FastIOV: Fast Startup of Passthrough Network I/O Virtualization for Secure Containers

Anonymous Author(s)
Submission Id: #97

Abstract

Single Root I/O Virtualization (SR-IOV) technology has advanced in recent years and can simultaneously satisfy the network requirements of high data plane performance, high deployment density and fast startup for applications in traditional containers. However, it falls short with secure containers, which have become the mainstream choice in multi-tenant clouds. The secure containers require passthrough I/O to enable SR-IOV, which hinders the container startup performance and prevents its usage in time-sensitive tasks like serverless computing. In this paper, we advocate that the startup performance of SR-IOV can be further boosted, making it suitable for building a container network interface (CNI) for secure containers. We first dissect the end-to-end concurrent startup process and identify three key bottlenecks that lead to the slow startup, including Virtual Function I/O (VFIO) device set management, Direct Memory Access (DMA) memory mapping and Virtual Function (VF) driver initialization. We then propose a CNI named FastIOV that addresses these bottlenecks through lock dissembling, unnecessary mapping skipping, decoupled zeroing, and asynchronous VF driver initialization. Our evaluation shows that FastIOV reduces the overhead of enabling SR-IOV in secure containers by 96.1%, achieving 65.7% and 75.4% reductions in the average and 99th percentile end-to-end startup time.

1 Introduction

Nowadays, mainstream cloud providers have been progressively shifting from virtual machines to containers as their new compute instances. The container-enabled cloud services, such as NoSQL database (*e.g.,* Azure Cosmos [12]) and serverless function compute (*e.g.,* AWS Lambda [4]), necessitate network access to either serve incoming requests or interact with other services like cloud storage. The container network is required to achieve not only high data plane performance but also high deployment density, *i.e.,* a large number of virtual network devices on a single server, and fast startup [2, 19, 21, 32, 45, 50, 54, 56, 57].

The hardware-assisted network device virtualization technology, Single Root I/O Virtualization (SR-IOV) [1], has emerged as the best performing approach to simultaneously satisfy the above three requirements for traditional containers. First, SR-IOV virtualizes a Network Interface Card (NIC) into multiple virtual NIC named Virtual Functions (VFs). It allows containers to directly interact with the NIC resources and achieve near bare-metal data plane performance, while other network solutions, like software based network, incur obvious overhead in throughput and latency [3, 13, 38, 46, 47]. Second, the deployment density of SR-IOV has also been greatly improved with the emerging technologies mdev [53] and scalable IOV [18]. The newest commercial NICs like Mellanox CX-7 [49] and Intel IPU [26] have announced the vanilla support of 1K VFs. Finally, the startup of SR-IOV for a traditional container is fast, as its main procedure is just moving a pre-created VF into the container's network namespace.

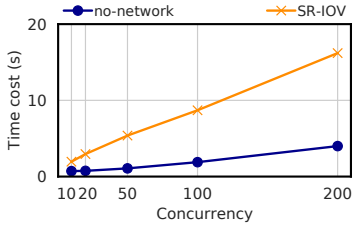


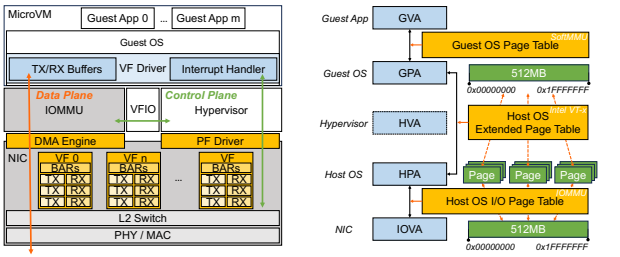
Figure 1. Overhead of enabling SR-IOV[1] on secure container startup time with concurrency up to 200. The concurrency setting is based on the statistics that over 200 container invocation requests arrive nearly simultaneously at an Alibaba serverless platform node [32].

However, despite the commendable performance of SR-IOV for traditional containers, it still falls short when applied to secure containers. Secure containers like Kata [21] and RunD [32] have nowadays become the mainstream choices in multi-tenant clouds where security is highly valued. They run the container processes inside micro Virtual Machines (microVMs) with trimmed and independent kernels to provide better isolation against attacks such as privilege escalation. Due to the existence of the independent kernels, an extra virtualization process named *passthrough I/O* is required for the VF to be used by the microVM. We find its overhead greatly hinders the startup performance of secure containers. Fig. 1 illustrates the effect of enabling SR-IOV on the average time of concurrently starting 10 ∼ 200 secure containers. We observe that enabling SR-IOV incurs a significant time overhead that increases with the concurrency. The time overhead is 12.2s when the concurrency is 200, increasing the average time by 305%. Such slow startup poses a significant obstacle for developing a desirable secure container network solution with SR-IOV.

[1]We are actually showing the performance of the optimized version of SR-IOV CNI that resolves an implementation flaw of driver rebinding in Kata, as described in §5. The original version [11] performs much worse.

In this work, we look into the problem of *achieving fast concurrent startup for SR-IOV in secure containers*. By breaking down the end-to-end concurrent startup procedure of SR-IOV enabled secure containers, we identify several key bottlenecks that have not been addressed before in low-density scenarios. Then, we propose FastIOV, an SR-IOV based network solution that tackles those bottlenecks and achieves ultra fast startup. Our contributions can be summarized as follows.

- **Measurement results (§3):** We dive into the details of the components, from the user-space CNI plugin and the container runtime, to the kernel-space device driver and OS modules. Three major bottlenecks related to passthrough I/O are identified: *VFIO device set (devset) management*, *DMA memory mapping* and *VF driver initialization*. These bottlenecks are not coupled with any specific CNI or secure container framework implementations. They contribute more than 70% and 80% of the average and 99th percentile container startup time, respectively. As far as we know, we are the first to thoroughly analyze and elaborate on the end-to-end startup process of SR-IOV enabled secure containers.
- **Optimization solutions (§4):** Targeting the key bottlenecks, FastIOV first disassembles the coarse-grained lock design in VFIO devset management by proposing a hierarchical lock framework, which parallelizes VFIO device operations while ensuring the consistency (§4.2.1). Second, we identify the causes of the inefficiency in DMA memory mapping as the mapping of unnecessary memory regions and memory zeroing overhead. FastIOV tracks and skips the unnecessary regions, and decouples memory zeroing from mapping to enable lazy zeroing (§4.3). Finally, FastIOV asynchronously executes VF driver initialization with container application launching, effectively masking the overhead (§4.2.2).
- **Implementation and performance gain (§6):** We implement FastIOV with a portable Linux kernel module, a CNI plugin, and other optimizations in the secure container framework and OS modules. We conduct extensive experiments and demonstrate that FastIOV reduces the time overhead of enabling SR-IOV by 96.1%, leading to 65.7% and 75.4% reductions in the average and 99th percentile container startup time compared with vanilla SR-IOV CNI [11]. We also evaluate FastIOV on four representative serverless applications and show that FastIOV reduces the average and the 99th percentile task completion time by 12.1%-53.5% and 20.3%-53.7%, respectively.
- **Community contribution:** We will open source the whole implementation of FastIOV as well as the benchmarking tools and dataset to benefit the community.



(a) SR-IOV in passthrough I/O. (b) Address spaces and translations.

Figure 2. SR-IOV and passthrough I/O architecture.

2 Background

2.1 SR-IOV and Passthrough I/O

Fig. 2(a) shows the architecture of SR-IOV with passthrough I/O. One SR-IOV NIC has a least one Physical Function (PF) bound to the host OS through the PF driver, which manages the overall resources of the NIC. The goal of SR-IOV is to divide the NIC resources, such as registers and TX/RX queues, into multiple isolated sets and generate multiple virtual NICs, referred to as Virtual Functions (VFs). While traditional containers can directly use the VFs as normal PCIe devices, secure containers require the further processing of passthrough I/O to use the VFs.

On the data plane of the passthrough I/O, the data transmission of each VF bypasses the host network stack and the hypervisor, which shortens the I/O path and reduces CPU overhead, leading to lower latency and higher throughput. Such bypassing is achieved by the Direct Memory Access (DMA) engine in the NIC. DMA utilizes the hardware-assisted memory mapping module, *i.e.,* Input/Output Memory Management Unit (IOMMU), to translate memory addresses and directly move packets between VF's TX/RX queues and microVM's TX/RX buffers.

In contrast, the control plane still remains under the management of the hypervisor. The hypervisor first creates and configures the VFs through PF drivers. When a VF is assigned to the guest, *i.e.,* microVM, it is bound to and managed by a Linux driver named Virtual Function I/O (VFIO). The hypervisor interacts with the VFIO driver to configure the corresponding memory mapping to the IOMMU module. After the initialization is completed, the guest can directly interact with the device in subsequent data transmission, and only interrupt signals are relayed through the hypervisor.

