Enhancing Performance and Energy Efficiency for Hybrid Workloads in Virtualized Cloud Environment

Chi Xu, *Student Member, IEEE*, Xiaoqiang Ma, *Member, IEEE*, Ryan Shea, *Member, IEEE*, Haiyang Wang, *Member, IEEE*, and Jiangchuan Liu, *Fellow, IEEE*

Abstract—Virtualization has attained mainstream status in enterprise IT industry. Despite its widespread adoption, it is known that virtualization also introduces non-trivial overhead when tasks are executed on a virtual machine (VM). In particular, a combined effect from device virtualization overhead and CPU scheduling latency can cause performance degradation when computation intensive tasks and I/O intensive tasks are co-located on a VM. Such an interference also causes extra energy consumption. In this paper, we present *Hylics*, a novel solution that enables efficient data traverse paths for both I/O and computation intensive workloads. This is achieved with the provision of in-memory file system and network service at the hypervisor level. Several important design issues are pinpointed and addressed during our prototype implementation, including efficient intermediate data sharing, network service offloading, and QoS-aware memory usage management. Based on our real-world deployment on KVM, we show that Hylics can significantly improve computation and I/O performance for hybrid workloads. Moreover, this design also alleviates the existing virtualization overhead and naturally optimizes the overall energy efficiency.

Index Terms—Platform virtualization, virtual machine monitors, network interfaces, middleware, load management

1 Introduction

The past decade has seen a great paradigm shift to cloud computing in the IT industry. The active participation of such major IT companies as Amazon, Google, and Microsoft significantly stimulates the prosperity of this new generation of service model. These cloud services leverage virtualization to achieve high resource utilization as well as performance isolation among co-located virtual machines (VMs). Despite the widespread adoption, it is known that existing virtualization technologies, such as Xen and KVM, also introduce non-trivial overhead due to the hypervisor interception when tasks are executed on a VM [1]. This leads to longer and unstable task completion time for computation-intensive applications. Moreover, such an overhead also causes *self interference* [2] for *hybrid workloads* that involve both computation and I/O intensive tasks. Different from cross-VM interference,[1] self interference happens within a VM when the I/O handling process of the VM is interfered or even starved by other processes inside the VM. This is very common when the co-located computation processes aggressively use the CPU resources.

To bring an efficient virtualization to the cloud, many studies focused on the I/O performance improvement in virtualized systems. These studies can be broadly classified into four categories: 1) reducing device virtualization overhead [3], [4], [5]; 2) optimizing I/O path [6], [7]; 3) providing middleware support at the hypervisor level [8], [9]; 4) customizing the scheduling policy for I/O intensive VMs [2], [10]. With the major focus on resolving the I/O bottleneck, the existing studies do not provide comprehensive evaluations on the performance of hybrid workloads in cloud environments. Herein, hybrid workloads may experience performance degradation in multiple aspects, including I/O throughput, computation speed, and memory usage. The impact remains largely unexplored, and a solution is yet to be developed for common cloud services demanding both data processing and transmission.

In this paper, We for the first time performed a comprehensive measurement study to quantify the impact of self interference with typical hybrid workloads. In our experiments, when running a hybrid transcoding and streaming workload

- *C. Xu, R. Shea, and J. Liu are with the School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada. E-mail: {chix, rws1, jcliu}@sfu.ca.*
- *X. Ma is with the School of EIC, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China, and also with the School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada. E-mail: mxqhust@gmail.com.*
- *H. Wang is with the Department of Computer Science, University of Minnesota Duluth, Duluth, MN 55812 USA. E-mail: haiyang@d.umn.edu.*

Manuscript received 6 Sept. 2017; revised 17 Mar. 2018; accepted 16 Apr. 2018. Date of publication 15 May 2018; date of current version 5 Mar. 2021. (Corresponding author: Xiaoqiang Ma.)
Recommended for acceptance by V. Piuri.
Digital Object Identifier no. 10.1109/TCC.2018.2837040

1. Cross-VM interference refers to the interference caused by the impact of co-located VMs due to the imperfect isolation provided by the hypervisor [11], [12], [13], [14].

