A Light Application Level Virtualization Framework with Zero-Copy and Migration Agent

Anonymous Author(s)
Submission Id: 181[†]

Abstract

Cloud computing utilizes physical hardware through virtualization technology. Virtualization allows users to overcommit resources so that virtual machines can utilize physical resources. However, applications running in a virtual machine suffer significant performance degradation as they do not take advantage of the features of the physical hardware, which cannot be overcome by the optimization of the virtual hardware. In order to reduce this performance overhead, an application virtualization mechanism is proposed to expose the application instance running in the host OS to the virtual machine, the virtual machine can avoid the overhead of virtual machine and run applications with optimal performance. A virtio-based application virtualization implementation named virtio-vapp is also proposed to show the adaptions needs in a hypervisor and an operating system. The virtio-vapp is evaluated against other communication components in Qemu to show the high load capacity and low latency of virtio-vapp. In the evaluation, virtio-vapp has a bandwidth of 36.4 $Gbit/s$ and a transmission delay of only 60 μs in experiments, which is almost the same performance as the host. In addition to high load capacity and low latency, the virtio-vapp also supports limited migration to allow virtual machines to run applications with the same data. It is obvious that using application virtualization significantly reduces the performance head and improves flexibility and usability compared to running applications directly in the virtual machine.

CCS Concepts: **• Software and its engineering → Virtual machines**; **• Computer systems organization → Cloud computing**.

Keywords: Application virtualization, Virtualization, Utilization

1 Introduction

Virtualization has become a key technology of cloud computing with the benefits of isolation, flexibility, usability, and high availability[10]. It makes all the physical hardware resources fully used in a machine, since multiple application instances can run in isolated virtual environments in one physical machine. Also, it can enhance security as all operations in the virtual environments are not affected the host environments [3, 14]. So applications and their environments can be easily deployed to any machine with high availability. Two main methods are popular to deploy a virtualized environment: container and virtual machine (VM). However, from the perspective of applications, virtualization also requires a transparent method that can efficiently support the operation of services while guaranteeing the isolation of virtualization. Therefore, the research in the area of application virtualization is necessary.

With the views of the traditional virtualization, container provides OS-level virtualization and VM provides hardware-level virtualization[24]. Containers have a performance advantage with a lower degree of virtualization, which can share resources easily. While, if applications require a bare metal environment, VM is the main way to adapt these applications, and it also can run the applications compiled for the target architecture different from the host. But, the usage of VMs will cause some problems, such as performance penalty, duplicate applications and environments, data sharing[1, 12, 18, 24]. First of all, the most notable problem is performance penalty[12], because most of the devices in the VM are emulated to use the host CPU to provide device functions, this leads to significant performance overhead in the whole system[15], and also cause severe latency for the applications in the VM[32]. Another problem for the VM is the duplicate applications and environments[18], when an application needs an OS and generic userspace to provide basic functionalities since VM needs a full-featured OS kernel in most cases. In the worst case, the number of VMs is the same as that of the instance, while the container only needs one image [24]. Furthermore, because of the superior isolation, the difficulty of direct data sharing between VMs is even harder than that of physical machine[1]. The only way is even through network stack or some para-virtualized hardware. As applications in the container may still be able to acquire the information of the host hardware, applications that require a high degree of isolation perfer to run inside a VM rather than inside a container.

Some advances have been made to improve the efficiency of VMs. A straightforward method for efficiency is to improve the efficiency of the virtual hardware. For instance, the virtio provides a de-factor standard for I/O device to use para-virtualization to increase performance [23]. As the virtual hardware do not have its own data processing capabilities, the efficient of the virtual hardware is hard to increase[25]. For instance, the widely used emulated NIC *e1000* requires CPU to performance related to the pci and its functionalities[22]. So, instead of running applications in the traditional VMs, a compromise may appear to shift the application payload to the native host without the emulated hardware and guest OS to utilize the physical hardware fully.

In this paper, the application virtualization is proposed, which is a novel virtualization method for guest VMs to access host programs and data as those in the VM. The application virtualization framework uses four modules to integrate both the host and the VM, as host userspace component, kernel component, virtual machine hypervisor component, and virtual machine userspace component. It also uses the abstracted data channels to achieve direct data transfer between the host and the VM. For the management of the host applications, the framework introduces the service hypervisor as the entry point and management platform for the virtualized applications. Meanwhile, the application virtualization can achieve restricted migration with the use of OS kernel as a relay agent. Thus, it allows workload shifting of supported applications from the VM to the host, and provides flexible permission management to guarantee data access across various VMs. The permission management allows a VM share its application data to another VM. Application virtualization can be integrated into various virtualization frameworks with limited changes. In order to achieve application virtualization, a virtio-based implementation named virtio-vapp is proposed. virtio-vapp is architecture-independent and focuses on high bandwidth and low latency I/O in addition to all functionalities of application virtualization. To test the capability of virtio-vapp, the performed bandwidth and concurrency test and the I/O test are performed on the virtio-vapp. These experiments show that VAPP can achieve a bandwidth of 36.4 $Gbit/s$ and a transmission delay of only 60 μs while maintaining 100% CPU utilization, which is almost the same performance of the host.

The main contribution can be concluded as the following:

- Provide a general mechanism for virtual machines to directly access the program and data in the host OS with optimal performance.
- Allow the virtual machine to shift the workload of applications to the host. The host OS can dynamically adjust the workload of applications the VM needs.
- Allow the virtual machine to access the same host programs with restrictions after migration.

Obviously, virtio-vapp can be used as the platform with restricted resources that needs virtualization (e.g., edge service node). It provides an out-of-tree kernel module and user runtime for both the VM and the host. With virtio-vapp, the service provider can guarantee the performance of service while keeping the data in a secure environment.

2 Background and Related Work

Partition is the secret to the high utilization of virtualization. VMs can utilize the hardware by scheduling and keeping it at high utilization. The partition of resources includes two main points: the type and the quantity.

2.1 Resource Dynamic Allocation

Resource allocation is the most important part of the virtualization preparation process. The success of virtualization relies on the ability to dynamically change the resources. This dynamic change can be applied to the CPU, memory, and storage of the VM [19]. So all VMs can be adjusted to run at optimal cost. For instance, a VM can dynamically reduce resource if the VM has a large amount of idle resources. Meanwhile, the VM can also increase the resource if the VM does not have enough resources for a heavy workload.

However, it is difficult to determine the actual size of resources needed for a VM [16]. Most applications are also not designed for running in the VMs which have limited resources and performance [5]. For instance, the database software requires lots of CPU, RAM, and disk resources to get optimal performance. But these resources are not available or enough for the VMs all the time. Another reason contributing to this difficulty is that the workload of application is dynamic. If the change of workload is frequent and large, the VM may need to keep a large amount of resources to run smoothly, or keep changing the amount of resources held by the VM [31]. The VM should find a way to handle this kind of software at a low cost.

It is not realistic to give large amounts of resources to a single VM in the production environment. A VM is hard to manage and migrate and will waste considerable resources if it has lots of resources [2, 26]. It is ideal that the application can split into several separate parts and deploy them into independent VMs. However, large and complex applications are difficult to split into multiple components that can be deployed independently in reality.

2.2 Virtual Hardware Optimization

In addition to resource allocation, another important part of virtualization is the performance and isolation of the virtual hardware. As the application executes on the virtual hardware but not the physical one. Improving the performance leads to a significant performance boost for the whole VM.