2.2 Address Spaces and DMA Memory Mapping

Fig. 2(b) shows the memory address spaces of the SR-IOV device, the host and the guest, in the context of passthrough I/O. We use the packet receiving process via a VF as an example to show how these address spaces are translated: (i) The guest OS notifies the DMA engine in the NIC to write the received packet to an I/O Virtual Address (IOVA). The IOVA is often of the same value as the Guest Physical Address (GPA) where the guest OS intends to store the received packets. (ii)

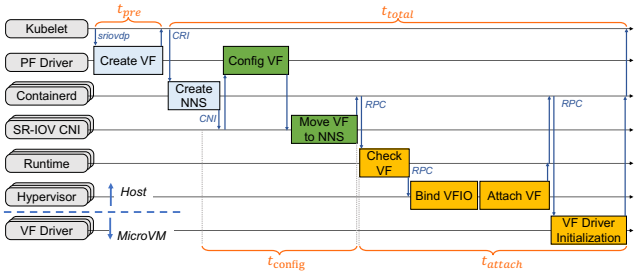


Figure 3. End-to-End startup procedure of SR-IOV CNI.

The DMA engine refers to the IOMMU hardware to translate the IOVA to the corresponding Host Physical Address (HPA) and performs actual packet writes to the physical pages. The translation is implemented by looking up a table in IOMMU, *i.e.,* I/O Page Table, which is maintained independently for each guest. As mentioned in §2.1, the table entries are configured by the VFIO driver when the VF is assigned to the guest, and the configuration process is referred to as *DMA memory mapping*. (iii) Upon completing the packet writes, the DMA engine notifies the guest OS that the data is ready by an interrupt relayed through the hypervisor. (iv) The guest OS retrieves the packet from the GPA that maps to the HPA. The mapping from GPA to HPA is implemented through a hardware table named Extended Page Table (EPT).

2.3 Startup Procedure of SR-IOV CNI

To provide a better understanding of the startup procedure of SR-IOV enabled secure container, we investigate the source code of several widely deployed community projects including the container orchestrator (Kubernetes/K8s [16]), container engine and runtime (Containerd [15], Kata [21]), CNI plugins (SR-IOV CNI [11] and sriovdp [10]), hypervisors (Kata-QEMU [21, 22] and KVM [34]) and Linux kernel [35], and summarize it in Fig. 3.

Before the K8s agent, *i.e.,* Kubelet, is informed to invoke multiple secure containers concurrently, it asynchronously calls the PF driver to create enough VFs. This pre-operation time t_{pre} is often large because it involves the hardware reconfiguration of the SR-IOV enabled NIC. Since the VF creation needs only to be done once after the booting of the host OS, we ignore this operation and exclude t_{pre} from the total time t_{total} in the rest of the paper. The container engine, *i.e.,* Containerd, is responsible for the life-cycle management of the containers. It first creates the isolated Network NameSpace (NNS) for each container and then successively calls the CNI plugin and the container runtime for VF configuration. The CNI plugin calls the PF driver to set up VF parameters like VLAN ID and rate limit, and then moves the VF to the container NNS (cf. t_{config} in Fig. 3). The container runtime checks the existence of the VF in the NNS and assigns it to the microVM (cf. t_{attach} in Fig. 3). The assigning process first binds the VF to the VFIO driver. Then the VFIO driver attaches the VF to the microVM by setting up the passthrough I/O as introduced in §2.1 and emulating the VF as a PCIe device. Like VF creation, the binding operation is a one-time task after booting the host server and its time cost can be ignored. Finally, the network driver of the VF inside the microVM, *i.e.,* VF driver, initializes and sets up the device as a Linux network interface.

It should be noted that the underlying logic of configuring a VF and attaching it to the microVM is in fact the same as that of enabling SR-IOV for a normal VM. However, compared with the normal VM use case, container applications have higher-volume invocation requirement and shorter lifespan, leading to higher requirement as well as new bottlenecks in startup time. This calls for further bottleneck identification and motivates our design for FastIOV.

3 Measurement and Motivation

3.1 Testbed for Startup Performance Measurement

Hardware setup. Our testbed uses servers that mirror the configurations used by leading cloud service providers' production environments. The specification includes: (i) CPU: Two NUMA-capable Intel Xeon Gold 6348 sockets running at 2.60 GHz, each housing 28 cores complemented by 80KB/1280 KB/42MB L1/L2/L3 Caches and with hyper-threading activated. (ii) Memory: 256GB DDR4 with 3200MHz clock frequency. (iii) NIC: A 25 GbE Intel E810 NIC that supports creating 256 VFs. Note that we also test with another NIC, 200 GbE Intel Mount Evans E2100 and observe similar results.

Software setup. The servers run CentOS 7 with Linux kernel v6.4.0. We choose the widely deployed container engine Containerd v1.7.3 [15], secure container runtime Kata Containers v3.2.0 [21] and SR-IOV CNI plugin v0.3 [11]. Kata Containers tailor the QEMU v6.2.0 hypervisor into a lightweight version named Kata-QEMU [21]. The kernel of the microVM is generated from Linux kernel v5.19.2, and the image is generated from Ubuntu 20.04. For each secure container, we allocate 0.5 vCPU and 512MB RAM through the configuration of Kata-QEMU and allocate one SR-IOV VF as its virtual NIC through the configuration of Containerd.

Measurement methodology. In the tests of startup time, we use *crictl* command to create the microVM without any container applications inside, as enabling SR-IOV only affects the startup process of the microVM. When we evaluate the performance of FastIOV on serverless applications in §6.6, we will report the task completion time, *i.e.,* the duration between the issuance of the startup command and the completion of the container application. To break down the timeline of the startup process, we develop a logging tool and integrate it into the above software components like Kata-QEMU and Linux kernel to collect finer-grained information. We ensure that the logging operations are asynchronous and our tests show that they incur nearly no additional overhead in startup time.

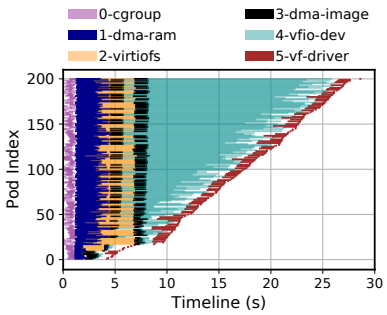


Figure 4. Breakdown of time-consuming steps. 200 SR-IOV enabled secure containers are launched concurrently.

3.2 Bottleneck Identification for SR-IOV CNI

3.2.1 Measurement result. We break down the timeline of concurrently starting 200 secure containers and show the most time-consuming steps in Fig. 4 and the corresponding statistics in Tab. 1. In the figure, (i) each horizontal line group stands for the timeline breakdown of each secure container; (ii) the segmented lines with different colors in each horizontal line group represent the different time-consuming steps. First, we observe that *4-vfio-devset*, *i.e.,* the opening of the VF from its VFIO device set, dominates the total time consumption and possibly experiences severe serialized operations. Second, other SR-IOV VF-related steps like DMA memory mapping (*1-dma-ram* and *3-dma-image*) and the initialization of network interface by the VF driver inside the microVM (*5-vf-driver*) also incur obvious overhead. The other two steps, *i.e., 0-cgroup* and *2-virtioFS*, refer to the process of setting up cgroups and shared file system for the secure container, and they are not related to the enabling of SR-IOV. Statistics in the table show that the VF-related steps take up 70.1% in the average startup time. The proportion increases to 80.8% when considering the long-tail latency with the 99th percentile. These statistics reveal a large room for accelerating the startup process. Next, we will analyze the root causes of the observed major bottlenecks before introducing our solutions in FastIOV.

3.2.2 Bottleneck 1: VFIO devset management. In the VFIO driver, a devset is used to manage a group of VFIO devices and control their reset behavior. When a device is bound to the VFIO driver, the VFIO driver first checks whether the device is attached to the PCI root bus or has the slot level reset capability. If neither, it means the reset of the device has to be performed at the bus level, *i.e.,* all devices attached to the same bus are reset together, and those devices are put into the same devset group. The main purpose of the devset is to ensure that when one VFIO device is being reset, all other VFIO devices affected are ready for reset as well. This requires that first, the VFIO driver scans the PCI bus to check if all devices on the bus are maintained in the VFIO devset group to ensure that no affected device is managed by other drivers. Second, the VFIO driver checks the total open count, *i.e.,* the number of processes or threads that are currently keeping the device open, of the devices in the devset group to ensure that no affected device is currently being used.

Step	Proportion in Average Time (%)	Proportion in 99th Percentile Time (%)
0-cgroup	2.9	2.3
1-dma-ram	**13.0**	**11.1**
2-virtiofs	13.3	13.6
3-dma-image	**5.6**	**4.3**
4-vfio-devset	**48.1**	**59.0**
5-vf-driver	**3.4**	**4.1**
Total (1, 3, 4, 5)	70.1	80.8

Table 1. Time proportions of time-consuming steps. The VF-related steps (1, 3, 4, 5) take up more than 70% and 80% of the average and 99th percentile startup time.