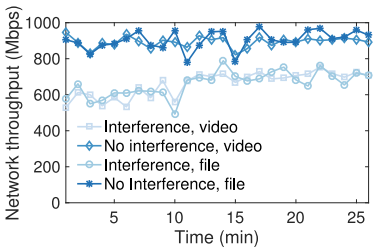


Fig. 1. Network interference.

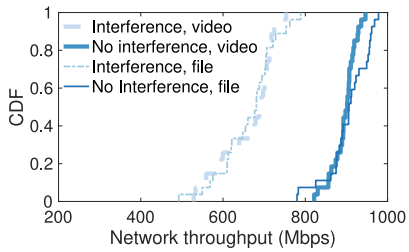


Fig. 2. CDF of network throughput.

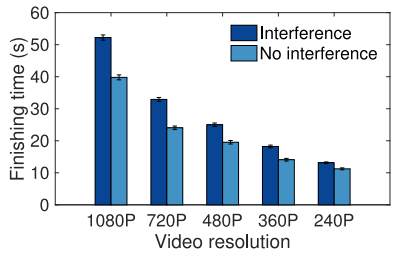


Fig. 3. Transcoding performance.

in the KVM environment, the network throughput falls by up to 32.1 percent, and the computation time increases by up to 32.5 percent, all due to the existence of self interference. Motivated by the measurement results and an in-depth analysis, we present *Hylics*, a novel virtualization architecture that jointly optimizes I/O and computation performance for hybrid workloads. The insight of the Hylics design is to shorten the data traverse paths for both data processing and transmission. Meanwhile, it also decouples I/O and computation operations for cloud VMs. In particular, Hylics stores cloud applications' data in the in-memory file system at the hypervisor level. By doing this, the data traverse path now originates, or ends, at the hypervisor-level memory space. The design also shifts VM's network operations to the hypervisor layer. The self interference is therefore minimized, enabling nearly bare-metal networking performance and enhanced computation performance. More importantly, our solution significantly improves the energy efficiency when handling hybrid workloads. The Hylics design is not confined to any specific protocols or applications, and conceptually raises the level of provisioned interface from physical device to high-level I/O service. We have addressed several key design issues in the Hylics architecture, including efficient intermediate data sharing, network service offloading, and QoS-aware memory usage management. We also developed a prototype system in the KVM environment. The system consists of loadable kernel modules and programming interfaces for both VM and hypervisor. Our extensive evaluations show that Hylics can improve network throughput by up to 27.8 percent via resolving the self interference. Meanwhile, the experiments also indicate that Hylics can reduce the completion time for computation tasks by up to 31.2 percent.

To summarize, our contributions are listed as follows:

1) We for the first time performed an in-depth study to quantify the impact of self interference with typical hybrid workloads in virtualized cloud environments;
2) We proposed and implemented Hylics, an enhanced virtualization framework to eliminate the self interference and addressed several important design issues therein;
3) We proved that Hylics can jointly boost the network and computation performance for hybrid workloads. The energy efficiency of the underlying server is also improved.

Compared with the preliminary version of this paper [15], we have made several additional contributions: 1) we introduced a new approach to assign and adjust the utilized in-memory file system space among different VMs; 2) we proposed two different schemes for offloading network transmission to the hypervisor layer, either by offloading network middleware modules, or by offloading valid socket copies; 3) we presented more comprehensive evaluations, including hypervisor memory usage analysis and energy efficiency measurements.

The rest of this paper is organized as follows: In Section 2, we introduce the background and motivations of this paper. In Section 3, we present the framework design of the Hylics architecture. Section 4 describes the implementation details. Section 5 introduces the analysis and enhancement of Hylics memory usage. In Section 6, we show the experimental results, as well as the further analysis. Section 7 surveys the related work and Section 8 concludes this paper.

2 Motivation

Previous research has confirmed the existence of self interference in virtualized environments, and quantified the degradation of network performance with benchmark tools [2]. To further understand the impact of self interference with real-world applications, we have conducted measurements on two sets of typical hybrid workloads in cloud environments: 1) video transcoding and streaming, and 2) file compression and delivery.

