RT-VirtIO: Towards the Real-time Performance of VirtIO in a Two-tier Computing Architecture

Siwei Ye[1], Minqing Sun[1], Huifeng Zhu[2], Yier Jin[3], An Zou[1]
[1]Shanghai Jiao Tong University, [2]Washington University in St. Louis,
[3]University of Science and Technology of China

***Abstract*—With the popularity of virtualization technology, ensuring reliable I/O operations with timing constraints in virtual environments becomes increasingly critical. Timing-predictable virtual I/O enhances the responsiveness and efficiency of virtualized systems, facilitating their seamless integration into time-critical applications such as industrial automation and robotics. Its significance lies in meeting rigorous performance standards, minimizing latency, and consistently delivering predictable I/O performance. As a result, virtual machines can effectively support mission-critical and time-sensitive workloads. However, due to the complicated system architecture, the I/O operations in virtualization face competition from tasks within the same virtual machine and those in different virtual machines who share the same host machine. This study presents RT-VirtIO, a practical approach to provide predictable real-time I/O operations. RT-VirtIO addresses the challenges associated with lengthy data paths and complex resource management. Through early-stage characterization, this study identifies key factors contributing to poor I/O real-time performance and then builds an analytical model and a learning-based data-driven model to predict the tail I/O latency. Leveraging these two models, RT-VirtIO effectively captures these dynamics, enabling the development of a general and applicable optimization framework. Experimental results demonstrate that RT-VirtIO significantly improves real-time performance in virtual environments (by $20.07\% \sim 30.90\%$) without necessitating hardware modifications, which exhibit promising applicability across a broader range of scenarios.**

***Index Terms*—I/O Scheduling, predictable I/O, real-time performance**

I. Introduction

In recent years, virtualization technology has advanced to enable different domains to safely run applications with varying levels of criticality, potentially under different operating systems, sharing the same hardware platform. This evolution has resulted in increased resource utilization, reduced hardware size, and lower costs [1]–[4]. Leveraging these capabilities, virtualization has gained widespread adoption in automotive and industrial applications. A key benefit is its ability to support multiple operating systems (OS), such as Linux [5], [6]. For example, in an intelligent car, one OS may handle complex tasks like navigation and speech recognition, while an OS manages safety-critical functions such as emergency braking. To ensure that these safety-critical systems meet stringent timing requirements, I/O virtualization techniques must provide predictable mechanisms for I/O operations [2], [3], [6], [7].

However, I/O virtualization involves complex access paths (i.e., indirection and interposition of privileged instructions) and complicated shared I/O resource management (i.e., scheduling and prioritization), resulting in decreased I/O performance, increased software overhead, and poor scalability. That is, predictability and timing accuracy are difficult to achieve with I/O virtualization [2], [6]. Our study (Sec. II) presents that the tail latency (i.e., 99.99th-percentile I/O latency) of a VM increases up to factor $126\times$ when I/O contention happens in a consolidated virtualization environment. Previous works try to enhance the real-time performance of virtual I/O from the point of partitioning hypervisor, scheduling algorithms, and hardware-assisted mechanisms [1], [7]–[12]. However, these solutions may involve runtime overhead [13]–[15] or require specific hardware features [16]–[18], which limits their compatibility with different systems, applications, and platforms.

Complementary to previous approaches, this work develops analytical and learning-based models to predict tail I/O latency of guest VM requests under various workloads, VM counts and execution conditions, and then proposes RT-VirtIO: a framework that optimizes the scheduler configurations and I/O resource allocations to meet the tasks' timing constraints. The key contributions of this article are:

- A comprehensive analysis and characterization are performed to identify the dominating factors that contribute to the tail I/O latency in VMs.
- An analytical model and a learning-based data-driven model are developed to capture the dynamics for the tail I/O latency considering dominating factors.
- A framework to improve the real-time performance of virtual I/O is presented, which optimizes the scheduler parameters and allocates the VM's I/O resources following the specific target latency, without modification of hardware nor dependency on specific features.
- Extensive experiments demonstrate that the proposed RT-VirtIO can significantly improve the predictable timing performance of virtual IO operations by $20.07\% \sim 30.90\%$ across different scenarios.

II. Background and Related Works

A. Virtualized I/O Operations

I/O operations in virtualization involve complex access paths and complicated shared I/O resource management [2]. The popular VM apps (like QEMU and KVM) use `virtio` [19] as their main platform for IO virtualization [20], [21]. The workflow of an I/O request in such a virtualization system is shown in Fig. 1. If an application invokes an I/O request from a guest VM, then this I/O request will first go through the VM's

An Zou is the corresponding author. This work is supported by NSFC 62202287.

978-3-9826741-0-0/DATE25/© 2025 EDAA

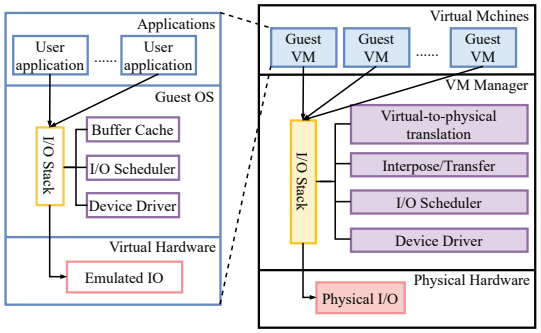
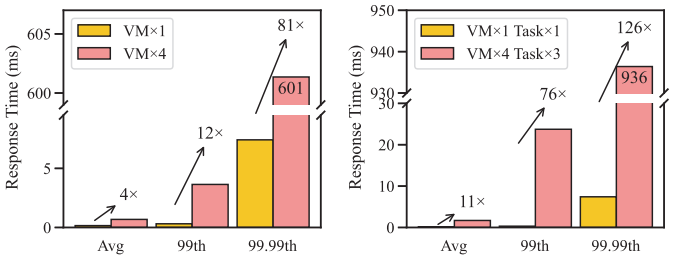


Fig. 1: Flow of I/O request in virtualization system.