When attaching a VF to the microVM, one of the key steps is to register it in the hypervisor. During registration, the hypervisor opens the VF through the VFIO driver, and obtains the file descriptor and other relevant device information. The opening of the VF increases its open count, and further affects the global state, *i.e.,* the total open count, of the devset. To ensure the correctness of the states, the current design of the VFIO driver utilizes one global mutex lock to make the operations on the VFIO device and the operations involving checking or updating the global state of the devset mutually exclusive. However, **such coarse grained mutex lock also serializes the opening operations on the different VFs belonging to the same devset**, and thus hinders the concurrent startup process of SR-IOV enabled secure containers. This accounts for the nearly linear increase in the time cost of *4-vfio-dev* observed in Fig. 4.

3.2.3 Bottleneck 2: DMA memory mapping. Apart from the registration of the VFIO device, another key step in attaching a VF is the DMA memory mapping. As introduced in §2.1, the hypervisor configures the IOMMU to establish the mapping for the microVM memory so the DMA data transmission operations can be correctly performed by the NIC. The DMA memory mapping process can be summarized as three major steps: First, the physical memory for the microVM is allocated in the host to obtain the corresponding HPA. Then the allocated physical memory is pinned in the system to keep it from being swapped out, so that the corresponding HPA remains effective. Finally, the mapping between HPA and IOVA is configured to the page table in IOMMU. We further illustrate the steps in Fig. 5 and analyze the cause of overhead. In the figure, *retrieving* and *zeroing* correspond to the first major step, *pinning* and *mapping* correspond to the other two steps, respectively.

- Page retrieving: When allocating physical memory for the DMA memory, the VFIO driver iteratively collects free physical pages until the requested total size is satisfied.
- Page zeroing: Free pages can contain residual data, which might lead to potential security issues in multi-tenant

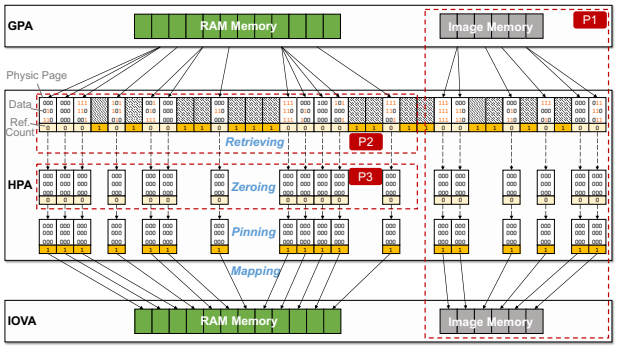


Figure 5. DMA mapping procedure. 4 main steps (page retrieving, zeroing, pinning, mapping) and 3 sub-bottlenecks.

clouds. Thus, the current physical memory allocation implementation ensures that these retrieved physical pages are filled with zeros to clear any sensitive information before they are returned to the VFIO driver.

- Page pinning: Once all free pages are retrieved and zeroed, they are pinned by the VFIO driver: their reference counts are increased to prevent them from being moved or swapped out by the OS. This ensures that the physical address of a physical page remains constant and the corresponding HPA remains effective during DMA operations.
- Page mapping: Then, the IOMMU's page table is updated to set up the mapping between the pinned physical pages (HPA) and the virtual addresses that the device will use (IOVA) for DMA operations.

During the profiling of DMA memory mapping process, we find the following three key factors that make DMA memory mapping a bottleneck in the startup process.

First, **there exists unnecessary DMA memory mapping in the microVM (P1 in Fig. 5).** The original design of the VFIO driver and IOMMU performs DMA mapping for all regions in the memory space of the microVM, as they assume that all the regions have the possibility to be accessed by DMA. However, we identify that the mapping of the microVM image memory region is unnecessary. The image contains the system files of the microVM and a secure container agent procedure used for managing container applications. Its region is read-only and invisible to the container applications that launch DMA operations. In our measurement setup, the microVM image uses 256MB of memory, and Tab. 1 shows that constructing memory mapping of this region constitutes 5.6% (*1-dma-image*) of the total time cost, but the cost is avoidable.

Second, **fragmented small physical pages incur a high retrieving costs (P2 in Fig. 5).** When the VFIO driver iteratively collects free physical pages, the free pages with continuous HPAs will be grouped together and operated as a batch to reduce the time overhead caused by excessive function calls. When the physical pages experience more fragmentation, fewer pages will be batched, resulting in higher retrieving cost. However, we find that such overhead is already effectively mitigated by simply enabling hugepages, a common practice in production environment, as it significantly reduces the number of pages to retrieve. Thus this cause of bottleneck is not a focus of our optimization.

Third, **page zeroing incurs a significant time cost (P3 in Fig. 5).** After reducing the retrieving cost by enabling hugepages, we find that page zeroing contributes to over 93% of the total DMA memory mapping time. Such time cost is not caused by any lock contention but pure zeroing operations. When SR-IOV is not enabled, no DMA memory mapping is performed, and a physical page of the microVM memory is allocated only when it is actually accessed by the application. As a consequence, a page is zeroed only when it is read or written. We refer to this as *lazy zeroing*, which avoids the zeroing overhead during startup as well as the zeroing of unused memory. Our key observation is that page zeroing can be decoupled from physical memory allocation in DMA memory mapping, which makes it possible to enable lazy zeroing for SR-IOV enabled secure containers and motivates our design.

3.2.4 Bottleneck 3: VF driver initialization.

After the VFIO driver configures the VF and hands it over to the microVM, a two-step initialization proceeds to set up the VF as a Linux network interface inside the microVM. First, the VF driver conducts PCI device enumeration to identify the device, registers the device as a network interface, configures its network parameters and updates its link status. Second, the daemon agent of the secure container framework inside the microVM assigns MAC and IP addresses to the interface. It takes a few hundred milliseconds up to seconds for all these operations to complete, and then the interface becomes available. This time cost further increases with the container concurrency. As secure container frameworks manage the initialization and other setup procedures of the microVM in a serial fashion, it only executes the subsequent setups after the interface becomes available, causing non-negligible overhead to the startup performance. Our design will show that such overhead can be effectively mitigated with asynchronous execution.

4 Design

4.1 FastIOV Overview

Fig. 6 displays the key components of FastIOV, including lock disassembling, unnecessary mapping skipping, decoupled zeroing and asynchronous execution. The four optimizations aim at addressing the bottlenecks analyzed in §3.2 to speedup the concurrent startup process. The main workflow of FastIOV is as follows.

When SR-IOV enabled secure containers are launched, VFs are attached to microVMs concurrently. First, FastIOV disassembles the coarse-grained lock in VFIO devset management by proposing a hierarchical lock framework (§4.2.1).

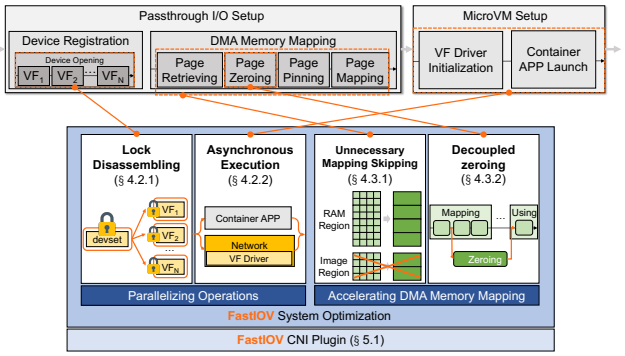


Figure 6. FastIOV Overview.

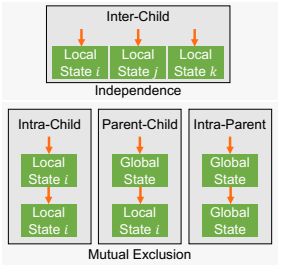
By doing so, FastIOV parallelizes the device opening operations during device registration while maintaining the state correctness of VFIO devset. Then, when the VFIO driver performs DMA memory mapping for the VFs, FastIOV tracks and skips the unnecessary mapping region, *i.e.,* microVM image memory (§4.3.1). As for the remaining regions, FastIOV decouples page zeroing from physical memory allocation to enable lazy zeroing (§4.3.2), which avoids the zeroing time overhead during startup as well as the zeroing of unused memory. Finally, FastIOV asynchronously executes the VF driver initialization inside the microVM and overlaps it with the launching of container application to mask the overhead (§4.2.2).

Next, we will introduce the optimizations in detail. As lock disassembling and asynchronous execution are both aimed at parallelizing operations for speedup, we put them in the same category and introduce them first.