2.1 Measurement and Observation

The first experiment is conducted on a VM which serves as a video streaming server and also a transcoder with real-world applications `LIVE555` and `FFmpeg`.[2] The VM is allocated with 8 vCPUs, 32GB RAM, running in the KVM environment and fulfilling on-demand transcoding and streaming requests from 500 clients. We first present our experiment results on networking performance in Fig. 1 (labelled with "video"). In the beginning, the throughput of this VM is relatively stable around 892 Mbps when it is dedicated to handling the streaming traffic. Unfortunately, when we started to add concurrent transcoding tasks to the VM, the throughput becomes as low as 661 Mbps, not to mention the high variations. Before the transcoding task finished, the clients experienced a 32.1 percent throughput degradation. We also plot the CDF of the network throughput in Fig. 2 to present the performance loss. On the other hand, for the concurrent transcoding tasks, the processing is also delayed. Our experiments show that the transcoding task completion time increased by 8.3-32.5 percent with different output settings, which is shown in Fig. 3. Our measurement results of the second set of hybrid workloads are captured while simultaneously running file compression and delivery tasks. The experiment is performed with `pigz` and `Lighttpd` web server. In this case, the VM serves as a

2. The detailed experiment configuration is described in Section 6.

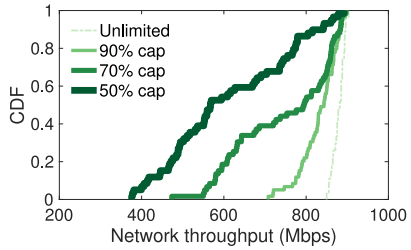


Fig. 4. Scheduling policy.

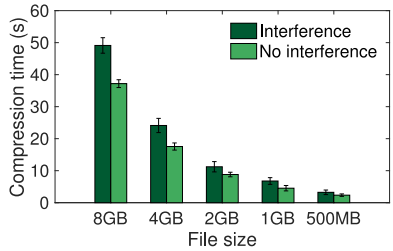


Fig. 5. File compression performance.

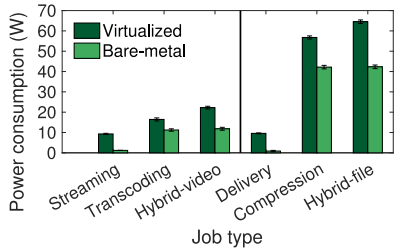


Fig. 6. Energy comparison.

file entrepot for compressing and transferring data blocks. The network throughput results of the webserver are also shown in Figs. 1 and 2 (labeled with "file"). The web server also has a 33.4 percent throughput degradation when co-located with the compression tasks. To take a step further, we maintained a hard limit on the CPU usage of the VM. Such a non-work conserving scheduling policy has often been adopted in real-world commercial clouds to achieve better isolation and fairness between VMs. Unfortunately, the CDF of the VM's network throughput in Fig. 4 indicates that, as we set a lower cap on the CPU usage of the VM, the average network throughput significantly decreases, together with a perpetual network instability. As for the compression performance, Fig. 5 shows similar results as in the previous set. The task completion time increased by up to 52.11 percent. To summarize, our measurement results indicate that adopting such a scheduling policy further deteriorates the performance of the hybrid workloads in multiple aspects.

As a matter of fact, many cloud applications may involve even more complex hybrid workflows. It is reasonable to believe that there is a prevalent existence of the self interference in the cloud context. The root cause of such self interference is a combined effect of the I/O subsystem design and the scheduling policy in virtualized environments. To take a step further, we use the KVM environment as an example to introduce the I/O architecture design in a typical virtulized system. A closer look into KVM's network architecture is given in Fig. 7, which lists the key steps involved in delivering network packets to a VM. A state-of-the-art solution, vhost-net,[3] is a high-performance virtio network device emulation that takes advantage of advanced zero-copy and interrupt handling features. Despite the reduced data copies, the network packets still have to traverse through multiple protection layers before eventually reaching the end application. Furthermore, when hybrid workloads are running on the VM, the computation tasks may consume the entire CPU quotas required for handling the network transmission, and vice versa. If a VM expends the entire CPU time slice, it will probably go to the end of the scheduler's queue and have to wait a considerable amount of time before reaching the front again. If the system administrator uses a non-work conserving scheduling policy on the VM, the VM will not be scheduled again until the next scheduler accounting period, even if there are available CPU resources.