(a) Multiple VMs Inference (b) Multiple Tasks Inference

Fig. 2: Effect of IO contention on I/O latency.

own I/O stack, where the front-end driver passes the request to `virtio-blk` kernel module, i.e. the emulated IO. The request is later transmitted to the back-end drivers in the host's Virtual Machine Monitor (VMM), then the host OS's I/O stack, and finally the Physical IO device.

Each part in the flow of an I/O request may enlarge the I/O latency, i.e., the time difference between the launch and finish of an I/O request. To make the problem even worse, although the isolation nature of virtualization techniques prevents direct I/O operation interferences between VMs, the I/O latency in each VM can still be affected when the overall I/O workload is heavy for the host platform.

B. Tail I/O Latency

The tail (99.99th percentile) I/O latency is vulnerable to time-critical applications, especially when multiple VMs and tasks are contending with each other for the host's I/O resources. To verify the effect of I/O contention on I/O latency, we run 1∼3 `fio` workloads on a single and 4 VMs. The `fio` workload launches read I/O requests with random order using POSIX synchronous mode. As shown in Fig. 2, when 4 VMs are running heavy workloads at the same time, the average latency of random read increases by $4 \sim 11\times$ compared to the single VM condition. However, the growth of the 99.99th percentile latency can be as large as $81 \sim 126\times$.

In a real-time system, the issue of large tail latency is critical because it may lead to deadline violations, increased jitter, resource starvation, and inefficient scheduling [22]–[24]. The causes of high tail latency include hardware, kernel, and application-level factors. This work focuses on the latter two factors and these factors will be discussed in detail in Sec. III.

C. Related Works

The previous solutions that improve I/O latency or its predictability in a virtualized environment can be categorized into software-based approaches, hardware-based approaches, or their combination [1], [7]–[12].

Software-based solutions focus on resource management and task scheduling [7], [25]–[27]. Representative approaches involve priority-based preemption on VM [13], workload-aware credit scheduler enhancement for Xen [28], scheduling algorithms that minimize I/O performance interference [15], partitioning hypervisor that isolates processor cores and non-real-time I/O devices [5], and broker-based real-time communication framework [14]. Hardware-assisted solutions enhance I/O virtualization by enabling more direct access to hardware. SR-IOV [16] and Intel VT-d [17] optimize PCIe device management and allow direct device assignment, reducing latency and overhead. Jiang et al. [2], [4], [29]–[31] integrated hardware-assisted I/O management into the BlueVisor hypervisor [3]. Muench et al. [18], [32] leveraged PCIe SR-IOV, IOMMU, and IOMPU to support I/O virtualization in mixed-criticality real-time systems.

Most prior research has focused on partitioning hypervisors, scheduling mechanisms, or hardware-assisted techniques, which often introduce runtime overhead [13]–[15] or rely on specific hardware features [16]–[18], limiting their compatibility. In contrast, this work provides a general framework for improving virtual I/O real-time performance without requiring specialized hardware or scheduling designs.

III. Characterization of Tail I/O Latency

The section presents the characterization of the tail I/O latency, especially at kernel and application level.

A. Filesystem (JBD2 & cache)

In Linux, the ext4 filesystem uses journaling to prevent metadata inconsistencies during system crashes [33]. JBD2 (Journaling Block Device 2) handles this by logging changes to the filesystem before they are written to storage. By default, JBD2 commits every 5 seconds, generating I/O that disrupts ongoing operations, leading to periodic I/O latency. To minimize this, we disable write barriers by using the '`barrier=0`' mount option. While this may compromise data integrity during power loss or crashes, it allows for better analysis of factors affecting tail I/O latency in this study.

The page cache stores file data in memory after it's read from the disk. If a VM requests data already in the host's page cache, it can be delivered without disk access, reducing I/O latency. However, a full page cache during testing can result in false low latency, masking the true virtual I/O bottleneck. To ensure accurate measurement of virtual I/O performance, especially tail latency, the host's cache is cleared before experiments.

B. IO Scheduler

1) none: In Linux, the default I/O scheduler, `none`, is a multi-queue no-op scheduler that processes requests in FIFO order without reordering or prioritization [34]. This is suitable for SSDs, as it reduces CPU overhead and minimizes latency.

2) mq-deadline: The scheduler adapts the deadline I/O scheduler for multiqueue devices [34]. I/O requests are placed in both a sorted queue (by Logical Block Addressing) and a FIFO deadline queue (by expiration time). Separate queues exist for read and write requests, with read requests prioritized.

Requests are issued from the sorted queue unless a FIFO request expires [34].

3) bfq: The `bfq` scheduler [35] in Linux ensures high throughput, low latency, and fairness by allocating each process a budget of I/O operations, making it effective in multi-application environments. It also supports fine-grained I/O resource allocation using cgroups, distributing bandwidth based on cgroup weights. Experiments show `bfq` reduces latency by 2.22 times compared to cfq and at least 4.41 times compared to other schedulers while maintaining high throughput [35].

Given these features, `bfq` is used in this work to prevent VM monopolization of disk resources and assign different I/O priorities through cgroup weights.