A lot of works have been applied to improve virtual network interface efficiency [9, 17], as the traditional network efficiency is usually the most noticed. There are also some work trying to bing vGPU into the VM[13, 27]. In the field of memory, a virtio-based memory device has already been proposed and improved[8]. And virtualized persistent memory is also considering as a standard component for high performance application[11].

In addition to the traditional hardware, some general transferring protocols have also been attempted for efficiency in VMs with virtual hardware [6, 20]. One step closer, some application-defined I/O transfer methods are also studied, for instance, migration [2], kernel-bypass network [28] abd RDMA [7, 30]. Even some API framework, like CUDA, are also evaluated in the VM [21].

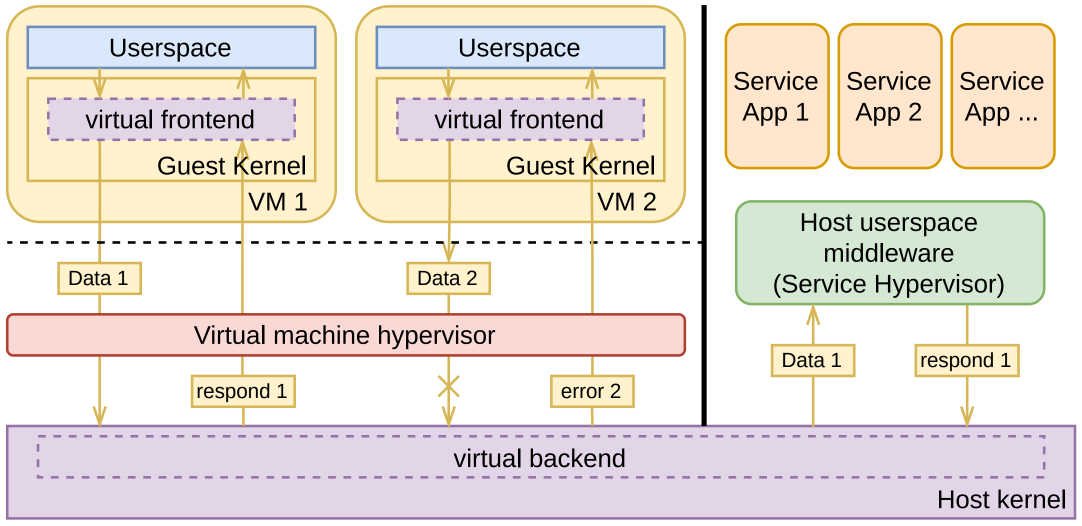


Figure 1. Architecture overview for the application virtualization framework.

2.3 Virtualization Restrictions

The traditional way to use a VM is to deploy the application directly inside the VM. The application can only access data in the VM. If applications need data from other applications, they should use host-to-host communication such as the network. This method has its own problems. Since virtualization needs to emulate a VM to operate on, it brings performance overhead to the whole system. This overhead is increased while running multiple VMs [15]. Also, hardware virtualization support is relatively poor for some architectures. For instance, VMs on the ARM platform suffer a performance loss of more than 50% compared to x86 platform [4]. Another problem is duplicate applications. If multiple VMs run the same service, the applications they use will need to exist in all these VMs, which results in wasted storage.

The reason for the high utilization achieved with virtualization is that hardware resources can be logically partitioned. But there are several differences between physical hardware and applications. The application needs to be not only a physical entity, but also a logical entity to provide full functionality. Therefore, unlike physical hardware, which is a physical entity but can be logically partitioned, applications cannot be logically partitioned directly. But the workload of applications are similar to the hardware and can be assigned to different programs as the combination of the physical resources occupied by the application.

To reduce the overhead and duplication associated with virtualization, the attention needs to be focused on the application itself rather than the hardware used by the VM [29]. Although compared to optimizing the hardware, optimizing the applications is less general. However, the performance overhead can be significantly reduced if the application execution process can be moved to the host OS. Meanwhile, program duplication can be reduced since all the VMs can share the application needed.

3 Application Virtualization Overview

In this section, the overview designed of application virtualization framework is proposed. This section begins at the overview of the framework, and followed by the feature and the detailed discussions in different field of virtualized, for instance, the security and migration.

3.1 Architecture Overview

The key idea of application virtualization framework is to provide a transparent, isolated and application-oriented data channel between the VM and the host, especially in environments with limited resource or lack of virtualization support. In order to achieve this idea, the following main problem should be resolved.

1. How to provide a pass-through data channel between the VM and the host?
2. How to cover the detail of the data channel in service application integration?
3. How to dynamically integrate service applications?
4. How to isolate the service applications and VMs from unauthorized access?

These problem are considered to cover both kernel and userspace of the OS. So the framework is not divided by functions, but by the location the component affected. The framework is divided into four components depending on the location it affect: host userspace component, kernel component, virtual machine hypervisor component and virtual machice userspace component. These components are shown in figure 1. The key functions of these components are summarized as follows.

- **host userspace component**: It acts as an service provider, scheduler and permission manager for other components in the application virtualization framework and provides generic interface for service applications to integrate with the framework at a low cost.
- **kernel component** It contains both the VM kernel component and the host kernel component. It is responsible for exchanging data effectively between the host and the VM and blocking unauthorized access from the VM. This component also exports interface for userspace to manage VM(only in the host), permission management and transfer data.
- **virtual machine hypervisor component** It registers the VM in the host kernel component and performs resource allocation and initialization.
- **virtual machice userspace component** It provides the same application interface of the service applications for programs inside the VM. Meanwhile, it convert the invocations from programs into payloads that can be transferred by the kernel component.

The operation of the application virtualization framework is across the entire VM and the host. The basic workflow of the framework is also shown in figure 1. There are two scenarios for the data to be processed: authorized or unauthorized (the VM 1 and the VM 2). For an authorized data, the data transmission begins at the userspace middleware in the VM, the middleware sends the data into kernel frontend. The kernel frontend indexes the metadata and packs the data into a packet. Then the frontend adds the data packet to the end of the send queue and notifies the data transmission structure to send the data. After sending data to the backend, the data packet is unpacked and the backend will check the packet whether is authorized. If not, the backend will issue an error to the frontend and drop the packet. Otherwise the data will be forwarded to the service hypervisor. According to the metadata of the packet, the service hypervisor selects the relevant application and invokes the application with the payload. After execution, the service hypervisor gets the result and returns the data to the frontend. The result data uses the same data path as that of the sending data. After VM receives the data, the user program can extract the data from the kernel frontend via the middleware in the VM.

The application virtualization framework is designed as an extension in the existed virtualization mechanisms. It does not have to depend on the hardware support. In fact, it does not process the data except packaging, instead, it offloads data and tasks to the host to get a considerable performance gain. This offload capability allows VM to run some platform dependent applications on the host.

Due to the the overall architecture, The application virtualization framework need a fully trusted and accessible host. Since the service applications are deployed in the host OS, a trusted and accessible host becomes a basic requirement. Another reason for a trusted and accessible host is the fact that the host kernel also needed to be modified, which means the host need to be accessed.

The main scenario for application virtualization framework is fully virtualized VM that lacks hardware virtualization support. Other scenarios for this framework can share the data or hardware between both the host and the VM with high bandwidth and low latency. At last, the container cannot benefit from application virtualization as it does not have a virtual middleware between the host and running applications.

3.2 Security

Security is a key concern in the VM. The security of a VM is mainly based on the isolation of the VM. The application virtualization framework also has its own way to guarantee this isolation. The isolation of the framework includes two topics: data transmission and application access.