Apart from the network I/O operations, the hybrid workloads also involve a large number of disk I/O operations. For example, in the aforementioned transcoding tasks, it also involves reading the raw files from the virtual disk space. In Fig. 8, we present the disk I/O workflow in the KVM environment. In the native KVM design, a disk read operation involves these following steps: an application inside the guest VM uses a generic virtual file system (VFS) interface to issue disk read requests. The guest VM first fills in the request descriptors, then writes to the virtio-blk virtqueue and notifies the related registers. Afterward, the QEMU process issues the I/O requests on behalf of the guest VM, then the QEMU process fills in the request footer and injects the completion interrupts. The guest VM then receives the interrupts and executes the I/O handler. Eventually, the application reads data from the kernel buffer. Similarly as in the network I/O subsystem, the disk I/O data also need to traverse through multiple protection boundaries, which also incurs a great amount of virtualization overhead.

Meanwhile, the ceaseless state changing of a VM (e.g., from running state to block state), together with the context switching between computation and I/O handling inside the VM, increases the power consumption of the underlying server. As a concrete example, Fig. 6 shows that, when we run the stand-alone streaming task or the stand-alone transcoding task inside a VM, the CPU power consumption of the physical server increases by 8.1 and 5.2 W, respectively. However, when we run the hybrid workload inside the VM, the CPU power consumption increases by 10.3 W. Fig. 6 also shows a similar power consumption increase when running the file compression and delivery workload. All these results are compared with the bare-metal case. Such a noticeable increase calls for a revisit on the current virtualization architecture design.

2.2 Existing Approaches and Opportunities

Recent works on improving I/O performance in virtualized environments can be broadly classified into four categories: 1) reducing device virtualization overhead [3], [4], [5]; 2) optimizing network I/O path [6], [7]; 3) providing middleware support at the hypervisor layer [8], [9]; and 4) customizing the scheduling policy for I/O intensive VMs [2], [10]. We will then discuss the existing approaches and some opportunities in this section.

SR-IOV enabled NICs [16] can handle network traffic without the involvement of hypervisor. Unfortunately, such devices do not allow hypervisors to inspect and control VMs' I/O activities [17], e.g., performing security and QoS regulations. Another well-known issue is that a VM with passthrough devices is not compatible with live migration. Using exit-less interrupt delivery mechanisms [3], [4] can effectively reduce the interrupt handling overhead of virtual devices. In particular, I/O events are passed to a VM

3. http://www.linux-kvm.org/page/UsingVhost

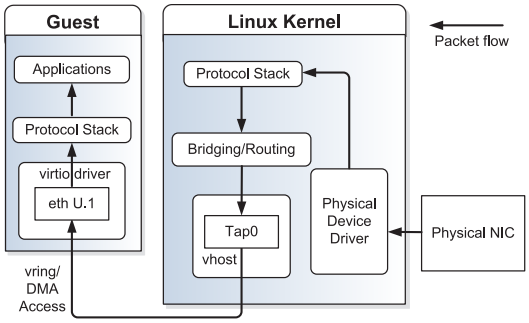


Fig. 7. KVM network I/O subsystem.

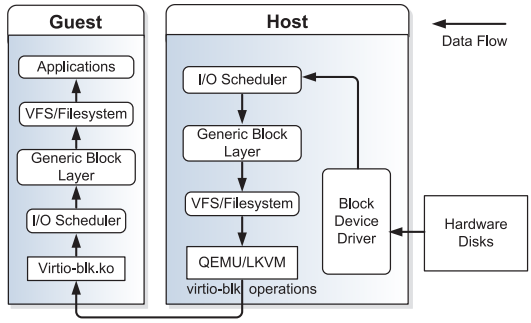


Fig. 8. KVM disk I/O subsystem.

without exiting the hypervisor. However, such methods cannot eliminate the self interference between the computation and I/O processing inside the VM. The negative effect of VM consolidation also dominates the I/O performance in virtualized systems. For example, as an increasing number of VMs share a given CPU, the scheduling latencies, which can be in the order of tens of milliseconds, substantially increase the typically sub-millisecond round-trip time for TCP connections in a datacenter, causing significant throughput degradation. To mitigate such problems, vPro [6] proposes to offload TCP protocol functionality, including congestion control and acknowledgement to hypervisor. The solution only benefits *small* TCP flows and does not work for UDP protocol. vPipe [7] discusses the opportunity of enabling piped I/O at the hypervisor layer for static data transfer. vRead [8] and VAMOS [9] represent the effort on providing I/O middleware support at the hypervisor level. These works target on specific workloads in cloud environments, e.g., Hadoop workload and MySQL database workload.