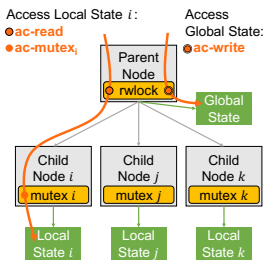
4.2 Parallelizing Operations

4.2.1 Lock disassembling in VFIO devset. The coarse lock problem in VFIO devsets can be abstracted as follows. A devset acts as a parent node and the VFIO devices belonging to it act as child nodes. The parent node has a global state, which is related to the local states of its children. The current design of VIFO driver implements only a global mutex for the entire devset, so it requires the contention of the same mutex whether it is to access the global state of the parent or the local state of a child. When a heavy contention occurs in inter-child operations, *e.g.,* concurrently opening multiple VFs, the system parallelism degrades significantly. On the other hand, simply removing the global mutex will compromise the state consistency in the multi-thread accessing procedure. Our insight is that we can *disassemble the lock to enable independent inter-child operations and hence improve the startup performance, while keeping other operations mutually exclusive to ensure the consistency.*

We distinguish four types of relations between operations according to the data they access: (i) *inter-child operations* access the local states of different child nodes, (ii) *intra-child operations* access the local state of the same child node, (iii) *intra-parent operations* access the global state of the parent node, (iv) *parent-child operations* access the global state of the parent node and the local states of a child node, respectively. As shown in Fig. 7(a), inter-child operations are independent and can be performed in parallel, while operations of the other three types should be mutually exclusive and performed in serial.



(a) Independent and mutually exclusive relations.



(b) Implementing parent-child lock with *rw-lock* and *mutex*s.

Figure 7. Lock disassembling with parent-child lock

To achieve the above requirements, we propose a hierarchical lock disassembling framework built on two Linux kernel locks, read/write lock (*rwlock*) and *mutex*, as shown in Fig. 7(b). In this framework, the parent node is equipped with a global *rwlock* and each child node i is equipped with a local $mutex_i$. When accessing the global state, one needs to acquire the *rwlock* write permission (denoted by *ac-write*). When accessing the i-th local state, one needs to acquire both the *rwlock* read permission (denoted by *ac-read*) and $mutex_i$ (denoted by $ac\text{-}mutex_i$).

We can show that the proposed disassembling framework indeed satisfies the requirements. Here we consider the case of inter-child operations. The other cases can be shown in a similar fashion and omitted. Suppose two inter-child operations on local state i and local state j occur concurrently. Since two *ac-read*s are independent according to the definition of *rwlock*, and $ac\text{-}mutex_i$ and $ac\text{-}mutex_j$ are naturally independent, these operations can be executed in parallel.

Although inventing a new Linux kernel lock can also satisfy the requirements, we believe that reusing off-the-shelf kernel locks keeps the design simple and ensures the effectiveness. Moreover, we believe this lock disassembling framework can be promoted to other scenarios rather than just being used in the VFIO devset.

4.2.2 Asynchronous execution in VF driver initialization. We make two observations regarding the VF driver initialization process, where the network driver inside the microVM initializes and sets up the VF as a Linux network interface. First, the network interface is not utilized until the container application is launched and begins execution inside the microVM. Second, the initialization of the network interface is independent of the other startup stages. This allows the asynchronous execution of the initialization in parallel with other stages, in particular the launching of

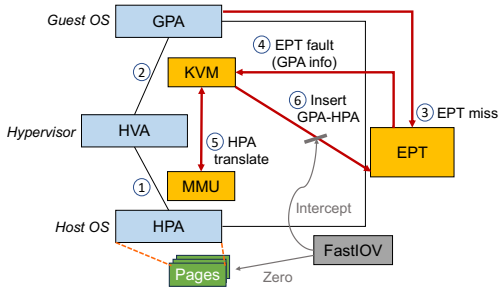


Figure 8. EPT fault based interception and lazy zeroing.

container application in the microVM. The launching process involves transferring container images from the host to the microVM via the shared filesystem and creating the container process. Our empirical measurements show that with a high container concurrency of 200, this process can span several seconds, which is enough to mask the initialization time. We adapt the secure container framework to initialize the network interface asynchronously and employ the framework's daemon agent inside the microVM to periodically check the status of the network interface, ensuring the network is available as the application begins execution.

4.3 Accelerating DMA Memory Mapping

4.3.1 Skipping unnecessary mapping region. FastIOV tracks and skips the unnecessary DMA memory mapping, *i.e.,* microVM image memory, to reduce the overhead. Before the hypervisor, *e.g.,* QEMU, enumerates the DMA memory regions and calls the VFIO driver to perform DMA memory mapping, FastIOV notifies the hypervisor of the information of the image memory region, *i.e.,* its name and size. The hypervisor then skips DMA memory mapping for this region and falls back into its non-DMA memory managing logic.

4.3.2 Decoupling zeroing from mapping. For the remaining regions that are not skipped, *i.e.,* the RAM of the microVM, FastIOV decouples the page zeroing operation from physical memory allocation to enable lazy zeroing. Recall that lazy zeroing means the physical pages are zeroed only when they are actually read or written. The high level idea is to intercept the memory access to physical pages conducted by the microVM, and perform page zeroing when the page is read or written for the first time. We identify three key challenges in achieving this goal.

- First, when a microVM accesses a physical page, it bypasses the hypervisor and relies instead on the hardware-assisted module EPT (previously introduced in §2.1) for address translation. How can we intercept this process and zero the physical pages before their usage?
- Second, if we intercept every memory access to check whether the physical pages are accessed for the first time, it will be very costly and significantly degrade memory performance. How can we avoid such overhead?
- Third, there exist exceptions that the first memory access to a physical page is not conducted by the microVM that it is allocated to. Specifically, the hypervisor may write to the physical pages before starting the microVM, and the para-virtualization components like the shared file system, *i.e., virtioFS*, may write to the physical pages before the microVM reads from them. In such cases, the relevant physical pages should be zeroed before being used by the hypervisor or para-virtualization components, and require no further zeroing before the first access by the microVM. How do we deal with such exceptions to ensure the correctness of zeroing?

The rest of §4.3.2 presents the detailed designs and show how FastIOV solves the above problems.

EPT fault based memory access interception and lazy zeroing. After digging into the details of the EPT address translation mechanism, we find that the entries in the EPT are constructed by an *EPT fault* right before the corresponding physical pages are read or written for the first time. The EPT fault carries the information of the accessed physical pages and is perceived by KVM, a hypervisor module. This gives us the opportunity to intercept the information and perform lazy zeroing. Recall that when the microVM is launched, the VFIO driver performs DMA memory mapping, which allocates physical memory for the microVM. As shown in Fig. 8, the physical memory allocation generates the HVA-HPA mapping in the Memory Management Unit (MMU) of the host (①). Also during the launch of the microVM, the hypervisor module KVM sets up and maintains the GPA-HVA mapping (②). When the microVM accesses a GPA for the first time, it looks it up in the EPT, only to find that there is no match entry (③). Then the microVM triggers an EPT fault, which sends KVM an EPT violation signal containing the GPA information (④). KVM then translates the GPA to HVA, and utilizes the MMU to translate the HVA to HPA (⑤). Finally, KVM inserts the GPA-HPA mapping entry into the EPT (⑥), which is now ready for use by the microVM.

By intercepting the HPA information in the KVM, we can perform lazy zeroing during the EPT fault for the corresponding physical page. As the EPT fault is only generated the first time a physical page is accessed, no subsequent access to the same physical page will be intercepted, thus minimizing the impact on memory performance. Our evaluation in §6.5 will show that the incurred overhead is negligible.

Ensuring the correctness of lazy zeroing. We identify that there are exactly two exceptional scenarios where a physical page requires no further zeroing before the first access by the microVM.

- **Hypervisor data write**. Before launching a microVM, the hypervisor writes to the memory allocated to it in order to perform necessary setup, including loading read-only regions like BIOS and kernel into the memory. Such writes are performed directly without involving the EPT.

After launching, when the microVM tries to access these memory regions for the first time, *e.g.*, to execute kernel code, it will trigger an EPT fault and cause FastIOV to incorrectly zero the data written by the hypervisor, leading to a system crash.

- **Para-virtualization based data transfer**. Devices can utilize para-virtualization protocols, like the widely used *virtio* protocol, to exchange data between the microVM and the host through shared buffers. A typical example is the `virtioFS`, which is a shared file system that allows the container inside the microVM to access designated files on the host. When the microVM reads a file, it first writes the addresses of the file and a shared buffer into a `vring`, which is itself a shared buffer. The backend of *virtio* on the host fetches the addresses from the `vring`, writes the file data into the shared buffer, and notifies the microVM to read it. If the buffer memory has not been accessed before by the microVM, the read operation will trigger an EPT fault, which will cause the FastIOV to incorrectly zero the requested file data before it is read.

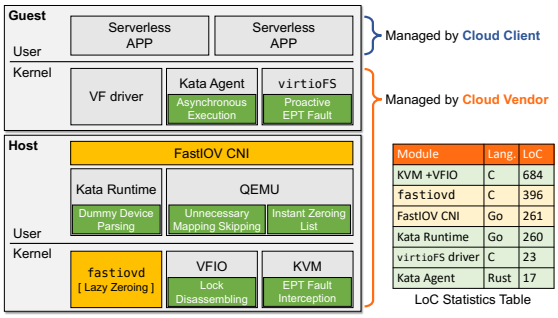
To ensure the correctness of lazy zeroing, FastIOV tackles the above two problems by maintaining an *instant zeroing list* and triggering *proactive EPT fault*, respectively.