IV. Analytical Modeling

A. Model Definition

This work presents an analytical model to predict tail I/O latency based on the principle of I/O workload divided by I/O throughput, and the mathematical formula considers various influencing factors. The model includes n Guest VMs $\mathcal{G} = G_1, ..., G_n$ running on a Host $\mathcal{H}$, all using the `bfq` scheduler. Each guest G_i runs a set of tasks Γ^{G_i}, where each task $\tau_j \in \Gamma^{G_i}$ issues I/O requests R_j^k with size L_j^k KB. The total size of requests from τ_j during $[t_1, t_2)$ is $A^{\tau_j}(t_1, t_2)$. The host's aggregate throughput $T_{agg}^{\mathcal{H}}$ (KB/sec) is partitioned among guests, with $T_{agg}^{G_i}$ allocated to each G_i. Task-specific throughput $T_{agg}^{\tau_j}$ follows similar notations, with definitions omitted for brevity.

Upon launching a guest G_i, the host assigns it a cgroup weight ϕ^{G_i}, with the `bfq` scheduler allocating I/O throughput proportional to this weight. To facilitate calculation, we use normalized weight $\phi_{norm}^{G_i} = \frac{\phi^{G_i}}{\phi Total}$, where $\sum_{i=1}^{n} \phi_{norm}^{G_i} \leq 1$. For simplicity, ϕ is used to represent ϕ_{norm} in the rest of the paper. Each guest has a budget B^{G_i} (KB). The host `bfq` scheduler selects G_i to access the disk until it exhausts its backlog or budget. Budget adjustments follow rules from [35], with B_{max} as the maximum budget. Key `bfq` parameters, `timeout_sync` and `slice_idle`, affect tail I/O latency from guest G_i (host $\mathcal{H}$) side, and they are denoted as $\mathcal{TS}^{G_i(\mathcal{H})}$ and $\mathcal{SI}^{G_i(\mathcal{H})}$. Table I summarizes the symbols used in this paper.

The analytical model predicts $\mathcal{L}(\tau_j, (t_1, t_2))$, the tail I/O latency of task τ_j in guest G_i within time interval $[t_1, t_2)$, given the workload, guest VM numbers, priority, and scheduler configurations. Without losing generality, we set the time interval to be 1 second ($t_2 - t_1 = 1\ sec$), and simplify the notation into $\mathcal{L}(\tau_j)$. Here $\mathcal{L}(\tau_j)$ refers to the tail latency of synchronous I/O read requests, the key real-time performance bottleneck for most applications. Based on literature [21], [35] and experimental profiling, $\mathcal{L}(\tau_j)$ is divided into guest portion and host portion, i.e.,

$$\mathcal{L}(\tau_j) = \mathcal{L}^{G_i}(\tau_j) + \mathcal{L}^{\mathcal{H}}(\tau_j) \tag{1}$$

B. Effect of `bfq` scheduler parameters

Prior to presenting the detailed tail I/O latency formula, it is essential to define the impact of the `bfq` scheduler parameters.

Symbol	Description	Value (unit)
$\tau_j \in \Gamma^{G_i}$	the j-th task in G_i	N/A
$R^{\tau_j,k}$	k-th request issued by τ_j	N/A
$L^{\tau_j,k}$	Size of $R^{\tau_j,k}$	variable (KB)
$T_{agg}^{\mathcal{H}}$	Aggregate throughput of host $\mathcal{H}$	245 (MB/s)
$T_{agg}^{G_i(\tau_j)}$	Aggregate throughput of guest G_i (task τ_j)	variable (MB/s)
$B^{G_i(\tau_j)}$	Budget assigned to G_i (task τ_j)	variable (KB)
B_{max}	Maximum budget that can be assigned to a task	8192 (KB)
$\mathcal{TS}^{\mathcal{H}(G_i)}$	`timeout_sync` for `bfq` in host $\mathcal{H}$ (guest G_i)	input (ms)
$\mathcal{SI}^{\mathcal{H}(G_i)}$	`slice_idle` for `bfq` in host $\mathcal{H}$ (guest G_i)	input (ms)
$\phi^{G_i(\tau_j)}$	Weight for guest G_i (task τ_j)	input
$L_{min}^{\tau_j,k}$	Minimum size of $R^{\tau_j,k}$	input (KB)
$A^{G_i(\tau_j)}$	Total size of the requests issued by $G_i(\tau_j)$ in 1 second	input (KB)
$\mathcal{L}(\tau_j)$	Tail I/O latency for task τ_j	output (ms)
λ^{ϕ}	Influence coefficient for ϕ	parameter
$\lambda^{\mathcal{TS}^{\mathcal{H}(G_i)}}$	Influence coefficient for $\mathcal{TS}^{\mathcal{H}(G_i)}$	parameter
$\lambda^{\mathcal{SI}^{\mathcal{H}(G_i)}}$	Influence coefficient for $\mathcal{SI}^{\mathcal{H}(G_i)}$	parameter

TABLE I: Lists of symbols.

1) `timeout_sync`: This parameter specifically controls the maximum amount of time synchronous requests that are allowed to be delayed in the scheduler before they are served. By setting a shorter `timeout_sync`, the system can reduce latency for synchronous I/O by ensuring these requests are given priority and processed faster. For the `bfq` scheduler in host $\mathcal{H}$ (guest G_i), the effect can be formulated as:

$$\mathcal{E}(\mathcal{TS}^{\mathcal{H}(G_i)}) = \lambda^{\mathcal{TS}^{\mathcal{H}(G_i)}} \times (\mathcal{TS}^{\mathcal{H}(G_i)}/1000)$$

2) `slice_idle`: This parameter sets the idle time before serving the next I/O request in a queue. Reducing `slice_idle` within a proper range can lead to lower I/O latency because the scheduler minimizes the wait time between processing requests, increasing the I/O throughput utilization rate. For the `bfq` scheduler in host $\mathcal{H}$ (guest G_i), the effect can be formulated as:

$$\mathcal{E}(\mathcal{SI}^{\mathcal{H}(G_i)}) = (1 - \lambda^{\mathcal{SI}^{\mathcal{H}(G_i)}} \times \frac{\mathcal{SI}^{\mathcal{H}(G_i)}}{\mathcal{SI}^{default}})$$