The scheduling latency of network I/O intensive VM is also critical in virtualized environments, since VM scheduling can bring noticeable delays to network I/O processing. A recent study [2] suggests reducing CPU time slice for I/O intensive VM. This enables an I/O intensive VM to get scheduled more often so as to improve its I/O throughput. vTurbo [10] takes a step forward to offload VMs' I/O processing to a dedicated core with an extremely small time slice. In summary, these works focus on modifying the hypervisor scheduler to reduce the scheduling delays. However, these approaches bring increased VM state changes, as well as complicated CPU resource allocation strategies.

We can see that the previous approaches mainly focus on *I/O stack* optimization, without evaluations on the overall performance of hybrid workloads with concurrent CPU and I/O operations. In this work, we aim to jointly optimizing I/O and computation performance via enabling cloud applications to better cooperate with the virtualization layer. Our solution rests on the facts that,

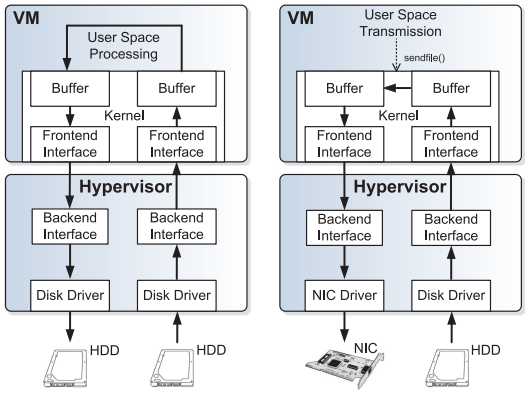
1) The network I/O data transfer is essentially expensive between VM and hypervisor. As mentioned before, the complex network encapsulation and configuration in virtualized environments is one key bottleneck to be revisited. What has been largely missed in mainstream virtualization technologies is the opportunity for raising the level of provisioned interface from physical device to high-level I/O service. In this paper, we discuss the opportunity of providing guest VM with an entire networking service;
2) Hypervisor-level network I/O operations can achieve nearly bare-metal performance as well as high energy efficiency. In a general virtualized system, since the hypervisor has the privilege to use native device drivers, it is therefore reasonable to take advantage of hypervisor-level network I/O operations to shorten the network packet traverse path which originally begins or ends inside a VM;
3) For the computation part in hybrid workloads, instead of performing read and write operations on the file system hosted in virtual disk, it will be generally more efficient to operate on the file system hosted in memory space. Therein, instead of providing a virtual disk device, we also seek to provide a lightweight in-memory file system interface for hybrid workloads.

Based on these observations, we propose Hylics, an enhanced virtualization framework for hybrid workloads in cloud environments.

3 Framework Design

In this section, we present the complete design of the Hylics framework. To mitigate the self interference inside VM and jointly optimize I/O and computation performance, the principle is to decouple network I/O operations and computation operations, as well as shorten the data traverse path for both of the two parts. In particular, Hylics provides an in-memory file system for storing application data and shifts VM's network operations to the hypervisor layer. Such network operation offloading borrows the idea of separating the control plane and data plane of network transmission. As such, in the Hylics architecture, the data transfer only takes place at the hypervisor layer.

To provide an intuition of the Hylics design, in Figs. 9 and 10, we present a comparison between the Hylics abstraction and the native virtualization abstraction. For the data processing part, in the conventional virtualization architecture design, a pair of frontend and backend virtual disk interface is offered to transfer data between a VM and its hypervisor, which incurs a great amount of virtualization overhead. The data first need to be loaded from the hardware disk drive to the hypervisor memory space. Such overhead is then eliminated with the in-memory file system provision in Hylics. For the network data transmission part, the Hylics design directly enables VM to send data from the in-memory file system, with the network service provisioned at the hypervisor layer.