The *instant zeroing list* is a white list of physical pages that are not managed by FastIOV and are zeroed instantly when they are allocated. The read-only memory regions like the BIOS and kernel memory are determined before the start of the microVM, and the hypervisor registers them to the *instant zeroing list* maintained by FastIOV. The exclusion of those regions from its management may limit the gain of FastIOV. However, our test shows that with a normal Linux kernel, those regions take up only about 9.4% of the total memory for a microVM with 512MB of memory. The percentage decreases with a larger allocated memory, as the size of the excluded regions remains fixed. Thus FastIOV can still effectively reduces DMA memory mapping time by optimizing the page zeroing of the remaining regions.

To address the exception caused by para-virtualization based data transfer, we proactively triggers EPT faults when the microVM writes the address of a shared buffer to the `vring`, so that FastIOV correctly zeroes the corresponding physical pages before the backend of *virtio* on the host writes the file data back into the buffer. Such proactive EPT faults are triggered by performing data read to the first byte of each page of the buffer.

5 Implementation

The implementation of FastIOV includes a portable Linux kernel module named `fastiovd`, a FastIOV CNI plugin, and several modifications in the hypervisors, container frameworks and host/guest kernel modules. Fig. 9 illustrates their detailed functionalities and the statistics of lines of code (LoC). Note that FastIOV is deployable because all of those



Module	Lang.	LoC
KVM +VFIO	C	684
fastiovd	C	396
FastIOV CNI	Go	261
Kata Runtime	Go	260
virtioFS driver	C	23
Kata Agent	Rust	17

LoC Statistics Table

Figure 9. Implementation of FastIOV.

new modules and modifications are within the management of the cloud vendor. Several details are presented below.

Lazy zeroing implementation. First, we disable the original page zeroing operation in the VFIO kernel module, and maintain in `fastiovd` a two-tier hash table containing the information of the physical pages to be lazy zeroed. The first-tier key uses the process ID (PID) as the independent identifier for each microVM, and its value is the pointer to the secondary hash table maintained for that microVM. The second-tier key is the HPA and its value contains detailed page information. Second, we modify the KVM module to trigger lazy zeroing before it inserts the EPT entry during an EPT fault. The KVM notifies `fastiovd` of the page triggering the EPT fault. If it is in the two-tier hash table, `fastiovd` will zero the page, remove it from the hash table, and notify KVM upon completion. Besides the above lazy zeroing logic, we also maintain a background thread in `fastiovd`, which periodically scans the two-tier hash table, zeroes the remaining pages, and then removes them from the table. Such background clearing in fact overlaps the zeroing with other startup stages to reduce the EPT fault time to further improve container application performance.

FastIOV CNI plugin implementation. The vanilla SR-IOV CNI plugin [11] is designed for traditional containers, where it pre-binds VFs to the host network driver, and simply moves a pre-bound VF to the container network namespace when launching a container. However, when it is applied to secure containers, the pre-binding requires the Kata runtime to rebind the VF to the VFIO driver every time a microVM is launched. Such rebinding is time-consuming and should be avoided. We find that the only reason for pre-binding is to generate a Linux network interface, which serves two functions. First, the Kata runtime identifies the VF by detecting the interface. Second, the CNI performs network operations like IP configurations on the interface, which then passes the configurations to the Kata runtime when it is detected. Therefore, to free VFs from pre-binding, we create dummy Linux network interfaces to fulfill the above two functions instead. This allows us to bind the VF to the VFIO driver only once after the server's booting as mentioned in §2.3. This simple optimization greatly reduces the startup time of

vanilla SR-IOV CNI, from several minutes to 16.2 seconds when concurrently starting 200 containers. However, we regard this binding problem as an implementation drawback that may depend on the specific container framework, Kata containers in this case. In order to focus on problems intrinsic to the SR-IOV CNI, we apply the above optimization to the vanilla SR-IOV CNI in our evaluation for fair comparison.

6 Evaluation

6.1 Experiment Setup

Testbed setup. We conduct two categories of experiments that evaluate FastIOV's network startup performance and overall performance with serverless application benchmarks, respectively. The former runs on a single test server, while the latter on two directly-connected test servers acting as the application server and the storage server, respectively. All test servers mentioned above have the same hardware and software configurations as specified in §3.1.

Baselines. We compare FastIOV with the following baselines to validate the effectiveness of our designs.

- ***No network***: The startup without enabling network. This represents a lower bound for optimizing network startup.
- ***Vanilla***: The original implementation of SR-IOV CNI [11] without optimization for passthrough I/O. Recall that for fair comparison, we enhance *Vanilla* with the dummy Linux network interface optimization as described in §5.
- ***FastIOV* variants**: In order to evaluate the effectiveness of our four optimization designs, *i.e.,* **L**ock disassembling, **A**synchronous execution, unnecessary mapping **S**kipping and **D**ecoupled zeroing, we remove them from FastIOV one at a time and get *FastIOV-L*, *FastIOV-A*, *FastIOV-S* and *FastIOV-D*, respectively.
- **Memory pre-zeroing methods**: Memory **Pre**-zeroing is a popular technique proposed by HawkEye [52] that performs page zeroing during memory idle time to achieve faster page fault. It has also been utilized by the open-source community to speedup DMA memory mapping and accelerate the booting of passthrough I/O enabled VMs. The performance of this baseline is affected by the fraction of memory pre-zeroed during memory idle time. To evaluate its performance across different scenarios, we set the fraction to be 10%, 50% and 100%, and represent them by *Pre10*, *Pre50* and *Pre100*, respectively.
- **Software CNI**: Besides the SR-IOV baselines, we also compare FastIOV to a software CNI in §6.4 aiming at illustrating the bottleneck differences between the two types. We choose the basic software CNI *IPvtap*, because (i) it shares similar virtual network device implementation with popular software CNIs like Flannel [9] and Calico [7], but has faster startup due to its lack of support for more advanced network features; and (ii) it is the basic software CNI with the fastest startup for secure containers according to our preliminary measurements.

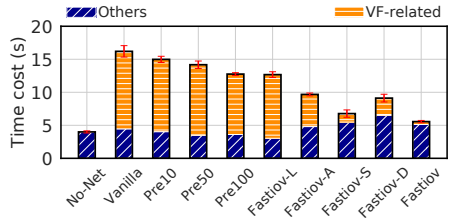


Figure 10. Average startup time. Concurrency = 200

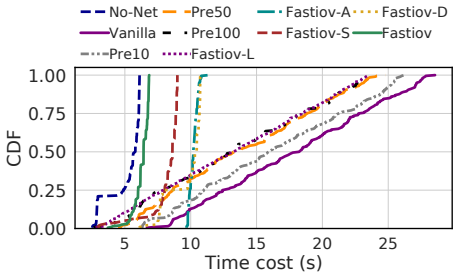


Figure 11. Startup time distribution. Concurrency = 200

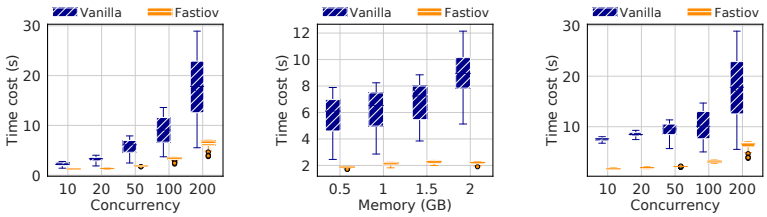
6.2 Startup Performance

We compare *FastIOV* with other baselines by measuring the startup time with 200 concurrently invoked secure containers. Fig. 10 displays the average time and the breakdown into two parts, *VF-related* and *others*. *VF-related* refers to the time of the four VF-related stages previously introduced in §3.2 and *others* represents the remaining part of time. We draw the following three key conclusions from the results.

First, *FastIOV* significantly outperforms the vanilla SR-IOV CNI in both the average and long-tail time cost. *FastIOV* reduces the average startup time by 65.7% compared with *vanilla*. Specifically, *FastIOV* reduces the time overhead directly related to VF operations by 96.1%, significantly mitigating the effect of enabling VF on secure container startup. Moreover, the time distribution in Fig. 11 shows that *FastIOV* also reduces the 99th percentile startup time of *vanilla* by 75.4%, largely improving the long-tail performance. In addition, FastIOV achieves a startup time close to that of *No-Net*, with the average and the 99th percentile startup time being 39.1% and 11.6% higher, respectively. In contrast, the corresponding figures of *vanilla* are substantially larger, *i.e.,* 305.2% and 354.5%.

