M. Ghobaei-Arani, J. Pouramini, M. Ghorbian, Auto-scaling mechanisms in serverless computing: a comprehensive review, Comput. Sci. Rev. 53 (2024) 100650.
[29] F. Strati, X. Ma, A. Klimovic, Orion: interference-aware, fine-grained GPU sharing for ML applications, in: Proceedings of the Nineteenth European Conference on Computer Systems, 2024, pp. 1075–1092.
[30] B.P. Muller, L.J. Wong, A.J. Michaels, Sensitivity analysis of RFML applications, IEEE Access (2024).
[31] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, C. Pu, An analysis of performance interference effects in virtual environments, in: 2007 IEEE International Symposium on Performance Analysis of Systems & Software, IEEE, 2007, pp. 200–209.
[32] B. Lu, J. Fang, X. Hong, J. Shi, Energy-efficient task scheduling for mobile edge computing with virtual machine i/o interference, Future Gener. Comput. Syst. 148 (2023) 538–549.
[33] S.-M. Tseng, B. Nicolae, F. Cappello, A. Chandramowlishwaran, Demystifying asynchronous i/o interference in hpc applications, Int. J. High. Perform. Comput. Appl. 35 (4) (2021) 391–412.
[34] A. Tzenetopoulos, D. Masouros, S. Xydis, D. Soudris, Interference-aware workload placement for improving latency distribution of converged HPC/big data cloud infrastructures, in: International Conference on Embedded Computer Systems, Springer, 2021, pp. 108–123.
[35] X. Bu, J. Rao, C.-z. Xu, Interference and locality-aware task scheduling for mapreduce applications in virtual clusters, in: Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing, 2013, pp. 227–238.
[36] Y. Liu, X. Deng, J. Zhou, M. Chen, 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.
[37] M.G. Xavier, C.H.C. Cano, V. Meyer, C.A.F. De Rose, Intp: quantifying cross-application interference via system-level instrumentation, in: 2022 IEEE 34Th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE, 2022, pp. 231–240.
[38] T. Jin, Z. Cai, B. Li, C. Zheng, G. Jiang, J. Cheng, Improving resource utilization by timely fine-grained scheduling, in: Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–16.
[39] A.H. El-Gamal, R.R. Mostafa, N.A. Hikal, Load balancing enhanced technique for static task scheduling in cloud computing environments, in: Internet of Things–Applications and Future: Proceedings of ITAF 2019, Springer, 2020, pp. 411–430.
[40] N. Malik, M. Sardaraz, M. Tahir, B. Shah, G. Ali, F. Moreira, Energy-efficient load balancing algorithm for workflow scheduling in cloud data centers using queuing and thresholds, Appl. Sci. 11 (13) (2021) 5849.
[41] V.R. Anu, S. Elizabeth, IALM: Interference aware live migration strategy for virtual machines in cloud data centres, in: Data Management, Analytics and Innovation: Proceedings of ICDMAI 2018, Volume 2, Springer, 2019, pp. 499–511.
[42] L.C. Jersak, T. Ferreto, Performance-aware server consolidation with adjustable interference levels, in: Proceedings of the 31St Annual ACM Symposium on Applied Computing, 2016, pp. 420–425.
[43] S.M. Nabavinejad, M. Goudarzi, Data locality and VM interference aware mitigation of data skew in hadoop leveraging modern portfolio theory, in: Proceedings of the 33Rd Annual ACM Symposium on Applied Computing, 2018, pp. 175–182.
[44] S. Wang, W. Zhang, T. Wang, C. Ye, T. Huang, Vmon: monitoring and quantifying virtual machine interference via hardware performance counter, in: 2015 IEEE 39Th Annual Computer Software and Applications Conference, 2, IEEE, 2015, pp. 399–408.
[45] D.V. Manjula, V.N.T. Sravanthi, M.K. S. L.M. Priyanka, A LOAD BALANCING ALGORITHM FOR THE DATA CENTRES TO OPTIMIZE CLOUD COMPUTING, J. Sci. Technol. (JST) 8 (4) (2023) 72–79.
[46] N. Schmitt, J. Bucek, J. Beckett, A. Cragin, K.-D. Lange, S. Kounev, Performance, power, and energy-efficiency impact analysis of compiler optimizations on the spec cpu 2017 benchmark suite, in: 2020 IEEE/ACM 13Th International Conference on Utility and Cloud Computing (UCC), IEEE, 2020, pp. 292–301.
[47] A. Pokharana, R. Gupta, Using sysbench, analyze the performance of various guest virtual machines on a virtual box hypervisor, in: 2023 2Nd International Conference for Innovation in Technology (INOCON), IEEE, 2023, pp. 1–5.
[48] Dynamic time warping, https://en.wikipedia.org/wiki/Dynamic_time_warping.
[49] Goodness-of-Fit, https://www.investopedia.com/terms/g/goodness-of-fit.asp.
[50] Q–Q plot, https://en.wikipedia.org/wiki/Q%E2%80%93Q_plot.
[51] Y. Wang, L. Zhao, From theory to visualization: deriving and simulating catacaustic curves, in: Fourth International Conference on Applied Mathematics, Modelling, and Intelligent Computing (CAMMIC 2024), 13219, SPIE, 2024, pp. 54–60.
[52] S. Moll, S. Hack, Partial control-flow linearization, ACM SIGPLAN Notices 53 (4) (2018) 543–556.
[53] M. He, F. Liu, S.W.S. Do, Heterogeneous hyperthreading, in: 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2024, pp. 68–78.
[54] Microsoft, Azure Batch technical overview, 2023, Accessed: 2025-07-11, https://learn.microsoft.com/en-us/azure/batch/batch-technical-overview.
[55] X. Sun, Q. Wu, Y. Tan, F. Wu, Mvei: an interference prediction model for cpu-intensive application in cloud environment, in: 2014 13Th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, IEEE, 2014, pp. 83–87.
[56] L. Pons Escat, Interference analysis and resource management in server processors: from HPC to cloud computing, Master's thesis, Universitat Politècnica de València, València, Spain, 2023.
[57] Z. Zhou, Y. Si, Y. Cheng, An efficient container bin-packing method for large scale cloud-native clusters, in: 2023 IEEE 9Th International Conference on Cloud Computing and Intelligent Systems (CCIS), IEEE, 2023, pp. 500–504.