(a) Processing (b) Transmission

Fig. 9. Native workflow for hybrid workloads.

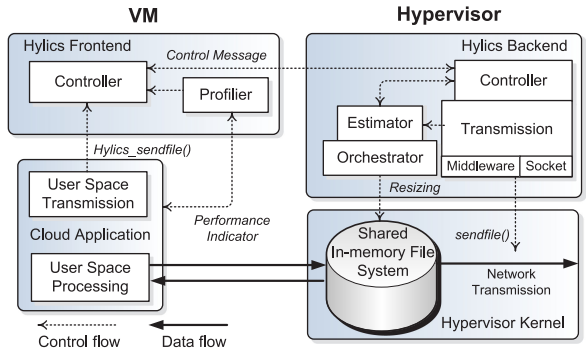


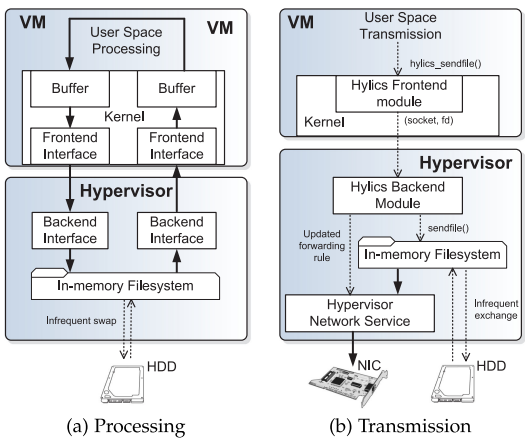
Fig. 11. Detailed system architecture.

3.1 Overview

As presented in Fig. 11, the Hylics design consists of the following components:

Shared in-Memory File System. The shared in-memory file system stores the intermediate data for cloud applications. The intermediate data in this context refer to the application data stored in the file system that are required to be processed, for example, the raw video files to be transcoded and then delivered, the original files to be compressed and then transferred. In our design, the shared in-memory file system is allocated in the virtual memory maintained by the host kernel, which can dynamically grow and shrink to accommodate the files it contains. Note that the maximum size limit of the file system can be adjusted on-the-fly; this feature enables us to provide an SLO-aware memory usage control scheme. We will further present the detailed shared in-memory file system design in Section 4.1.

Cloud Applications. Cloud applications that involve data processing and network transmission can utilize the in-memory file system provided by the Hylics architecture. This is enabled by using the VFS interface provided by the Linux kernel. The purpose of a VFS interface is to allow applications to access different types of concrete file systems in a uniform way. Note that the processing logic of cloud applications remains unchanged, which is beneficial to the application developers. As for the network transmission part, cloud applications are provided with a modified version of system calls to issue network transmission request. In this fashion, the actual data transfer is originated from the in-memory file system with the offloaded network service at the hypervisor level.



(a) Processing (b) Transmission

Fig. 10. Hylics workflow for hybrid workloads.

Hylics Frontend Module. Hylics frontend module provides programming interfaces for dealing with the network transmissions of cloud applications. To cope with different applications, we propose two different schemes for offloading network transmission to the hypervisor layer, either by offloading network middleware modules, or by offloading valid socket copies. The frontend module communicates with cloud applications and serves as an agent to retrieve file descriptor, destination information, and socket level information. It will then pass the information to the backend module to perform actual data transfer. Meanwhile, it also closely measures the performance metrics of cloud applications to provide raw profiling results for further performance optimization.

Hylics Backend Module. Hylics backend module is responsible for VMs' network transmission, and runs in the user space of the hypervisor layer. With the middleware or socket operations offloaded to the hypervisor layer, the backend module serves the VMs by starting multiple threads to send the data from the in-memory file system. The backend module utilizes hypervisor-layer network stack to transmit the application data, which can achieve nearly bare-metal networking performance. To ensure security, the backend module runs inside a separated zone in the host space. We will further elaborate the design in Section 4.2.

System Profiler and Estimator. The system profiler in the Hylics architecture is a collection of monitoring tools to get the application performance indicators in a real-time fashion, e.g., task response time, file system usage information and detailed logs. We collect such data for the estimator to analyze and identify the system model. In particular, we leverage a widely-used queueing model predictor to achieve a fast system model approximation. We also propose a self-adaptive control scheme to mitigate the residual model errors. The identified system model is further used for optimizing the memory usage. The detailed estimator design is discussed in Section 5.