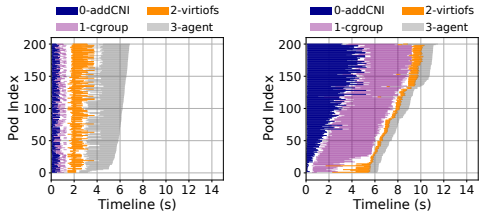
Second, each of our optimization techniques makes obvious contribution to the time reduction achieved by FastIOV. Compared with *Vanilla*, *FastIOV-L*, *FastIOV-A*, *FastIOV-S* and *FastIOV-D* reduce the average time by 21.8%, 40.3%, 58.2% and 43.7%, respectively, all smaller than the 65.7% reduction achieved by the full FastIOV.

Third, FastIOV outperforms the memory pre-zeroing methods and further reduces the average time by 56.4% compared with *Pre100*. The performance of pre-zeroing strongly depends on the fraction of memory pre-zeroed during memory idle time. In practice, cloud vendors tend to maintain a high level of memory utilization for more revenue. For example,



(a) Varying concurrency. (b) Varying resource. (c) Fully loaded server.

Figure 12. Impacting factors.



(a) FastIOV (b) IPvtap

Figure 13. Comparison with software CNI.

AWS uses a bin-pack algorithm for this purpose, and its average server memory utilization ranges from 84.6% to 100%, with a median of 96.2% [59]. This leaves short memory idle time and can further limit the performance of pre-zeroing.

6.3 Impacting Factors

Concurrency. Fig. 12(a) shows the impact of varying concurrency. It reports the startup time distribution with container concurrency increasing from 10 to 200 and each allocated 512MB of memory. We observe that FastIOV is effective across all concurrency, achieving time reductions ranging from 46.7% to 65.6%. The reduction is more obvious with a higher concurrency, as the lock contention in VFIO devset becomes more severe with more concurrently invoked VFs.

Resource allocation. Fig. 12(b) shows the impact of varying per-container resource requirement. More precisely, it shows the time distribution of concurrently starting 50 containers with memory allocation for each container increasing from 512MB to 2GB. We observe an obvious increase of 60.5% in the average startup time of *vanilla* as the memory allocation increases to 2GB, while only 21.5% with FastIOV. This is because the optimization of *FastIOV* on DMA memory mapping makes its startup time less sensitive to allocated memory. Thus it achieves higher time reduction ratio when more resources are allocated.

Fully loaded server. As mentioned before, cloud vendors like AWS tend to schedule containers to maximize the utilization of server resources, *i.e.,* memory and CPU. Here we consider a scenario that tries to partially capture this behavior. We vary the concurrency, and for each given concurrency, we evenly divide all the resources of the server among the concurrent containers. Note that fewer containers means more allocated resources for each. The startup time distribution in Fig. 12(c) shows that FastIOV achieves large time reductions across all settings, even with low concurrency. In fact there is an increase in the time reduction ratio, from 65.7% to 79.5% as the concurrency decreases from 200 to 10. This is because a lower concurrency reduces the time of the other startup steps unrelated to SR-IOV, while the optimization of DMA memory mapping is unaffected, as the total allocated memory stays unchanged.

6.4 Bottleneck Differences with Software CNI

We compare FastIOV with the software CNI *IPvtap* to illustrate how the startup bottlenecks of a software CNI differ from those of an SR-IOV based solution. The software CNI emulates the physical network devices of microVMs, and thus obviates the time-consuming passthrough I/O setup procedure. A comparison of Fig. 10 and Fig. 13 shows that *IPvtap* has faster startup than *vanilla SR-IOV*, although with a much worse data-plane performance. On the other hand, Fig. 13 shows that *FastIOV* achieves 41.3% and 31.8% lower total and average startup time than *IPvtap*.

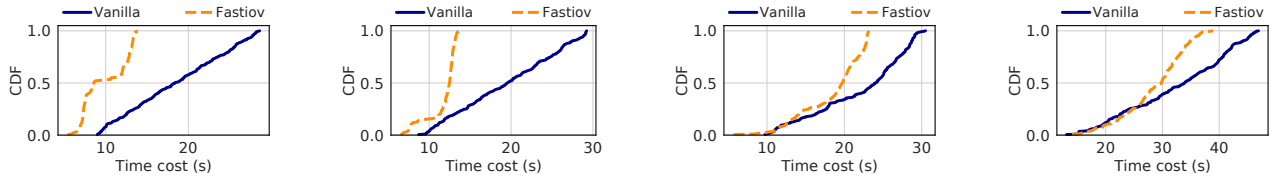
The deficiency of *IPvtap* results mostly from two parts: (i) the creation and configuration of the virtual network device (denoted by *addCNI*), and (ii) the host resource isolation (denoted by *cgroup*). Through detailed measurements, we identify that the severe lock contentions in the kernel network calls and cgroup operations bring in much overhead. In contrast, SR-IOV based CNIs attach VFs to the secure container without creating any additional virtual network device. Thus with FastIOV optimizing the time-consuming passthrough I/O setup procedure, a SR-IOV based solution is more capable of achieving ultra-fast concurrent startup for secure containers.

6.5 Impact on Memory Access Performance

To evaluate the effect of FastIOV on the memory access performance, we use an open-source tool *Tinymembench* to test the memory throughput and latency within the secure container. To obtain the throughput, the tool performs *memcpy* operations on 2048-byte data blocks for 5 seconds and repeats the whole process for 10 times. To obtain the latency, it performs random byte reading for 10 million times. The results show that *FastIOV* achieves comparable memory access performance as *vanilla*, with memory throughput degradation and latency increase within 1%. Since *FastIOV* only intercepts the EPT page fault once upon the first-time memory access, it does not affect the subsequent memory operations and thus causes negligible performance degradation.

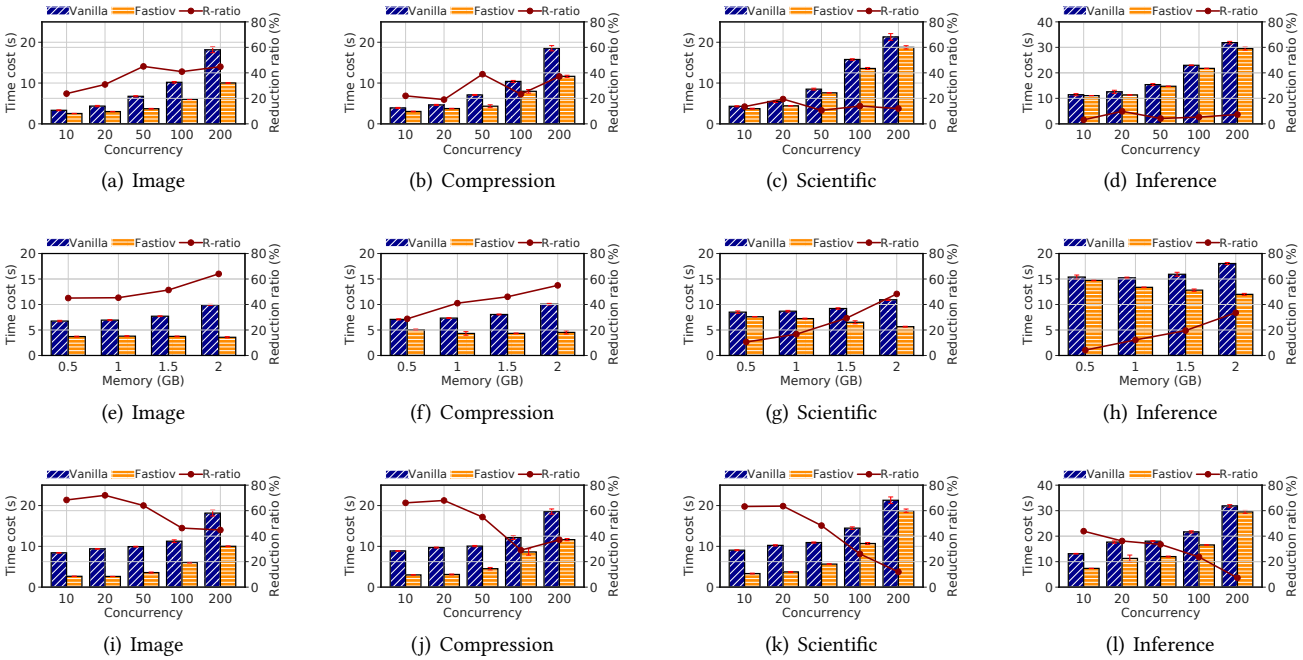
6.6 Performance in Serverless Applications

Benchmark applications. To evaluate the overall speedup brought by FastIOV on serverless applications, we choose



(a) Image (b) Compression (c) Scientific (d) Inference

Figure 14. Serverless application performance.



(a) Image (b) Compression (c) Scientific (d) Inference

(e) Image (f) Compression (g) Scientific (h) Inference

(i) Image (j) Compression (k) Scientific (l) Inference

Figure 15. Performance of FastIOV on serverless applications with varying concurrency (a~d), varying resource allocation (e~h), or fully loaded server (i~l). *R-ratio*: time reduction ratio achieved by *FastIOV* compared with *Vanilla*.

four representative tasks, *i.e.*, *Image*, *Compression*, *Scientific* and *Inference*, from the widely adopted SeBS [17] serverless benchmark. *Image* resizes an input image to a thumbnail of size 100x100. *Compression* zips an input file of 9.7MB. *Scientific* performs a breadth-first search to traverse a graph of 100000 nodes. *Inference* utilizes ResNet-50 model for ImageNet classification task. Each application first downloads input data from the storage server through the VF assigned to its secure container before performing the computation.