C. Tail I/O Latency (Host Portion)

For a task $\tau_j \in \Gamma^{G_i}$, the host portion of its tail I/O latency $\mathcal{L}^{\mathcal{H}}(\tau_j)$ can be expressed as:

$$\begin{aligned}\mathcal{L}^{\mathcal{H}}(\tau_j) = SW(|\Gamma^{\mathcal{H}}|) + \frac{A^{G_i} + B_{max}^{G_i} - L_{min}^{G_i}}{\mathcal{E}(\mathcal{SI}^{\mathcal{H}}) \times T_{agg}^{G_i}} \\ + \frac{\sum_{G_m \in \Gamma^{\mathcal{H}} \setminus G_i} A^{G_m} + B_{max}}{\mathcal{E}(\mathcal{SI}^{\mathcal{H}}) \times T_{agg}^{\mathcal{H}}} + \mathcal{E}(\mathcal{TS}^{\mathcal{H}}).\end{aligned} \tag{2}$$

The first term in Eq. (2) represents the time spent on host-level task switching, increasing with the size of $\Gamma^{\mathcal{H}}$ (task set in Host $\mathcal{H}$). The second term represents the worst-case latency for guest G_i to serve I/O requests, with the numerator as the request size and $B_{max}^{G_i} - L_{min}^{G_i}$ accounting for unprocessed requests. The denominator includes $T_{agg}^{G_i}$, the throughput allocated to G_i based on ϕ^{G_i}, as defined in Eq. (3). The third term includes the total size of I/O requests from other guests plus B_{max}, divided by $T_{agg}^{\mathcal{H}}$. As mentioned in section IV-B2, `slice_idle` has direct impact on I/O throughput, and thus both $T_{agg}^{G_i}$ and $T_{agg}^{\mathcal{H}}$ are adjusted by $\mathcal{E}(\mathcal{SI}^{\mathcal{H}})$. Lastly, $\mathcal{E}(\mathcal{TS}^{\mathcal{H}})$ is added due to the impact of `timeout_sync`.

$$T_{agg}^{G_i} = \lambda^{\phi} \times \phi^{G_i} \times T_{agg}^{\mathcal{H}}. \tag{3}$$

D. Tail I/O Latency (Guest Portion)

For a task $\tau_j \in \Gamma^{G_i}$, the guest portion of its tail I/O latency $\mathcal{L}^{G_i}(\tau_j)$ can be expressed as:

$$\mathcal{L}^{G_i}(\tau_j) = SW(|\Gamma^{G_i}|) + \frac{A^{\tau_j} + B^{\tau_j}_{max} - L^{\tau_j}_{min}}{\mathcal{E}(\mathcal{SI}^{G_i}) \times T^{\tau_j}_{agg}} + \frac{\sum_{\tau_n \in \Gamma^{G_i} \setminus \tau_j} A^{\tau_n} + B_{max}}{\mathcal{E}(\mathcal{SI}^{G_i}) \times T^{G_i}_{agg}} + \mathcal{E}(\mathcal{TS}^{G_i}). \quad (4)$$

Similar to Eq. (2), $T^{\tau_j}_{agg}$ is τ_j's throughput allocated by G_i according to ϕ^{τ_j}:

$$T^{\tau_j}_{agg} = \lambda^{\phi} \times \phi^{\tau_j} \times T^{G_i}_{agg}. \quad (5)$$

The other terms in Eq. (4) share similar definitions as those in host portion. The difference is that the size of I/O requests, budgets and `bfq` parameter effects are all with respect to tasks within guest G_i.

V. Learning Based Modeling

Besides the analytical mode, this work also builds a learning-based model to predict tail I/O latency utilizing the Feedforward Neural Network (FNN). The FNN model involves only unidirectional data propagation. FNN is suitable due to the lack of explicit temporal sequences in the data, making the use of a Recurrent Neural Network (RNN) unnecessary. The model's effectiveness depends on its architecture's suitability for the training purpose and input features. Within the FNN model category, Multilayer Perceptron (MLP) and Convolutional Neural Network (CNN) are common. MLP is widely applicable due to its fully connected layers, while CNN excels in image recognition with high-dimensional inputs. However, since our input is a one-dimensional array, using a CNN is not justified.

We use an MLP with fully connected layers, each with several neurons, and ReLU activation for non-linearity. The model reads input values (Table I) and predicts I/O latency, comparing it with actual values to calculate MSE (mean square error) loss during training. As a data-driven model, it minimizes loss through gradient descent, encapsulating the problem without requiring explicit functions or explanations for the prediction outcome.

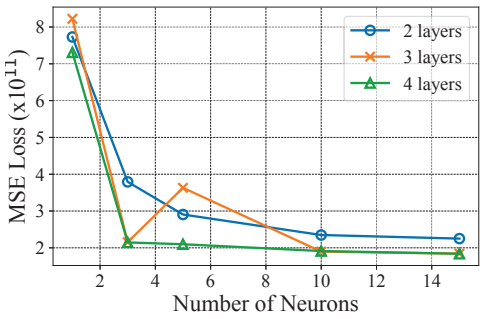


Fig. 3: MSE loss vs. model size.

To maximize model accuracy, the relationship between model size and loss is examined. The results, illustrated in Fig. 3, show that with over 5 neurons per layer, the loss stabilizes and performance converges. Beyond this point, increasing layers has minimal effect. Therefore, for evaluation, a network with 3 layers and 15 neurons per hidden layer is chosen to ensure optimal neural network performance.