The figure 1 also shows the three divided parts of a working instance. The userspace are divided into three parts: VM, VM hypervisor and service hypervisor and applications. The solid line shows VMs are fully isolated from the host userspace. And the dotted line means that a limited connection is allowed between the VM and the VM hypervisor. And the VM hypervisor can interact directly with the kernel backend. Meanwhile, the service applications can also communicate with the kernel backend through the service hypervisor. So applications in the VM can invoke service applications in the host. However, only valid and authenticated messages are allowed to be forwarded to the service applications, otherwise it will be rejected by the kernel backend. The authentication information for the data used by the kernel backend comes from the service hypervisor during the VM is initialized. If needed, this authentication can also be changed at runtime.

As direct data transmission between the host and the VM does reduce the isolation of the VM, the framework uses kernel as a wrapper to provide a barrier between the VM and the service application. As described above, the kernel backend drops all the invalid data at an early stage, which protects the host userspace from being polluted by invalid data. Since host userspace is not affected by the VM, a basic isolation is guaranteed.

Another concern is the service applications used by the VM. Since all the VMs share the same host and they may use the same application, it is unavoidable to isolate applications used by the service hypervisor. A common solution is to use multiple instances. Each VM has its own instances of the service. But if the service does not support multiple instances, the service hypervisor will try to partition the data areas into different virtual areas. Each VM can only access the data area it owns. These methods ensure all the VM in the framework can access data separately.

3.3 Migration

The migration of virtio-vapp can be partitioned into two parts: the one is preparations for migration and the other is the process of migration. The migration depends on stable connection and migratable application. The stable connection is needed to isolate the backend application and the guest. And the migratable application allows VMs to access the same application data after migration.

Connection Stability. The application virtualization framework needs to keep the data consistent during the migration. So the middleware should not communicate with the VM directly. Instead, it communicates with a wrapper in kernel space. The kernel forwards all the data from the middleware to the VM. This guarantees that the VM can still communicate with the same device after migration. Therefore, in order to maintain a stable connection interface, a virtual device is needs to be created in the host.

Migratable Applications. The application associated with the VM may not always be available if the VM performs a hot migration between different hosts. While the application can be installed in the new host, the application data of the VM needs to be migrated from the original host. Considering the fact the application in application virtualization is shared, it is necessary to hot-migrate or hot-backup the application data to the target host rather than migrating data by stopping the application. Another restriction on the application is that if the connection between application and middleware has a state, this state needs to be shared among middleware in a different host. Otherwise, the status of the connection between the VM and the application is connection and unrecoverable after migration.

Migration Process. The migration in both intra-host and different hosts requires the same pre-migration process and post-migration process. The figure 2 shows the detail about how the middleware performs an additional process on the VMs. The additional pre-process starts when the service hypervisor receives a signal from the VM hypervisor. Firstly, the hypervisor will tell the driver to lock the data transmission between the VM and the host. The driver then sends a stop message to the VM through control data channel. The guest locks all the data channel and the driver to stop data transmission after parsing the message. The guest kernel also informs the userspace that the transmission is temporarily stopped. After stopping all the data transmission in the VM, the guest sends feedback to the host and locks itself. After receiving feedback, the driver flushes all the data from the data channel into the VM. After flushing, the driver tells the middleware that the VM is ready to start migration. The post-process performs the same operations as pre-process, but unlocks all structures in the VM to resume data transmission.

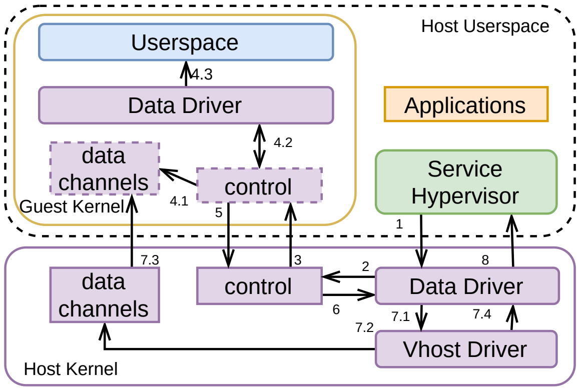


Figure 2. Addition pre/post process for hot migration

3.4 Comparisons with Container and VM

The application virtualization framework, container and the VM have differences in many fields. Meanwhile, they also share some similarities in depth. The table 1 shows the difference and similarities between them.

The main difference between the container and the VM is that container does not need a separate OS. VMs have a completely virtual environment. The application virtualization framework, basing on the VM, also needs this virtual environment. So it is necessary for the VM and the framework to have an OS to provide basic functionalities. As for applications, the container and the VM have the same situation as for OS running environment. However, the framework can make applications run on the host. This means the virtualization level of application virtualization is above the VM and below the container. The underlying virtualization method of the application virtualization is emulation and proxy, which is also different from that of the container and the VM. In summary, the difference between the container and the VM comes from the fact that the VM has an OS. And the difference between the VM and the application virtualization framework comes from the fact that the framework has a special proxy between the host and the VM.

In addition to the difference, these virtualization mechanisms have the same security model and resource restriction method. Since all the things in the VM or the container is processes, the isolation of these mechanisms is based on the process isolation. The resource restriction also shares the same base point with the isolation. This reveals the essence of virtualization as processes in the eyes of the host OS.

4 Implementation

In this section, the generic implementation of data structures in application virtualization is introduced. In the end, the implementation of virtio-vapp will be shown as an example of adapting application virtualization to a hypervisor and an operating system.

Table 1. The similarities and differences between application virtualization, container and VM

	Container	VM	Application virtualization
OS running environment	–	virtual	virtual
App running environment	native	virtual	virtual / native
Allow different OS	No	Yes	Yes
Virtualization level	OS level	physical level	hybrid level
Underlying methods	namespace	emulation	emulation / proxy
Inter-communication methods	IPC / Network	hypercall / Network	IPC / Network / hypercall
Persistent Storage	container image	virtual disk	virtual disk & data in the host
Security model		process isolation	
Resource restriction		cgroup / process restriction	

4.1 Generic data structure

Data Transmission Structure (DTS). The communication between VMs and host OS needs to be abstracted to support different virtualization methods. The data transmission structure (DTS) is a basic abstract structure for masking details of communication between guest VMs and host OS. It is hard to describe the implementation detail of DTS since it depends on the specific virtualization method.

DTS exists in pairs in the host and VMs. The bandwidth can be boosted by using multiple DTSs, or be reduced by removing several DTSs. In addition to handling data transmission, the DTS is also used to exchange control messages between the host and the guest.

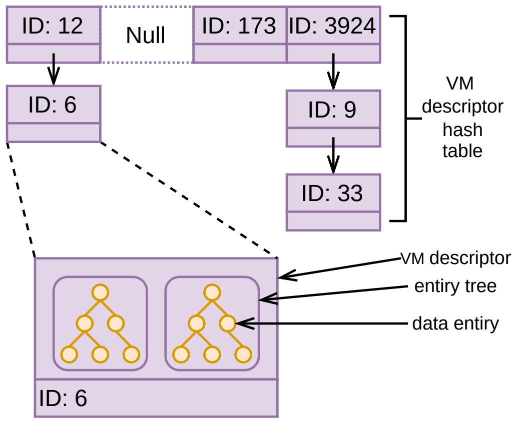


Figure 3. The organization of transmission entities in the kernel and VMs.