Chennian Xiong received the B.S. degree in computer science at South China University of Technology in 2017. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China. His research interests include cloud computing, performance interference modeling and interference-aware scheduling.

Weiwei Lin received his B.S. and M.S. degrees from Nanchang University in 2001 and 2004, respectively, and his Ph.D. in Computer Application from South China University of Technology in 2007. He has been a visiting scholar at Clemson University from 2016 to 2017. Currently, he is a professor in the School of Computer Science and Engineering at South China University of Technology. His research interests include distributed systems, cloud computing, and AI application technologies. He has published more than 150 papers in refereed journals and conference proceedings. He has been a reviewer for many international journals, including IEEE TPDS, TSC, TCC, TC, TCYB, etc. He is a distinguished member of CCF and a senior member of the IEEE.

Huikang Huang received the M.S. degree in Computer Science and Theory from the South China Agricultural University, Guangzhou, China, in 2021. Now, he is a Ph.D. candidate in the School of Computer Science and Engineering, South China University of Technology. His research interests mainly focus on cloud computing.

Jianpeng Lin received the M.S. degree in computer science at Guangdong University of Technology in 2019. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China. His research interests include cloud computing, data center thermal management and sustainable computing.

Keqin Li is a SUNY Distinguished Professor of Computer Science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication. He has authored or coauthored over 870 journal articles, book chapters, and refereed conference papers. He is among the world's top 5 most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He is currently an associate editor of the ACM Computing Surveys and the CCF Transactions on High Performance Computing. He has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing. He is an IEEE Fellow and an AAIA Fellow. He is also a Member of Academia Europaea (Academician of the Academy of Europe).