VI. Enhance the Real-time Performance

We propose RT-VirtIO, a framework to enhance real-time performance using both analytical and learning-based (MLP) models. As shown in Fig. 4, RT-VirtIO first collects model parameters, I/O workload information, and target I/O latency. The parameters include λ^{ϕ}, $\lambda^{\mathcal{TS}^{\mathcal{H}(G_i)}}$, and $\lambda^{\mathcal{SI}^{\mathcal{H}(G_i)}}$ for the analytical model, or the architecture and learned parameters for the MLP model. The workload infomation includes the number of VMs, tasks, and IOPS, while the target latency specifies the maximum allowable I/O latency for each task.

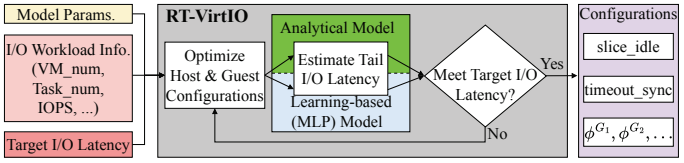


Fig. 4: RT-VirtIO Framework.

When the inputs are ready, RT-VirtIO launches optimization loops, which use local search method to optimize the host and guest configurations. On each iteration, the collected inputs and optimized configurations are thrown to either the analytical model or the MLP model, which predicts the tail I/O latency for each task. Afterward, RT-VirtIO checks whether the predicted values meet the target I/O latency. If met, the `slice_idle`, `timeout_sync` values for the host and guests, and the cgroup weight sets ($\phi^{G_1}, \phi^{G_2}, ...$) are exported to the system to begin the execution of tasks; if not met, the loop continues, and RT-VirtIO assigns larger cgroup weights (more I/O bandwidth) to the tasks. If the target I/O latency is so demanding that $\sum_{i=1,...,n} \phi^{G_i} > 1$, i.e., the total I/O bandwidth asked by VMs exceeds the amount that can be provided by the host, RT-VirtIO makes a compromise, ensuring the real-time performance of the most urgent VM and sacrificing the other VM(s).

VII. Evaluation

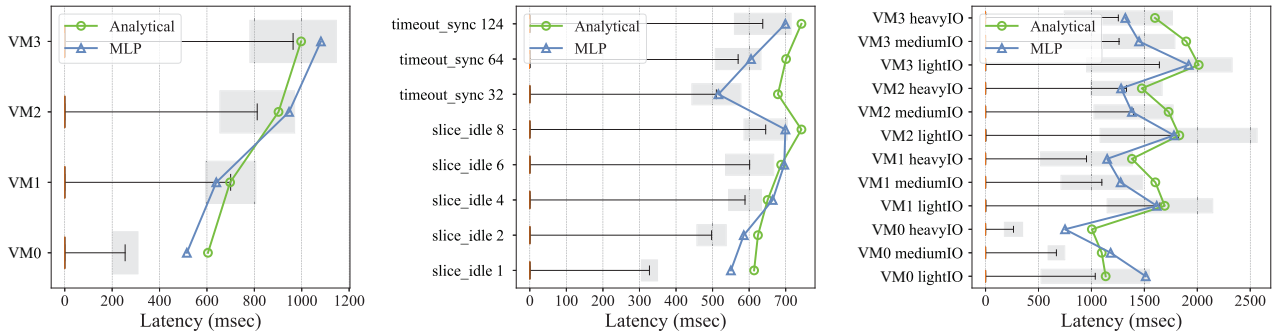
A. Modeling Training and Results

To train the fine-grained analytical and learning-based (MLP) models, we simulate scenarios where multiple guest VMs execute tasks with varying I/O workloads under different host (guest) `bfq` parameters and guest cgroup weights (ϕ). With optional values of settings listed in Table II, the tasks launch a series of 4KB read/write I/O requests via `fio`. The resulting 28,000 data sets include test conditions and corresponding tail I/O latency as inputs and reference outputs for training.

Test Condition	Optional Values
VM number (size of $\Gamma^{\mathcal{H}}$)	{1,2,3,4}
weight of VM (ϕ^{G_i})	{50,100,150,250,300,500,700}
Task number (size of Γ^{G_i})	{1,2,3}
IOPS of I/O task τ_j	{100,500,1500}
$\mathcal{SI}^{\mathcal{H}(G_i)}$	{1,2,4,6,8}
$\mathcal{TS}^{\mathcal{H}(G_i)}$	{32,64,124}

TABLE II: Training dataset test condition setup.

After optimizing with the training dataset, the analytical model obtains the optimum parameters listed in Table III. The values of $\lambda^{\mathcal{TS}^{G_i}}$ and $\lambda^{\mathcal{SI}^{G_i}}$ are close to 0, indicating that the guest `bfq` parameters `timeout_sync` and `slice_idle` have minimal impact on tail I/O latency. However, positive



(a) Different cgroup weight (b) Different `bfq` parameters (c) Different I/O workload

Fig. 5: Comparison of experimental tail I/O latency (box plots) and predictions (marked-line plots) by the trained models under different conditions.

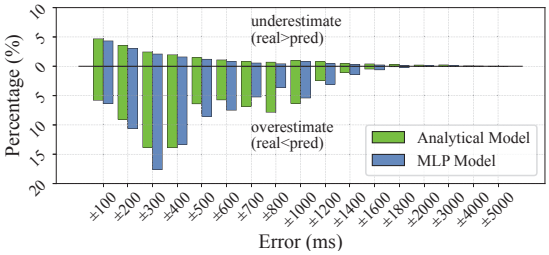


Fig. 6: Prediction error of I/O latency for both models.

values for $\lambda^{\mathcal{TS}^{\mathcal{H}(G_i)}}$ and $\lambda^{\mathcal{SI}^{\mathcal{H}(G_i)}}$ suggest that reducing these parameters on the host lowers tail I/O latency.