Transmission Entity and Entity Tree. The transmission entity contains different fields that carry out the functionalities of the entity, including identity, ownership, payload and status. The identity is used to uniquely locate a transmission entity in the data driver. To avoid conflicts, this identity is assigned globally in the VM. The ownership indicates whether this entity is created in the guest or in the host. The payload contains data from userspace. Lastly, the status is used as a hint of the process when the application virtualization framework processes the entity.

Transmission entities are organized in the framework as an index tree called entity tree to provide fast access. The entity tree also has a spinlock to provide low wait concurrency control. Each VM descriptor has two entity trees, one for entities from VM and the other for those from the host. And multiple VM descriptors form a chaining hash table. This nested structure is shown in the figure 3.

All operations on transmission entities can be divided into 3 categories: insertion, deletion and search. All these operations use the structure described before as the entry point. The detailed steps for these operations are shown in algorithm 1. These operations share the action of querying entity ownership. And the rest of the operations on the entity tree are converted into the operations on the index tree with entity identity as the index.

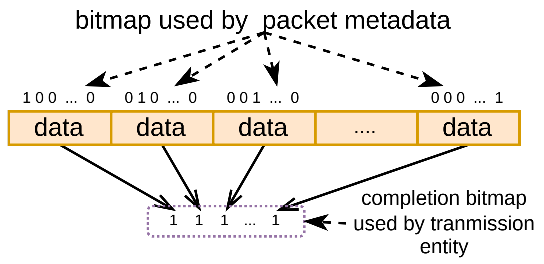


Figure 4. The relationship between the two computed bitmaps in Algorithm 2.

Packet Assembling and Splitting. A bitmap is needed as a hint for packet operations. The bitmap provides a computable and comparable identification for packets with different data offsets. The completion bitmap for each transmission entity is a fixed length and contains the same number of bits *1* as the number of packets since the completion bitmap is the *OR* of the bitmaps in all packets. The details of the calculation are shown in algorithm 2. The bitmap of a packet is calculated based on the offset of the packet payload in the

Algorithm 1 The insertion, search, and deletion of a data entity in the data driver

Input: E: the entity to insert; Id_{vm}: the identity of VM; Id_E: the identity of entity; T_{global}: the global hash table with all VM descriptors;
Output: $Error$: The execution status of the function

```
 1: function GetEntityTree($T_{global}, Id_{vm}, E$)
 2:     $Tree \leftarrow NULL$
 3:     $Pos_{vm} \leftarrow Hash(Id_{vm})$
 4:     Lock $T_{global}$
 5:     if $T_{global}.contains(Pos_{vm})$ then
 6:         $Desc_{vm} \leftarrow T_{global}.get(Pos_{vm})$
 7:         $Tree \leftarrow Desc_{vm}.getTreeByType(E.ownership)$
 8:     end if
 9:     Unlock $T_{global}$
10:     return $Tree$
11: end function
12: function EntityInsert($T_{global}, Id_{vm}, E$)
13:     $Error \leftarrow OK$
14:     $Tree \leftarrow GetEntityTree(T_{global}, Id_{vm}, E)$
15:     if $Tree \neq NULL$ then
16:         Lock $Tree$
17:         $E_{tree} \leftarrow Tree.get(E.id)$
18:         if $E_{tree} = NULL$ then
19:             $Tree.insert(E)$
20:         else
21:             $Error \leftarrow$ Entity exists
22:         end if
23:         Unlock $Tree$
24:     else
25:         $Error \leftarrow$ No descriptor
26:     end if
27:     return $Error$
28: end function
29: function EntitySearch($T_{global}, Id_{vm}, Id_E$)
30:     $Tree \leftarrow GetEntityTree(T_{global}, Id_{vm}, E)$
31:     if $Tree \neq NULL$ then
32:         Lock $Tree$
33:         $E_{tree} \leftarrow Tree.get(E.id)$
34:         Unlock $Tree$
35:     end if
36:     return $E_{tree}$
37: end function
38: function EntityDelete($T_{global}, Id_{vm}, Id_E$)
39:     $Tree \leftarrow GetEntityTree(T_{global}, Id_{vm}, E)$
40:     if $Tree \neq NULL$ then
41:         Lock $Tree$
42:         if $Tree.get(Id_E)$ then
43:             $Tree.delete(E.id)$
44:         end if
45:         Unlock $Tree$
46:     end if
47: end function
```

Algorithm 2 Calculate the completion bitmap based on entity splitting or payload offset

Input: L_{entity}: the length of entity payload; L_{pkt}: the length of packet payload;
Output: M_c: completion bitmap in entity;

```
1: function CalculateByEntity
2:     Init all bit in $M_c$ to 0
3:     $NR_{pkt} \leftarrow \lceil L_{entity} \div L_{pkt} \rceil$
4:     for $i \leftarrow 1$ to $NR_{pkt}$ do
5:         Set the $i$ bits of $M_c$
6:     end for
7: end function
```

Input: L_{entity}: the length of entity payload; Off_{pkt}: the offset of packet payload;
Output: M_p: bitmap in packet;

```
1: function CalculateByPayload
2:     Init all bit in $M_p$ to 0
3:     $Pos_{pkt} \leftarrow \lfloor L_{entity} \div Off_{pkt} \rfloor + 1$
4:     Set the $Pos_{pkt}$ bits of $M_p$
5: end function
```

entity, which is also shown in algorithm 2. The position of *1* in the bitmap is determined by the position of the packet in all packets, which is shown in the figure 4.

Packet operations include 2 parts: splitting and assembling. The splitting process is completed in the data driver. The driver allocates several data packets according to the length of the transmission entity. After allocation, the driver calculated the completion bitmap and assigns the metadata of each packet. In addition to the completion bitmap, the packet header includes the offset and length of the packet payload. The packet payload is a pointer to the data in the transmission entity with the offset in the metadata. The splitting process is finished after the driver assigns value to these structures. The assembling process is handled by the DTS. Firstly, DTS reads the packet metadata. Then the DTS uses the metadata to locate the offset of the packet payload in the transmission entity. After locating the offset, the DTS read the packet payload and write it into the entity directly. The DTS performs OR operation on the bitmap of the packet and the completion bitmap of the entity, and writes the result to the bitmap in the entity. Lastly, the DTS compares the completion bitmap in the packet with that in the entity to determine whether the transmission is complete.

Send Queue. The send queue is a first-in-first-out (FIFO) queue used to buffer packets from transmission entities. The send queue is implemented as a doubly linked list whose nodes are sorted in chronological order. Since all operations on send queue are simple, the concurrency control of send queue is provided by a spin lock.