Overall performance. Fig. 14 illustrates the task completion time distribution of running the four serverless applications on 200 concurrently launched containers. The task completion time refers to the duration from the issuing of the startup command to the completion of the container application. Compared to *vanilla*, *FastIOV* achieves 12.1%-53.5% and 20.3%-53.7% reduction in the average and the 99th percentile task completion time across all applications. We notice that the reduction ratio decreases from application *Image* to *Inference*, which is attributed to the fact that the task execution time increases from *Image* to *Inference*, reducing the portion that container startup takes up in the total time. This suggests that the benefits of FastIOV are more pronounced with shorter-lived applications.

Impacting factors. Similar to §6.3, we evaluate the performance of FastIOV on serverless applications with varying concurrency, varying resource allocation, and also in the case of a fully loaded server. Fig. 15 reports the average task completion time as well as the time reduction ratio achieved by *FastIOV*. The overall trend is similar to that in §6.3: (i) with a fixed per container resource allocation, FastIOV achieves higher performance gain with a higher concurrency (a~d); (ii) with a fixed concurrency, FastIOV achieves higher performance gain with larger per container resource allocation (e~h); (iii) with a fully loaded server, FastIOV achieves obvious time reductions across all applications and all concurrency settings, and the reductions are pronounced with lower concurrency (i~l).

Compared with §6.3, there is a notable difference when we vary the resource allocation per container: with more resource allocation, the task completion time of FastIOV stays unchanged (cf. Fig. 15(e) and Fig. 15(f)) or even decreases (cf. Fig. 15(g) and Fig. 15(h)). This is because the increased resource allocation shortens the task execution time and thus reduces the task completion time for FastIOV. However, the task execution time reduction fails to outweigh the increased DMA memory mapping overhead for *vanilla*, thereby increasing its task completion time. This demonstrates that *FastIOV* enables applications to more effectively reap the benefits of increased resource allocation.

7 Related Works

CNI enhancements. The cloud-native community has been proposing plenty of widely-used CNI plugins like Flannel [9], Calico [7] and Cilium [8]. The state-of-the-arts mainly choose them as baselines and try to optimize their data-plane performance using techniques like pipeline parallelism [29], resource allocation optimization[28] or VXLAN enhancement [14, 33]. Relatively fewer works have recognized the significance of the startup performance [45, 57], and those works only optimize the startup of software based CNIs for traditional containers. PCPM [45] pre-creates the virtual network devices and network configurations as pause containers, and dynamically attaches them to newly launched containers. However, when using SR-IOV for secure containers, the startup bottleneck does not lie in the creation of the VFs but in the attaching process. Particle [57] identifies the startup bottleneck of using the `veth`-based software CNI as the NNS moving and resolves this problem by sharing the NNS. However, our previous measurements show that this time cost is not the key bottleneck when using secure containers either with software based or SR-IOV based CNIs.

SR-IOV enhancements. Due to the good data-plane performance with the high throughput and low latency, SR-IOV outperforms other forms of network I/O virtualization and has been widely adopted in various applications [3, 13, 20, 25, 38, 41, 46, 47]. Many works make a step further to enhance SR-IOV's performance. They make up for the lack of the live migration [24, 51, 61, 63], improve the deployment density [18, 44, 53], avoid the performance degradation caused by frequent transmission interrupts [23, 31] and enhance the logic isolation and performance isolation for multi-tenancy scenarios [27, 64]. However, none of those works recognizes the demand for improving SR-IOV's concurrent startup performance or provides solutions as we do in this paper.

Passthrough I/O optimizations. An important line of work regarding passthrough I/O is the optimization of the IOMMU module [5, 6, 39, 40, 42, 58, 60]. Among those works, the most relevant to our FastIOV are the designs of virtual IOMMU [5, 58, 60]. vIOMMU [5] identifies that the page pinning operation of DMA memory mapping in IOMMU prohibits memory over-commitment. It introduces an IOMMU emulation layer to delay the mapping establishment and perform mapping when a memory region is actually accessed by DMA. coIOMMU [58] relieves vIOMMU's performance degradation problem by decoupling the DMA mapping and page pinning process. V-Probe [60] further solves the intrusiveness problem in coIOMMU's design by adopting an eBPF based design. The delayed DMA memory mapping in those virtual IOMMUs can reduce the startup cost of passthrough I/O. However, such reduction is coupled with the enabling of memory-overcommitment, which is not always the preferred option in multi-tenant clouds [36]. By comparison, our FastIOV decouples the root cause of overhead, *i.e.,* page zeroing, from memory mapping to accelerate the startup, making it more flexible and applicable whether overcommitment is enabled or not.

VM/Container concurrency improvements. The majority of related works in this category focus on optimizing the startup performance of traditional containers. They reduce startup time by accelerating container image distribution [30, 37], introducing specific checkpoint or general template-based runtime [19, 50], or providing warm startup solutions with technologies like workload prediction and adaptive pooling [55, 62, 65]. Another series of work optimizes the startup of microVMs or VMs using techniques such as kernel trimming [2, 32, 43], cgroup pre-creation [32], hypervisor lock [48] and control plane redesign [43]. Those works focus on optimizing the non-network part of the startup, and are orthogonal to our work.

8 Conclusion

In the context of secure containers, SR-IOV enabled network achieves a high data plane performance, a high deployment density, but a poor concurrent startup performance. The goal of this paper is to make up for its shortcoming. First, three key bottlenecks are identified: (i) the contention time cost of the coarse lock used in VFIO devset management, (ii) the absolute time cost of the unnecessary DMA memory mapping and the deeply coupled memory zeroing in the mapping procedure, (iii) the contention time cost of the VF driving initialization process. To conquer them, we propose a complete solution named FastIOV with dedicated optimization methods like lock dissembling, unnecessary mapping skipping, decoupled zeroing, and asynchronous VF driver initialization. In FastIOV, we first develop the two major components including a portable Kernel module and an optimized CNI plugin, and then apply several adoptable optimizations in the could-vendor managed infrastructures. Compared to the vanilla SR-IOV CNI, FastIOV reduces the VF-related startup time by 96.1% and the end-to-end startup time by 65.7% with negligible loss in the data plane performance.

References

[1] 2024. PCI Special Interest Group. http://www.pcisig.com/home. 2024-05-22.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of USENIX NSDI*. 419–434.

[3] Giuliano Albanese, Robert Birke, Georgia Giannopoulou, Sandro Schönborn, and Thanikesavan Sivanthi. 2021. Evaluation of Networking Options for Containerized Deployment of Real-Time Applications. In *Proceedings of IEEE International Conference on Emerging Technologies and Factory Automation*. 1–8.

[4] Amazon. 2023. AWS Lambda. https://www.aliyun.com/product/fc.

[5] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, Assaf Schuster, et al. 2011. vIOMMU: Efficient IOMMU Emulation. In *Proceedings of USENIX ATC*. 1–14.

[6] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *International Symposium on Computer Architecture*. Springer, 256–274.

[7] Github Authors. 2024. Calico CNI Project. https://github.com/projectcalico/calico. (Accessed on 2024-04-02).

[8] Github Authors. 2024. Cilium CNI Project. https://github.com/cilium/cilium. (Accessed on 2024-04-02).

[9] Github Authors. 2024. Flannel CNI Project. https://github.com/flannel-io/flannel. (Accessed on 2024-04-02).

[10] Github Authors. 2024. Network Device Plugin for Kubernetes. https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin. (Accessed on 2024-04-02).

[11] Github Authors. 2024. SR-IOV CNI Plugin Project. https://github.com/hustcat/sriov-cni. (Accessed on 2024-04-02).

[12] Azure. 2023. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db.

[13] Jae-Geun Cha and Sun Wook Kim. 2021. Design and Evaluation of Container-based Networking for Low-latency Edge Services. In *Proceedings of IEEE International Conference on Information and Communication Technology Convergence*. 1287–1289.

[14] Sunyanan Choochotkaew, Tatsuhiro Chiba, Scott Trent, and Marcelo Amaral. 2022. Bypass Container Overlay Networks with Transparent BPF-driven Socket Replacement. In *Proceedings of IEEE International Conference on Cloud Computing*. 134–143.

[15] CNCF Authors. 2024. Containerd: An Industry-Standard Container Runtime with An Emphasis on Simplicity, Robustness and Portability. https://containerd.io/. (Accessed on 2024-04-02).

[16] CNCF Authors. 2024. Kubernetes: An Open-Source System for Automating Deployment, Scaling and Management of Containerized Applications. https://kubernetes.io/. (Accessed on 2024-04-02).