Parameter	λ^{ϕ}	$\lambda^{\mathcal{TS}^{G_i}}$	$\lambda^{\mathcal{SI}^{G_i}}$	$\lambda^{\mathcal{TS}^{\mathcal{H}}}$	$\lambda^{\mathcal{SI}^{\mathcal{H}}}$
value	0.36	1E-06	1E-06	0.7	0.5

TABLE III: Optimal parameters for the analytical model.

For the neural network model, the training dataset is split into 5 folds for cross-validation, rotating the validation set to reliably assess performance and reduce overfitting. After training, predictions from the 5 models are averaged to improve accuracy and robustness through model ensemble.

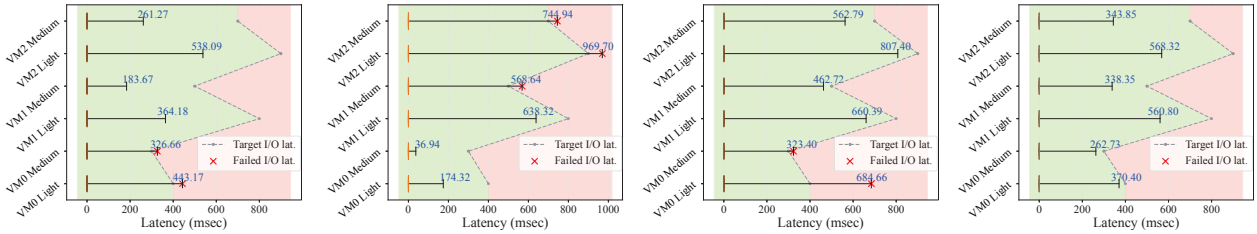
To evaluate model accuracy, we compare predicted tail I/O latency with experimental values under three typical conditions: (a) 4 VM 1 I/O task with different host `bfq` parameter values, (b). 4VM 1 I/O task with different VM cgroup weight, (c). 4VM 3 I/O task with light/medium/heavy I/O workload. The experimental values in this work are conducted on a commercial Linux PC with kernel version 5.14.0, Intel i7-10700 CPU, 32GB Memory, and Intel SSD 670p (1TB, 256 MB DDR3L cache, 3500 MB/s read and 2500 MB/s write maximum speed). The VMs are launched using QEMU (version 4.2.1). Figure 5 shows boxplots of experimental I/O latency, with the 99.99th percentile representing tail latency, averaged over 30+ tests per condition. Note that the minimum and quartiles are crowded into a single orange line because their values are so small ($< 0.5\ ms$) compared to the 99.99th percentile. The plots also include standard deviation (gray bars) to indicate latency variation. The blue and green curves in Fig. 5 demonstrate a good prediction quality for both analytical and neural network models. The analytical model's predictions are conservative, generally exceeding experimental values, while the neural network model predictions are closer but occasionally underestimate latency.

The overall prediction error for both models is displayed in Fig. 6. It shows that the analytical and the MLP models have similar error distribution. Both of the models have more negative error (overestimate) than positive error (underestimate), which is intended in the training process because underestimating the tail I/O latency makes RT-VirtIO assign inadequate I/O bandwidth to meet the guest VM's I/O latency requirement, which should be avoided. The majority of prediction error lies in $[-500, 0]\ ms$ region, which is acceptable considering the fact that tail I/O latency is around 1800 ms in a heavy workload environment and the standard deviation can be $\pm 900\ ms$ (e.g., the latency in Fig. 5c).

B. RT-VirtIO Performance

To evaluate RT-VirtIO's real-time performance enhancement, 24 benchmark tests were conducted, varying the number of VMs (1-4), tasks (1-3), and I/O workload (light: 100 IOPS, medium: 500 IOPS, heavy: 1500 IOPS). The other I/O request setup remains the same as in previous sections. The target tail I/O latency for each task was set based on task type and background workload, with more time-critical VMs (VM0>VM1>VM2>VM3) having smaller latency targets. To avoid imbalanced workloads, each VM launched the same I/O task or combinations (e.g., light with medium) across all benchmark test cases.

The comparison schemes include default bfq [35], ionice [36] and mimic RT-Xen [13]. The default bfq scheme uses `bfq`'s dynamic budget-fair rule without I/O priority or bandwidth assignment. The ionice scheme assigns different I/O scheduling classes (idle, best-effort, real-time) and priorities using `ionice`. An idle process only performs I/O when the disk is free, best-effort processes are scheduled by priority (0-7, with 0 being the highest), and real-time processes get constant disk access, potentially starving others. In benchmark tests, the VM with the most urgent latency requirement is given real-time priority, while others are assigned best-effort with priorities ordered by their latency targets. The mimic RT-Xen scheme is an adaption of RT-Xen [13], the widely recognized real-time hypervisor scheduling framework for VMs. In benchmark tests, we adapted RT-Xen's scheduling algorithm, originally designed for CPU allocation, to allocate I/O resources to VMs according to their latency targets.



(a) default bfq (b) ionice (c) RT-Xen (d) RT-VirtIO

Fig. 7: Example of pass/fail result by different schemes, with achieved tail I/O latency labeled on plots. (The target tail I/O latency for the tasks are are {400, 300, 800, 500, 900, 700 ms} from bottom to top.)