The interaction between send queue and the other structures consists of 3 types of operations: push, pop and error

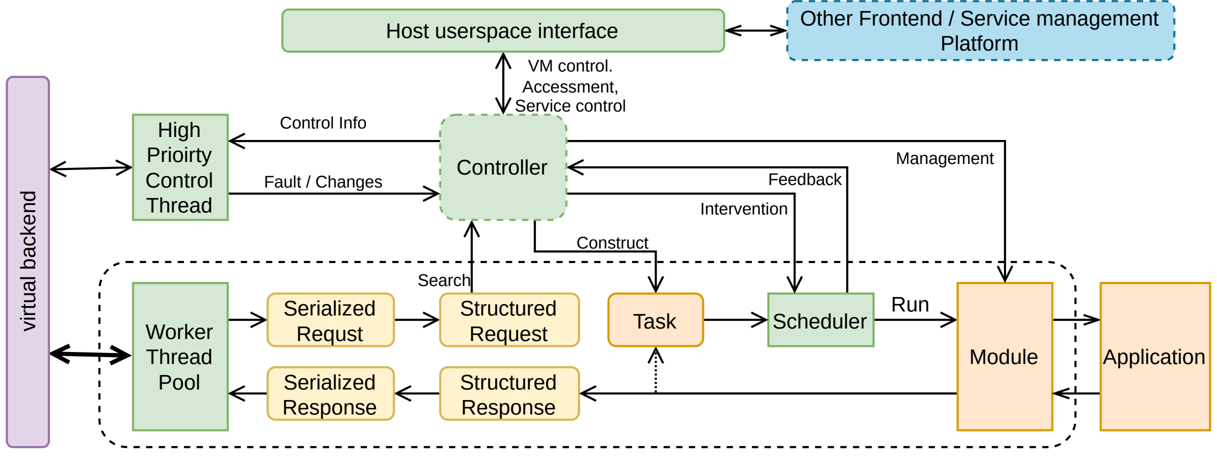


Figure 5. The detailed design of service hypervisor

recovery. The push operation is to add data packets from the transmission entity to the send queue. Since each transmission needs to cover the entire entity, all packets from the same entity need to be inserted into the send queue in a single operation. The data driver first organizes all the packets into a chain, and then appends this chain to the end of the send queue. The pop operation is just pop a packet from the front of the send queue. The error recovery occurs when the DTS is unable to send the packet. When the buffer of DTS is full or an error occurs, the packet to be sent is handed over to the send queue for the next transmission. Considering that the packets need to keep time order in the send queue, the DTS links the packets to the head of the send queue.

Service Hypervisor. The service hypervisor is the main part of the host userspace component. The figure 5 shows the three subcomponents of the service hypervisor: the worker thread pool, the controller and the scheduler.

The worker thread pool allocates threads to handle data packets from the kernel backend. There are two types of threads that are allocated by the pool: control threads and worker threads. As the figure 5 shows, the control threads read the control data, including fault and revoke request, from the backend. The control thread is also the entry point for the controller to deliver control information to the backend. The worker thread takes the execution of service application. The worker thread deserializes the data packet and forwards its payload to the application. The control threads are designed to run at high prioirty and the work threads at normal prioirty or low prioirty. This ensure the controller can interfere with the entire process once the control information is available.

The controller handles all the error during the transmission of the data packet from control thread. It also performs the packet revocation request from the frontend. For other userspace manager program, the communication interface of the service hypervisor is exported by the controller, and the controller takes all the operations it needs to respond. These operations include service control, VM accessment, permission management and migration. In addition to maintaining control information, the controller also constructs task struct based on requests from VMs and indexes them. The task scheduler is also interfered with by the controller.

The scheduler is responsible for scheduling tasks from the controller. In case of no controller intervention, the scheduler follows first-in-first-out (FIFO) to invoke tasks from the same program of the VM, and for tasks from different programs it follows the process management of the host OS. In case of having information from the controller, the scheduler will delay, promote, suspend, cancel, or terminate related tasks based on the information from the controller.

To implement virtio-vapp with Linux and Qemu, some generic design of application virtualization needs to adapt the Linux. In Linux, the virtio-vapp is divided into two modules: *vhost_vapp* in the host OS and *virtio_vapp* in the guest OS. In addition to the modules in the kernel, virtio-vapp also contains a daemon called virtual application hypervisor (VAH), which acts as the service hypervisor in the host. The whole process of virtio-vapp is shown in the figure 6.

4.2 Linux-based implementation

Qemu Implementation. The virtio-vapp implements the *virtio_vapp* device in the Qemu. As the figure 6 shows, the *virtio_vapp* device does not implement any core logic other than providing a basic virtual device. This device is used to register VM into the application virtualization framework and provide a hint to *vhost_vapp* for allocating resources.

vhost Driver Implementation. The *vhost_vapp* module includes a vhost driver, a data driver and multiple DTSs for data transmission. The vhost driver is mainly used to handle control commands related to VMs and *virtio_vapp* devices.

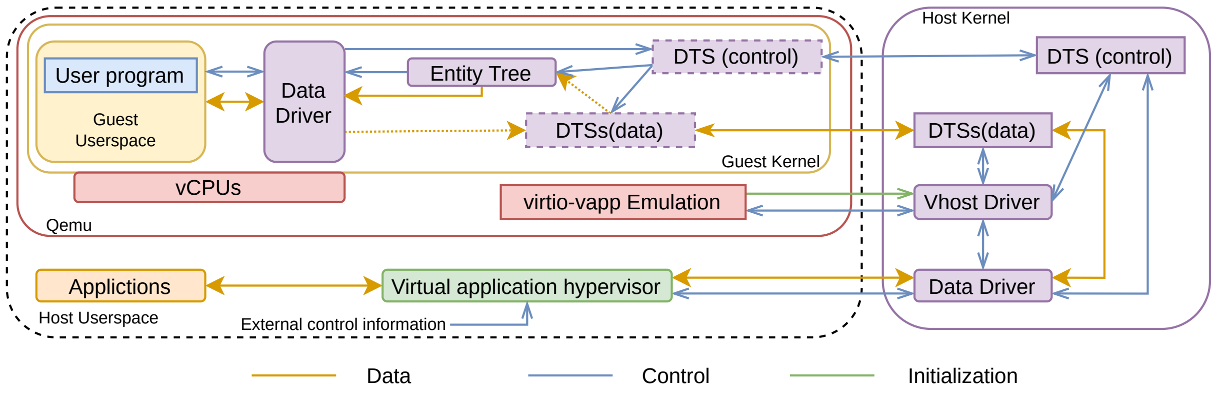


Figure 6. virtio-vapp with linux and Qemu

For instance, the dynamic allocation of DTS is initialized by the VIAPP device, which is handled by the vhost driver. The data driver is a flow balancer that balances requests from VMs. Meanwhile, the data driver exchanges data about virtualization requests with VAH. The DTS in *vhost_vapp* uses *vhost_virtqueue* as the transmission medium. The data DTS uses 65536 bytes as packet size to avoid a large number of packets for one entity. And the payload of the transmission entity is set to 4 MiB to ensure that the completion bitmap can be guaranteed to be operated atomically.

virtio Driver Implementation. The *virtio_vapp* has only one data driver in the userspace, as the guest userspace has no permission to send control commands to the application virtualization framework. Since the source of all requests is userspace, the entity tree is maintained by file descriptors opened in the userspace. However, the identity is allocated globally in *virtio_vapp* to avoid conflicts. The DTS in *virtio_vapp* uses *virtqueue* as the transmission medium to communicate with *vhost_vapp*. Since the *virtqueue* needs to be pre-filled with data for the host to transmit, The *virtqueue* will refill the packets after the packets in the queue are less than one-third of the total to guarantee the stability of the transmission.

Host Userspace Implementation. The VAH is a detailed adapting of the service hypervisor described in section 4.1. As shown in the figure 6, the external programs use signal messages to interact with the VAH. THe VAH itself can only interact with service applications and the data driver of the kernel component. The integration is handled as described above, using an shared library as an module to interact with service applications.

5 Experiments and Results

The implementation of virtio-vapp is evaluated with Qemu, Linux on x86_64 architecture. virtio-vapp compares bandwidth with different communication methods and in different mediums. It also compares the performance overhead of applications running in VMs with different mediums.

5.1 Environment setup

All experiments were performed on a 16-core Intel Xeon Silver 4123 CPU with 3.00 GHz and 32 GiB of DDR4.