[17] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of ACM/IFIP International Middleware Conference*. 64–78.

[18] Intel Corporation. 2024. Scalable I/O Virtualization Technical Specification. https://cdrdv2-public.intel.com/671403/intel-scalable-io-virtualization-technical-specification.pdf. (Accessed on 2024-04-02).

[19] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of ACM ASPLOS*. 467–481.

[20] Brice Ekane, Tu Dinh Ngoc, Boris Teabe, Daniel Hagimont, and Noel De Palma. 2022. FlexVF: Adaptive network device services in a virtualized environment. *Future Generation Computer Systems* 127 (2022), 14–22.

[21] Open Infrastructure Foundation. 2024. Kata Containers: the Speed of Containers, the Security of VMs. https://katacontainers.io/. (Accessed on 2024-02-05).

[22] Github Authors. 2024. Quick EMUlator (QEMU) Project. https://github.com/qemu/qemu. (Accessed on 2024-04-02).

[23] HaiBing Guan, YaoZu Dong, Kun Tian, and Jian Li. 2013. SR-IOV based Network Interrupt-Free Virtualization with Event based Polling. *IEEE JSAC* 31, 12 (2013), 2596–2609.

[24] Wei Lin Guay, Sven-Arne Reinemo, Bjørn Dag Johnsen, Tor Skeie, and Ola Torudbakken. 2012. A Scalable Signalling Mechanism for VM Migration with SR-IOV over Infiniband. In *Proceedings of IEEE ICPADS*. 384–391.

[25] Shu Huang and Ilia Baldine. 2012. Performance evaluation of 10ge nics with sr-iov support: I/o virtualization and network stack optimizations. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance: 16th International GI/ITG Conference, MMB & DFT 2012, Kaiserslautern, Germany, March 19-21, 2012. Proceedings 16*. 197–205.

[26] Intel. 2024. Intel Infrastructure Processing Unit (Intel IPU) SoC E2100 Product Brief. https://www.intel.com/content/www/us/en/content-details/818147/intel-infrastructure-processing-unit-intel-ipu-soc-e2100-product-brief.html. (Accessed on 2024-05-19).

[27] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. 2023. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *Proceedings of USENIX NSDI*. 31–48.

[28] Kyungwoon Lee, Kwanhoon Lee, Hyunchan Park, Jaehyun Hwang, and Chuck Yoo. 2022. Autothrottle: satisfying network performance requirements for containers. *IEEE Transactions on Cloud Computing* (2022), 1–13.

[29] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. 2021. Parallelizing Packet Processing in Container Overlay Networks. In *Proceedings of ACM EuroSys*. 1–16.

[30] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. DADI: Block-Level Image Service for Agile and Elastic Application Deployment. In *Proceedings of USENIX ATC*. 727–740.

[31] Jian Li, Shuai Xue, Wang Zhang, Ruhui Ma, Zhengwei Qi, and Haibing Guan. 2017. When I/O Interrupt Becomes System Bottleneck: Efficiency and Scalability Enhancement for SR-IOV Network Virtualization. *IEEE Transactions on Cloud Computing* 7, 4 (2017), 1183–1196.

[32] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-Concurrency Startup in Serverless Computing. In *Proceedings of USENIX ATC*. 53–68.

[33] Shengkai Lin, Peirui Cao, Tianyi Huang, Shizhen Zhao, Quan Tian, Qi Wu, Donghai Han, Xinbing Wang, and Chenghu Zhou. 2023. XMasq: Low-Overhead Container Overlay Network Based on eBPF. *arXiv preprint arXiv:2305.05455* (2023).

[34] Linux Kernel Organization. 2024. Kernel Virtual Machine. https://linux-kvm.org/page/Main_Page. (Accessed on 2024-04-02).

[35] Linux Kernel Organization. 2024. The Linux Kernel Archives. https://www.kernel.org/. (Accessed on 2024-04-02).

[36] Fangming Liu and Yipei Niu. 2023. Demystifying the cost of serverless computing: Towards a win-win deal. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[37] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. 2019. CFS: A Distributed File System for Large Scale Container Platforms. In *Proceedings of ACM SIGMOD*. 1729–1742.

[38] Hanjiang Liu, Yaoping Luo, Buhua Chen, and Yu Yang. 2023. Performance Evaluation of Container Networking Technology. In *Proceedings of IEEE International Conference on Information Technology, Big Data and Artificial Intelligence*. 815–818.

[39] Ming Liu, Tao Li, Neo Jia, Andy Currid, and Vladimir Troy. 2015. Understanding the virtualization "Tax" of scale-out pass-through GPUs in GaaS clouds: An empirical study. In *2015 IEEE 21st International*

Symposium on High Performance Computer Architecture (HPCA). 259–270.

[40] Chen Lv, Fuxin Zhang, Xiang Gao, and Chen Zhu. 2022. LA-vIOMMU: An Efficient Hardware-Software Co-design of IOMMU Virtualization. In *(ISPA/BDCloud/SocialCom/SustainCom)*. 246–253.

[41] Diego Rossi Mafioletti, Cristina Klippel Dominicini, Magnos Martinello, Moises R. N. Ribeiro, and Rodolfo da Silva Villaça. 2020. PIaFFE: A Place-as-you-go In-network Framework for Flexible Embedding of VNFs. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. 1–6.

[42] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. 2015. rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. New York, NY, USA, 355–368.

[43] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter and Safer than Your Container. In *Proceedings of ACM SOSP*. 218–233.

[44] NVIDIA Mellanox. 2024. Scalable Function Overview. https://github.com/Mellanox/scalablefunctions/wiki. (Accessed on 2024-04-02).

[45] Mohan, Anup and Sane, Harshad and Doshi, Kshitij and Edupuganti, Saikrishna and Nayak, Naren and Sukhomlinov, Vadim. 2019. Agile Cold Starts for Scalable Serverless. In *Proceedings of USENIX HotCloud*. 1–6.

[46] T.P. Nagendra and R. Hemavathy. 2023. Unlocking Kubernetes Networking Efficiency: Exploring Data Processing Units for Offloading and Enhancing Container Network Interfaces. In *Proceedings of IEEE Global Conference for Advancement in Technology*. 1–7.

[47] Dinh Tam Nguyen, Ngoc Lam Dao, Van Thuyet Tran, Khac Thuan Lang, Thanh Tu Pham, Phi Hung Nguyen, Cong Dan Pham, Tuan Anh Pham, Duc Hai Nguyen, and Huu Thanh Nguyen. 2022. Enhancing CNF Performance for 5G Core Network Using SR-IOV in Kubernetes. In *Proceedings of IEEE International Conference on Advanced Communication Technology*. 501–506.

[48] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. 2017. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. *ACM SIGPLAN Notices* 52, 7 (2017), 1–14.

[49] NVIDIA. 2024. NVIDIA CONNECTX-7 400G ETHERNET. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf. (Accessed on 2024-05-19).

[50] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of USENIX ATC*. 57–70.

[51] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. 2012. CompSC: Live Migration with Pass-through Devices. In *Proceedings of ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. 109–120.

[52] Ashish Panwar, Sorav Bansal, and K Gopinath. 2019. Hawkeye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of ACM ASPLOS*. 347–360.

[53] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. 2018. MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through. In *Proceedings of USENIX ATC*. 665–676.

[54] Shixiong Qi, Sameer G. Kulkarni, and K.K. Ramakrishnan. 2020. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability. *IEEE Transactions on Network and Service Management* 18, 1 (2020), 656–671.

[55] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of ACM ASPLOS*. 753–767.

[56] Kun Suo, Junggab Son, Dazhao Cheng, Wei Chen, and Sabur Baidya. 2021. Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing. In *Proceedings of IEEE International Conference on Cluster Computing*. 433–443.

[57] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: Ephemeral Endpoints for Serverless Networking. In *Proceedings of ACM SoCC*. 16–29.

[58] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. 2020. coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O. In *Proceedings of USENIX ATC*. 479–492.

[59] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of USENIX ATC*. 133–146.

[60] Yaohui Wang, Ben Luo, and Yibin Shen. 2023. Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Metadata Management. In *Proceedings of USENIX ATC*. 769–783.

[61] Xin Xu and Bhavesh Davda. 2017. A Hypervisor Approach to Enable Live Migration with Passthrough SR-IOV Network Devices. *ACM SIGOPS Operating Systems Review* 51, 1 (2017), 15–23.

[62] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In *Proceedings of ACM ASPLOS*. 768–781.

[63] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K Panda. 2017. High-Performance Virtual Machine Migration Framework for MPI Applications on SR-IOV Enabled InfiniBand Clusters. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*. 143–152.

[64] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. 2022. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *Proceedings of USENIX NSDI*. 1307–1326.

[65] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2023. Aquatope: QoS-and-Uncertainty-Aware Resource Management for Multi-Stage Serverless Workflows. In *Proceedings of ACM ASPLOS*. 1–14.