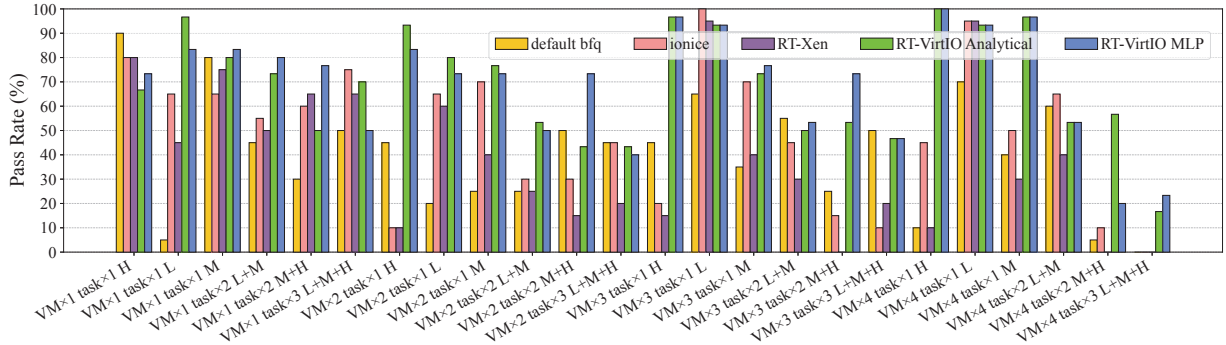


Fig. 8: Benchmark Pass Rate Comparison.

For a test case in our benchmark, a scheme is judged to pass a test run if all of its tasks' tail I/O latency is lower than the target value. Figure 7 presents an example of pass-fail evaluation, where the test cases considers 3 VMs running light and medium I/O tasks. The target tail I/O latency for this test case is labeled as a dashed broken line in each subfigure, where VM0 has the most urgent real-time requirement, then VM1, and lastly VM2. Intuitively, the green region stands for the pass region, and the red region stands for the fail region. For the default bfq scheme (Fig. 7a), three VMs have peer priority, which makes VM0 fail to meet its target. For ionice scheme (Fig. 7b), VM0 is set as a real-time process, which greatly reduces its tail I/O latency. However, the other two VMs suffer from starvation and fail to meet some of their tasks' latency requirements. For mimic RT-Xen (Fig. 7c), due to replenishment rules, the I/O resource is not fully utilized and thus VM0 fails its target. For RT-VirtIO, the I/O bandwidth is distributed properly with respect to each VM's target tail I/O latency. As shown in Fig. 7d, all of the tasks' tail I/O latency lie in the green region, which indicates the specific test run passes.

The evaluation conducts 20 repeated test runs (a test run lasts 100 minutes) for each of the 24 test cases. The pass rate comparison of the three schemes is presented in Fig. 8, where L, M, and H are the abbreviations of I/O task types (light, medium, and heavy). The result indicates that RT-VirtIO outperforms the default bfq, ionice and mimic RT-Xen scheme in 16 out of 24 test cases. Among the rest 8 test cases, 6 cases have almost equivalent pass rates (they differ by around 5%). Unsurprisingly, RT-VirtIO produces consistent real-time performance in the same test case, with either the analytical or the MLP model. The real-time performance privilege is especially prominent when the test case have multiple VMs and combined tasks. For example, in the case of $VM \times 3\ task \times 2\ M + H$, the default bfq, ionice and mimic RT-Xen scheme only have 25%, 15% and 0% pass rates, respectively, while RT-VirtIO raises the pass rate to over 50%. In the case of $VM \times 4\ task \times 3\ L + M + H$, the default bfq, ionice and mimic RT-Xen scheme fail all their test run, while RT-VirtIO remains the pass rate to be around 20%. The average pass rates over the 24 test cases are 40.42%, 48.96%, 38,54%, 69.03% and 69.44% respectively for default bfq, ionice, mimic RT-Xen, RT-VirtIO (analytical model) and RT-VirtIO (MLP model). Compared to the other schemes, RT-VirtIO increases the pass rate by $20.07\% \sim 30.90\%$.

VIII. Conclusion

This work aims at achieving predictable tail I/O latency in virtualized environments, crucial for multiple VMs and tasks competing for the I/O resources. We identify key factors affecting tail I/O latency and develop both analytical and learning-based models to predict these latencies accurately. Our proposed framework, RT-VirtIO, optimizes scheduler parameters and allocates I/O resources efficiently, without requiring hardware modifications. Extensive evaluations show that RT-VirtIO significantly enhances real-time I/O performance across various scenarios. In benchmark tests, RT-VirtIO outperforms default bfq, ionice and mimic RT-Xen schemes in most cases, achieving higher pass rates for meeting target latencies, especially in complex setups with multiple VMs and mixed I/O workloads. Compared to the comparison schemes, RT-VirtIO increases the pass rate by $20.07\% \sim 30.90\%$. In summary, RT-VirtIO provides a practical and effective solution for managing virtualized I/O resources, enhancing predictability and performance without complex software development or hardware feature support. Future work may refine the predictive models and extend the framework's applicability to dynamic resource management. In addition, the scenarios of multiple levels of timing criticality will be studied in the further work.

REFERENCES

[1] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, pp. 726–740, Oct. 2014.

[2] Z. Jiang, N. Audsley, and P. Dong, "BlueIO: A scalable real-time hardware i/o virtualization system for many-core embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 3, May 2019.

[3] Z. Jiang, N. C. Audsley, and P. Dong, "BlueVisor: A scalable real-time hardware hypervisor for many-core embedded systems," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, April 2018, pp. 75–84.

[4] Z. Jiang, "Real-time i/o system for many-core embedded systems," Ph.D. dissertation, University of York, August 2018.