In the hypervisor, we ran Arch Linux (date 20211206) with the supplied Linux kernel 5.15.6-arch2-1.x86_64 and a modified QEMU (6.2.0-rc0-21-g70f872ca91) as the hypervisor, compiled with "gcc (GCC) 11.1.0 20210513 (Arch Linux 11.1.0-1)". Inside the guest VMs, we ran Arch Linux (20211206) with a Linux kernel 5.15.6-arch2-1. All the guest VMs have 4 GiB memory and 8 host core emulated by the KVM. All the VM use multiple qcow2 images on the same SSD that use btrfs as the underlying filesystem. All the VM use writeback caching policy for the underlying mediums. Both vhost and virtio kernel modules used by the experiments are configured based on the kernel config 5.15.6-arch2-1.x86_64s supplied by Arch Linux package linux-headers, compiled with The "gcc (GCC) 11.1.0 20210513 (Arch Linux 11.1.0-1)".

The iperf used to measure bandwidth is provided by the Arch Linux (2.0.13). And the database programs for the host OS and the guest OS is mariadb 10.6.5-1 provided by Arch Linux. The database files are stored in a separate SSD which does not store virtual machine images.

5.2 Experiments

There are two experiments in this section: the bandwidth and concurrency test and the I/O test. The bandwidth and concurrency experiments test the bandwidth and the load of the application virtualization framework in a high concurrency user case. The I/O experiments test the latency of I/O operations with the large number of I/O requests.

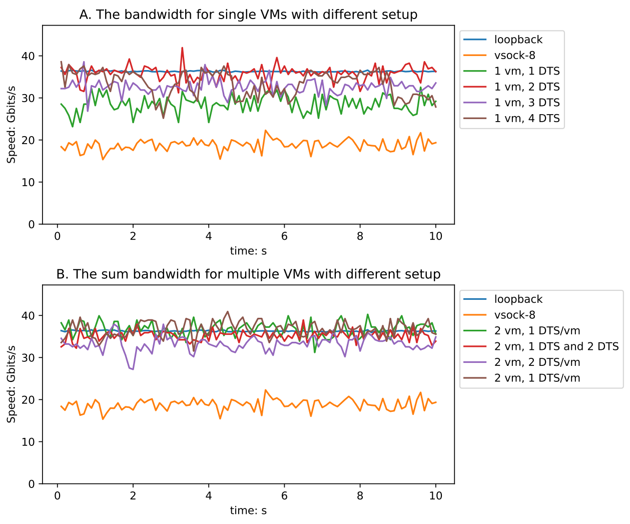


Figure 7. The sum bandwidth for different setup methods

5.2.1 Bandwidth and concurrency test.

In general, the virtio-vapp requires large data bandwidth and low transmission latency like other vhost drivers. So the bandwidth test is provided firstly to evaluate the I/O workload capacity of virtio-vapp as application virtualization is accompanied by heavy I/O requests.

Setup. The iperf program is used to simulate the request bandwidth of the loopback and *vhost_vsock*. To test the bandwidth and concurrency capability of the virtio-vapp, a multithread producer-consumer program is deployed in both the guest and the host. The program can generate request packets to simulate the large number of requests that an service (e.g., web server) needs to handle.

Result. The workload of both the hosts and the VM are flat. The workload of the host is 100% cpu usage in all cases. The workload of the guest is also nearly the same in all cases. The only difference is that the cpu usage of kernel in virtio-vapp is about 20% higher than that in *vhost_vsock* and loopback. As virtio-vapp uses multiple kernel threads for parallel transfers, the CPU weight in the kernel is greater than in other methods.

As for the bandwidth, the results are different. The figure 7 showns the bandwidth within different scenarios. The bandwidth of loopback is 36.295$Gbit/s$, and the bandwidth of *vhost_vsock* is 18.818$Gbit/s$. In the case where virtio-vapp is applied, the worst-case bandwidth is 28.564$Gbit/s$, which is still 51.80% higher than that of *vhost_vsock*. The best case is reached when there are 2 DTSs in a VM with a bandwith of 35.829$Gbit/s$, which is only 1.29% lower than that of loopback. So virtio-vapp can be used to isolate the communication between VMs and LAN when VMs have high bandwidth demand for LAN.

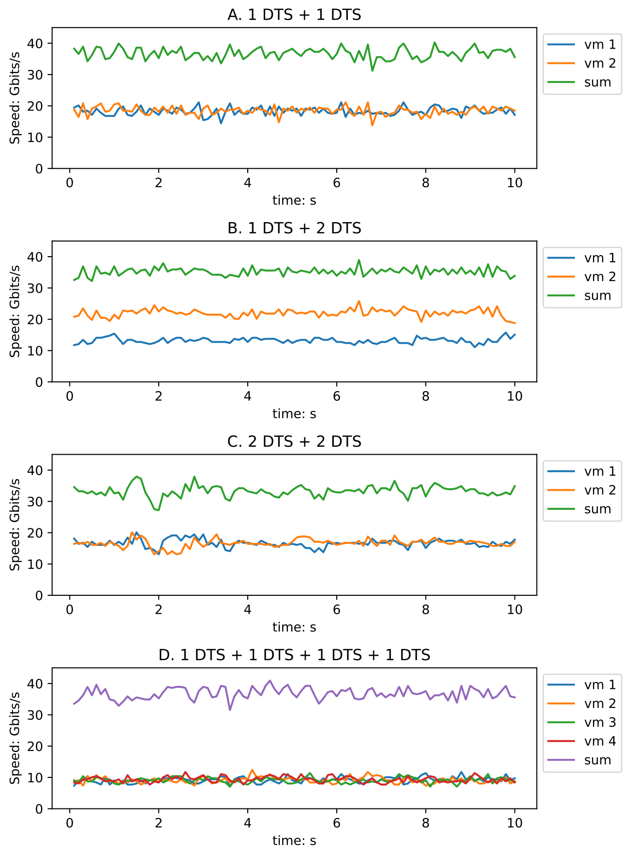


Figure 8. The detailed bandwidth for multiple VM with different configurations

It is clear that VM can reach the loopback bandwidth with 2 DTSs, but when there are more than 2 DTSs, the bandwidth of virtio-vapp decreases. The bandwidth is 32.598$Gbit/s$ with 3 DTS and 33.406$Gbit/s$ with 4 DTS, which is 9.02% and 6.77% lower respectively compared to the case with 2 DTS. In thess case, the usage of CPU is still high, which shows a lot of CPU time is wasted on data synchronization.

The subfigure B in the figure 7 shows the total bandwidth when multiple VM is invoked. It is not surprising that the bandwidth of the local loopback is reached in almost all cases. The only case whose bandwidth is slightly lower than loopback is that use 2 VMs with 2 DTSs each VM, which is 8.30% lower than the loopback. These equivalence suggests that the maximum process capability of the virtio-vapp backend is the loopback bandwidth. This capability suggests that not all the VM should use virtio-vapp if all the VMs on a single physical machine have relaxed resource constraints.

The figure 8 shows the detailed bandwidth of total bandwidth is the subfigure B in figure 7. These four subfigures also show that the main limitation of processing capability comes from the backend, especially when comparing subfigure A and subfigure D. In adding, the subfigure B shows the different numbers of DTS ratio results in different shares of the bandwidth for VMs. And the subfigure A and C show multiple DTSs in the same VM can have up to 5% synchronous overhead.