Hylics Controller and Orchestrator. We design the controller to coordinate the computation progress and network transmission of cloud applications. Meanwhile, the controller also manages the file system size allocated for each VM. The controller receives the feedback from the system profiler and estimator. It also accepts users' SLO indicator as an input. Based on the information, it makes optimization

decisions to enhance the resource usage. In consideration of computation complexity, we design a simple yet effective online feedback control loop. The design is further explained in Section 5. The orchestrator implements and executes the controller's optimization decisions, e.g., the optimized in-memory file system space assignment for each VM. The decisions are executed by the in-memory file system resizing mechanism with standard kernel interfaces.

3.2 Case Study

We use the hybrid workload "transcoding and streaming" as a concrete example to illustrate the entire workflow in the Hylics architecture. The raw video segments, originally stored in the virtual disk space, are now lifted and stored in the shared in-memory file system. The transcoding task fetches and processes the video segments from the shared in-memory file system. This is enabled with a paravirtualized file system interface via offloaded Linux VFS API. Due to the general abstraction of the Linux VFS API, transcoding task can still use the same system call on handling these file I/O operations. The post-processed video segments are also stored in the shared file system for the on-demand transmission. To fulfill the concurrent streaming request, the VM can use the network middleware module at the hypervisor layer for video streaming. The sending result is then passed back to the VM via a lightweight guest-host communication channel. Such communications only involve control message exchange between the VM and its hypervisor, causing minimum overhead as compared to transferring the actual video data. The VM also has another option to offload socket level operations to the hypervisor layer. In this scenario, the VM first needs to set up connections with valid sockets and then sends out video files with the modified programming interfaces provided by the Hylics architecture. While running the hybrid workload, the profiler module continuously collects the runtime information and keeps track of the performance indicators. In this case, we use transcoding response time and sending throughput as the default indicators. The Hylics controller gathers such information from both the guest VM and its hypervisor and then the estimator calculates the optimized memory space required by the VM. Afterward, the orchestration is performed to adjust the size of the in-memory file system.

4 System Implementation

4.1 Sharing In-Memory File System

We delve into the implementation details of the shared in-memory file system in this section. Typically, a file system is used to define how file data are stored and retrieved, which includes two types of information—the data blocks residing on the file system, and the control information used to maintain the state of the file system. An in-memory file system uses resources and structures of memory subsystem, which supports UNIX file semantics meanwhile is fully compatible with other common file systems. Hosting a file system inside the memory space provides better performance for file reading and writing. This feature is utilized by our Hylics design so that the general file access of cloud applications causes a memory-to-memory copy of data, no I/O requests for file control updates are generated. Meanwhile, since file system attributes are stored once in memory, no additional I/O requests are needed for file system maintenance. In the Hylics architecture, rather than directly using dedicated physical memory, we choose to utilize the operating system page cache maintained by the kernel for storing file data. Such an implementation generally provides increased read and write performance with no adverse effects on the system compatibility. This is because we can further take advantage of the native resource management policies in the Linux kernel. Another important aspect is the space management of the in-memory file system, instead of allocating a fixed amount of memory for exclusive use, using system page cache space enables a dynamic resizing mechanism depending on use, which allows the Hylics architecture to adjust its memory usage on-the-fly. To meet all the design criteria, we employed tmpfs file system [18] during the implementation. When we initialize the Hylics architecture, a VFS structure is allocated, initialized and added to the kernel's list of mounted file systems. A tmpfs-specific mount routine is then called, which allocates and initializes a tmpfs mount structure, and then allocates the root directory for the file system. It allows us to use anonymous memory in the page cache to store and maintain file data. Since the kernel does not differentiate tmpfs file data from other page cache uses, the stored file blocks can be written to swap space. This could happen when the system is in an urgent need of memory. Control information is maintained in physical memory allocated from kernel heap. The file data are then accessed through only one level of indirection provided by the virtual memory system.