[5] C.-F. Yang and Y. Shinjo, "Obtaining hard real-time performance and rich Linux features in a compounded real-time operating system by a partitioning hypervisor," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Lausanne Switzerland: ACM, Mar. 2020, pp. 59–72.

[6] D. Casini, A. Biondi, G. Cicero, and G. Buttazzo, "Latency analysis of i/o virtualization techniques in hypervisor-based real-time systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 306–319.

[7] I. Sañudo, R. Cavicchioli, N. Capodieci, P. Valente, and M. Bertogna, "A survey on shared disk i/o management in virtualized environments under real time constraints," vol. 15, no. 1, pp. 57–63, March 2018.

[8] S. Abedi, N. Gandhi, H. M. Demoulin, Y. Li, Y. Wu, and L. T. X. Phan, "RTNF: Predictable latency for network function virtualization," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, pp. 368–379.

[9] L. Abdallah, M. Jan, J. Ermont, and C. Fraboul, "Reducing the contention experienced by real-time core-to-i/o flows over a tilera-like network on chip," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 86–96.

[10] K. Singer, K. Agrawal, and L. I-Ting Angelina, "Scheduling i/o latency-hiding futures in task-parallel platforms." Society for Industrial and Applied Mathematics.

[11] G. Ara, L. Abeni, T. Cucinotta, and C. Vitucci, "On the use of kernel bypass mechanisms for high-performance inter-container communications," in *High Performance Computing*, vol. 11887, pp. 1–12.

[12] G. Ara, T. Cucinotta, L. Abeni, and C. Vitucci, "Comparative evaluation of kernel bypass mechanisms for high-performance inter-container communications:," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, pp. 44–55.

[13] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: towards real-time hypervisor scheduling in xen," in *Proceedings of the ninth ACM international conference on Embedded software*. ACM, Oct. 2011, pp. 39–48.

[14] G. Schwaricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A real-time virtio-based framework for predictable inter-VM communication," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, pp. 27–40.

[15] R. C. Chiang and H. H. Huang, "TRACON: interference-aware scheduling for data-intensive applications in virtualized environments," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2011.

[16] Y. Dong, Z. Yu, and G. Rose, "Sr-iov networking in xen: architecture, design and implementation," in *Proceedings of the First Conference on I/O Virtualization*, ser. WIOV'08. USA: USENIX Association, 2008.

[17] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel® virtualization technology for directed i/o," vol. 10, no. 3.

[18] D. Muench, O. Isfort, K. Mueller, M. Paulitsch, and A. Herkersdorf, "Hardware-based i/o virtualization for mixed criticality real-time systems using PCIe SR-IOV," in *2013 IEEE 16th International Conference on Computational Science and Engineering*. IEEE, pp. 706–713.

[19] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, Jul. 2008.

[20] QEMU, "Writing VirtIO backends for QEMU." [Online]. Available: https://www.qemu.org/docs/master/devel/virtio-backends.html

[21] KVM. Virtio block latency. [Online]. Available: https://www.linux-kvm.org/page/Virtio/Block/Latency

[22] R. Han, J. Wang, S. Huang, C. Shao, S. Zhan, J. Zhan, and J. L. Vázquez-Poletti, "Interference-aware component scheduling for reducing tail latency in cloud interactive services," *2015 IEEE 35th International Conference on Distributed Computing Systems*, pp. 744–745, 2015.

[23] P. A. Misra, M. F. Borge, Íñigo Goiri, A. Lebeck, W. Zwaenepoel, and R. Bianchini, "Managing tail latency in datacenter-scale file systems under production constraints," *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[24] E. Asyabi, S. SanaeeKohroudi, M. Sharifi, and A. Bestavros, "Terriertail: Mitigating tail latency of cloud virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 2346–2359, 2018.

[25] Y. Xu, T. He, R. Sun, Y. Ma, Y. Jin, and A. Zou, "Shape: Scheduling of fixed-priority tasks on heterogeneous architectures with multiple cpus and many pes," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022, pp. 1–9.

[26] A. Zou, J. Li, C. D. Gill, and X. Zhang, "RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1450–1465.

[27] J. Sun, J. Li, Z. Guo, A. Zou, X. Zhang, K. Agrawal, and S. Baruah, "Real-time scheduling upon a host-centric acceleration architecture with data offloading," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, pp. 56–69.

[28] H. Guan, R. Ma, and J. Li, "Workload-Aware Credit Scheduler for Improving Network I/O Performance in Virtualization Environment," *IEEE Transactions on Cloud Computing*, vol. 2, no. 2, Apr. 2014.

[29] Z. Jiang and N. C. Audsley, "Vcdc: The virtualized complicated device controller," in *Euromicro Conference on Real-Time Systems*, 2017.

[30] Z. Jiang, N. Audsley, P. Dong, N. Guan, X. Dai, and L. Wei, "MCS-IOV: Real-time i/o virtualization for mixed-criticality systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, pp. 326–338.

[31] Z. Jiang and N. C. Audsley, "GPIOCP: Timing-accurate general purpose i/o controller for many-core real-time systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017.

[32] D. Muench, M. Paulitsch, and A. Herkersdorf, "IOMPU: Spatial separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non-transparent bridges," in *2015 IEEE 17th International Conference on High Performance Computing and Communications*. IEEE, 2015.

[33] Journal (jbd2). [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html

[34] C. I. King. IOSchedulers. [Online]. Available: https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers

[35] P. Valente and F. Checconi, "High Throughput Disk Scheduling with Fair Bandwidth Distribution," *IEEE Transactions on Computers*, vol. 59, no. 9, pp. 1172–1186, Sep. 2010.

[36] ArchLinux. ionice(1) - arch manual pages. [Online]. Available: https://man.archlinux.org/man/ionice.1.en