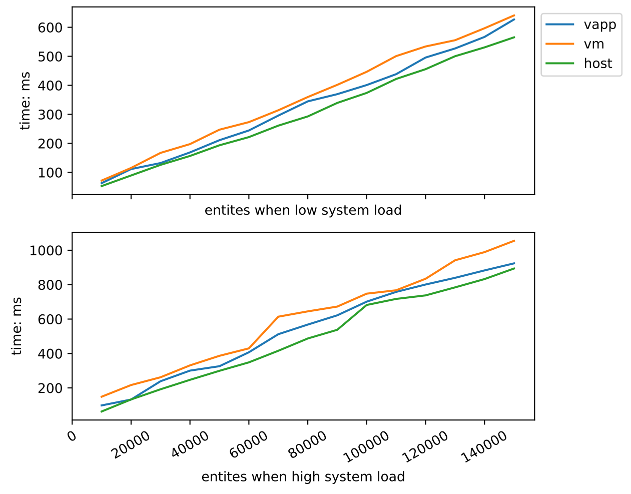
The stability of the transmission needs attention in addition to the bandwidth and concurrency. The variance of loopback bandwidth is just 0.017. And the variance of *vhost_vsock* is 1.672. When it comes to using virtio-vapp, the situation is different. All VMs with more than 2 DTS have a bandwidth variance greater than 2.00, which means significant performance fluctuations. If a VM only has one DTS, the variance is at most 1.5. Also, if the same four VMs with 1 DTS as in subfigure D in the figure 8 are used, the variance can be reduced to less than 1. Since VMs with virtio-vapp is not expected to have many resources, this performance fluctuation is acceptable.

5.2.2 I/O test. After confirming the workload and concurrency capabilities of the virtio-vapp, it is necessary to test the virtio-vapp in other forms, like I/O capability. In this section, a practical situation is used with the real program and compare it with the situation where it is executed only inside the VM with different system workload.

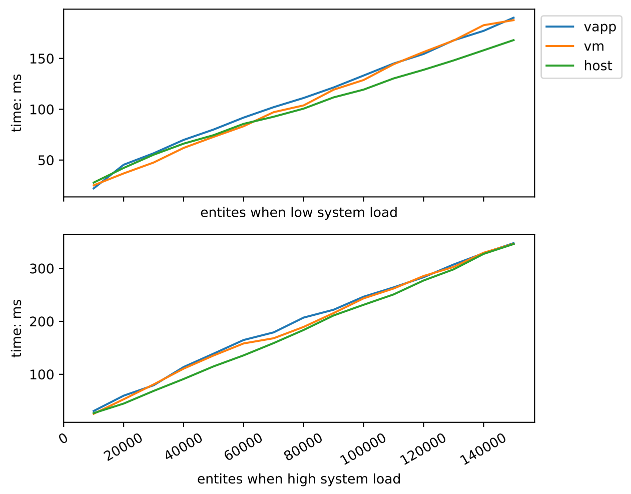
Setup. Each VM has 1 DTS, 2 vCPUs and 2 GiB RAM and a 16 GiB qcow2 images. These qcow2 images has the same context and are on the same disks. And each VM has its own backend database instance to operate on. The data of these database instances are stored on the same disk.

The tests include two parts: writing and reading, which is measured by the executon time of releated SQL. These two tests use the same database table with a single entity of 20 bytes. And the number of entities is from 10,000 to 150,000 with 10,000 as the interval in both the write and read test. The test is executed both on the host OS, the VM and inside the virtio-vapp with low and high system load. The high system load is achieved by running this test simultaneously in each VM. Filesystem cache is cleared to avoid possible impact of database cache.

Result. The figure 9 shows the cold execution time in both write test and read test. In all low system load cases, the execution time consumed to execute SQL in the host OS is minimal in all cases, which is 1.266 $\mu s/entity$ for reading and 3.816 $\mu s/entity$ for writing in average. And executing SQL just inside the VM has almost the worst performance, which is 1.346 $\mu s/entity$ for reading and 4.517 $\mu s/entity$ for writing in average. The performance of virtio-vapp is between the two, except for reading data. The average is 1.389 $\mu s/entity$ for reading and 4.163 $\mu s/entity$ for writing. The small reading performance penalty comes from the disk cache is provide by the VM hypervisor, which is hard to drop. Another thing that causes this difference is that the data inside the VM is inside the VM image, and there is a data locality bonus when



(a) "INSERT" SQL



(b) "SELECT" SQL

Figure 9. SQL cold execution time in different system load width in VMs, virtio-vapp and host OS

VM reads it. However, as the amount of data read increases, this penalty will gradually decrease, allowing VAPP to gain some performance advantage.

If the system load is high, the performance of virtio-vapp is closer to that of the host OS. As the figure 9 shows, the reading performance is almost the same. But the writing performance is significantly different. The average writing performance is 2.304 $\mu s/entity$ for the host OS, 7.533 $\mu s/entity$ for the VM and 6.760 $\mu s/entity$ for virtio-vapp. Although the performance of virtio-vapp is only one third compared to execution within the host, it still has a 10% performance improvement compared to execution in the VM. If data volume is large, the virtio-vapp can save up more than 15% write time. This means the VM can save enough time if it needs a large amount of data to write.

6 Conclusions

Virtualization has become a key importance to improve the utilization of the physical hardware in cloud computing. The virtual machine also has its own place in the virtualization. However, the VM has a serious performance overhead and storage duplication, and it is also hard to share data across VMs. To overcome these problems, the application virtualization is introduced to focus on applications rather than hardware in the virtual machine. And it provides a new channel between the VM and the host. The application virtualization framework has four components as host userspace component, kernel component, virtual machine hypervisor component, and virtual machine userspace component. It uses data transmission structures as abstract data channels to achieve direct data transfer between the host and the VM. For the management of the host applications, the framework introduces the service hypervisor as the entry point and management platform for the service applications. As for the migration, the framework can achieve restricted migration with the use of OS kernel as a relay agent. Then, virtio-vapp is virtio-based implementation and focuses on high bandwidth and low latency I/O in addition to all functionalities of application virtualization. Based on the evaluation of virtio-vapp in this paper, the application virtualization mechanism can provide significant performance improvement while preserving availability and usability, with a bandwidth of 36.4 $Gbit/s$ and a transmission delay of 60 μs.

In the future, it is necessary to extend a new access path for virtio-vapp to allow virtual machines to access these data files directly. While programs in the guest virtual machine can access the application data in the host through virtio-vapp, the data files generated by the application are not directly accessible by the virtual machine. In addition, effectively management of all the transmission entities on the same entity tree in the kernel space is still challenging. A new buffering mechanism needed to be applied to the management to avoid the performance penalty caused by this parallel access.

References

[1] Dmytro Ageyev, Oleg Bondarenko, Tamara Radivilova, and Walla Alfroukh. 2018. Classification of existing virtualization methods used in telecommunication networks. In *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. 83–86. https://doi.org/10.1109/DESSERT.2018.8409104

[2] Nilton Bila, Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Matti Hiltunen, and Mahadev Satyanarayanan. 2012. Jettison: Efficient Idle Desktop Consolidation with Partial VM Migration. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 211–224. https://doi.org/10.1145/2168836.2168858

[3] Igor Burdonov, Alexander Kosachev, and Pavel Iakovenko. 2009. Virtualization-Based Separation of Privilege: Working with Sensitive Data in Untrusted Environment. In *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems* (Nuremberg, Germany) *(VDTS '09)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/1518684.1518685

[4] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, and Jason Nieh. 2018. ARM Virtualization: Performance and Architectural Implications. *SIGOPS Oper. Syst. Rev.* 52, 1 (aug 2018), 45–56. https://doi.org/10.1145/3273982.3273987

[5] Ulrich Drepper. 2008. The Cost of Virtualization: Software Developers Need to Be Aware of the Compromises They Face When Using Virtualization Technology. *Queue* 6, 1 (jan 2008), 28–35. https://doi.org/10.1145/1348583.1348591

[6] Sahan Gamage, Ramana Rao Kompella, Dongyan Xu, and Ardalan Kangarlou. 2013. Protocol Responsibility Offloading to Improve TCP Throughput in Virtualized Environments. *ACM Trans. Comput. Syst.* 31, 3, Article 7 (aug 2013), 34 pages. https://doi.org/10.1145/2491463

[7] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. 2020. MasQ: RDMA for Virtual Private Cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3387514.3405849

[8] David Hildenbrand and Martin Schulz. 2021. Virtio-Mem: Paravirtualized Memory Hot(Un)Plug. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Virtual, USA) *(VEE 2021)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3453933.3454010

[9] DeokGi Hong, Jaemin Shin, Shinae Woo, and Sue Moon. 2017. Considerations on Deploying High-Performance Container-Based NFV. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking* (Incheon, Republic of Korea) *(CAN '17)*. Association for Computing Machinery, New York, NY, USA, 1–6.

[10] Jaeseong Im, Jongyul Kim, Jonguk Kim, Seongwook Jin, and Seungryoul Maeng. 2017. On-Demand Virtualization for Live Migration in Bare Metal Cloud. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 378–389. https://doi.org/10.1145/3127479.3129254

[11] Liang Liang, Rong Chen, Haibo Chen, Yubin Xia, KwanJong Park, Binyu Zang, and Haibing Guan. 2016. A Case for Virtualizing Persistent Memory. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 126–140. https://doi.org/10.1145/2987550.2987551

[12] Jin Tack Lim and Jason Nieh. 2020. Optimizing Nested Virtualization Performance Using Direct Virtual Hardware. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 557–574. https://doi.org/10.1145/3373376.3378467

[13] Jiacheng Ma, Xiao Zheng, Yaozu Dong, Wentai Li, Zhengwei Qi, Bingsheng He, and Haibing Guan. 2018. GMig: Efficient GPU Live Migration Optimized by Software Dirty Page for Full Virtualization. *SIGPLAN Not.* 53, 3 (mar 2018), 31–44. https://doi.org/10.1145/3296975.3186414

[14] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. 2007. Quantifying the Performance Isolation Properties of Virtualization Systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (San Diego, California) *(ExpCS '07)*.

Association for Computing Machinery, New York, NY, USA, 6–es. https://doi.org/10.1145/1281700.1281706

[15] Richard McDougall and Jennifer Anderson. 2010. Virtualization Performance: Perspectives and Challenges Ahead. *SIGOPS Oper. Syst. Rev.* 44, 4 (dec 2010), 40–56. https://doi.org/10.1145/1899928.1899933

[16] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. 2010. Efficient Resource Provisioning in Compute Clouds via VM Multiplexing. In *Proceedings of the 7th International Conference on Autonomic Computing* (Washington, DC, USA) *(ICAC '10)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/1809049.1809052

[17] Yoshihiro Nakajima, Hitoshi Masutani, and Hirokazu Takahashi. 2015. High-Performance vNIC Framework for Hypervisor-Based NFV with Userspace vSwitch. In *2015 Fourth European Workshop on Software Defined Networks*. 43–48. https://doi.org/10.1109/EWSDN.2015.59

[18] Dac Nguyen, Quy H. Nguyen, Minh-Son Dao, Duc-Tien Dang-Nguyen, Cathal Gurrin, and Binh T. Nguyen. 2020. Duplicate Identification Algorithms in SaaS Platforms. In *Proceedings of the 2020 on Intelligent Cross-Data Analysis and Retrieval Workshop* (Dublin, Ireland) *(ICDAR '20)*. Association for Computing Machinery, New York, NY, USA, 33–38. https://doi.org/10.1145/3379174.3392319

[19] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. 2009. Autonomic Virtual Resource Management for Service Hosting Platforms. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing (CLOUD '09)*. IEEE Computer Society, USA, 1–8. https://doi.org/10.1109/CLOUD.2009.5071526

[20] Audun Nordal, Åge Kvalnes, and Dag Johansen. 2012. Paravirtualizing TCP. In *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing Date* (Delft, The Netherlands) *(VTDC '12)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/2287056.2287060

[21] Javier Prades and Federico Silla. 2018. Made-to-Measure GPUs on Virtual Machines with RCUDA. In *Proceedings of the 47th International Conference on Parallel Processing Companion* (Eugene, OR, USA) *(ICPP '18)*. Association for Computing Machinery, New York, NY, USA, Article 19, 8 pages. https://doi.org/10.1145/3229710.3229741

[22] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. 2013. Speeding up Packet I/O in Virtual Machines. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (San Jose, California, USA) *(ANCS '13)*. IEEE Press, 47–58.

[23] Rusty Russell. 2008. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.* 42, 5 (jul 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[24] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. *SIGOPS Oper. Syst. Rev.* 41, 3 (mar 2007), 275–287. https://doi.org/10.1145/1272998.1273025

[25] Gaurav Somani and Sanjay Chaudhary. 2009. Application Performance Isolation in Virtualization. In *2009 IEEE International Conference on Cloud Computing*. 41–48. https://doi.org/10.1109/CLOUD.2009.78

[26] Xiang Song, Jicheng Shi, Ran Liu, Jian Yang, and Haibo Chen. 2013. Parallelizing Live Migration of Virtual Machines. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Houston, Texas, USA) *(VEE '13)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2451512.2451531

[27] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2016. GPUvm: GPU Virtualization at the Hypervisor. *IEEE Trans. Comput.* 65, 9 (sep 2016), 2752–2766. https://doi.org/10.1109/TC.2015.2506582

[28] Jianfeng Tan, Cunming Liang, Huawei Xie, Qian Xu, Jiayu Hu, Heqing Zhu, and Yuanhan Liu. 2017. VIRTIO-USER: A New Versatile Channel for Kernel-Bypass Networks. In *Proceedings of the Workshop on Kernel-Bypass Networks* (Los Angeles, CA, USA) *(KBNets '17)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3098583.3098586

[29] Muhammad Shams Ul Haq, Lejian Liao, and Lerong Ma. 2016. Transitioning Native Application into Virtual Machine by Using Hardware Virtualization Extensions. In *2016 International Symposium on Computer, Consumer and Control (IS3C)*. 397–403. https://doi.org/10.1109/IS3C.2016.108

[30] Dongyang Wang, Binzhang Fu, Gang Lu, Kun Tan, and Bei Hua. 2019. VSocket: Virtual Socket Interface for RDMA in Public Clouds. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 179–192. https://doi.org/10.1145/3313808.3313813

[31] Zhen Xiao, Weijia Song, and Qi Chen. 2013. Dynamic Resource Allocation Using Virtual Machines for Cloud Computing Environment. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1107–1117. https://doi.org/10.1109/TPDS.2012.283

[32] Minhoon Yi, Dong Hyun Kang, Minho Lee, Inhyeok Kim, and Young Ik Eom. 2016. Performance Analyses of Duplicated I/O Stack in Virtualization Environment. In *Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication* (Danang, Viet Nam) *(IMCOM '16)*. Association for Computing Machinery, New York, NY, USA, Article 26, 6 pages. https://doi.org/10.1145/2857546.2857573