Besides the in-memory file system provision, another critical issue is to efficiently share the file system between a VM and its hypervisor. In the current design, we enable a paravirtualized file system driver to minimize the virtualization overhead, which is based on the widely deployed virtio framework [19]. This interface presents some unique advantages over the traditional virtual block device. By paravirtualizing a file system interface, we further avoid a layer of indirection in converting VM's file system operations into block device operations and then again into host file system operations. A paravirtualized interface provides a preconnected and isolated channel between a VM and the hypervisor, which incurs none of the overhead of arbitrary and unnecessary encapsulation when going over a network stack incurs. Our implementation is to leverage a lightweight distributed file system protocol directly on top of paravirtualized transport [20]. We achieve this by providing a virtualization-aware transport interface through the virtio framework. The shared file system space is then ported as a local file system on the guest VM. To ensure system security, we also enable standard Linux access control mechanisms, including SELinux, chroot, and seccomp, to limit a QEMU process to access its own resources. Each QEMU process is restricted to only access the part of the shared file system space that is relevant to the VM it runs.

4.2 Offloading Network Operations

To complete the Hylics design, another critical issue is to offload VM's network operations to the hypervisor. One possible solution is directly using the network stack at the hypervisor layer with offloaded network middleware modules. Such modules run at the hypervisor level and

cooperate with the hypervisor resources. The execution results, interpreted as control messages, can be sent back to the VM via an inter-domain communication channel, e.g., virtio-vsock.[4] Such offloading is favorable for those workloads whose functional logic is easy to be further decoupled. They can use a client side in the VM and a server side running at the hypervisor level. The network I/O middleware modules offloaded at the hypervisor level can effectively reduce the number of VM-hypervisor context changes, as well as the device emulation overhead. However, running an additional module at the hypervisor level may potentially raise concerns about the complexity and security risks. We emphasize that, with a careful design, the modification is minimal. The network middleware module can be loaded as a user space process at the hypervisor layer, and runs in an isolated environment. To neutralize security vulnerabilities, the hypervisor can further restrict the privilege of the network middleware module. For instance, in the KVM environment, to defend against compromised running components, a QEMU process can use plugin-isolation mechanisms [21], [22].

Another possible solution is offloading TCP/UDP socket level operations. Therein, we also design interfaces for Hylics to pass the entire TCP/UDP processing functionality by establishing a socket copy in the hypervisor layer. Particularly, we first provide a variant of the system call `send()` or `sendfile()` for user applications. An application can either use `hylics_send()` or `hylics_sendfile()` for network transmission. The parameter includes socket descriptor (which is obtained by accepting clients' connection), file descriptor (by locating the targeted file), file offsets and byte count. After getting parameters from cloud applications, the Hylics frontend module in the VM first converts this socket descriptor to a socket structure with necessary information for establishing the socket copy in the hypervisor layer. Such information includes the source and destination IP addresses, source and destination ports. Meanwhile, the Hylics frontend also interprets the file descriptor as a file path inside the shared in-memory file system. As soon as these parameters are passed to the Hylics backend module, the backend module sets up a socket copy. It then begins to fetch data from the in-memory file system and performs the actual data transfer. When the transfer is completed, the number of bytes sent is returned to the cloud application. After the transmission begins, the original socket inside the VM is disabled and the traffic is redirected to the hypervisor layer by explicitly setting the packet forwarding policy.

5 Memory Usage Analysis and Enhancement

Hylics leverages the memory resource at the hypervisor layer to improve the overall performance of hybrid workloads. Therein, the space management of the shared in-memory file system is undoubtedly a key design issue. Intuitively, the static management or fixed in-memory file system size for each VM can provide isolation and fairness between VMs. However, it may also result in either resource waste or VM performance degradation. Furthermore, the rigid management of memory usage also suffers from the inflexibility in the presence of memory intensive tasks. Therefore, We believe that system administrators should at least be able to switch between static and dynamic strategies. Although almost all modern hypervisors implement memory overcommitment mechanisms such as ballooning, page sharing, and swapping; they lack policies to coordinate these mechanisms in order to minimize performance degradation for cloud applications. We then introduce an online approach to assign and adjust the utilized memory space among different VMs. By online, we mean that the controller design achieves system identification and makes adjustment decision by processing pairs of input-output measurements sequentially. In the cloud context, such online processing is important since the task size and arrival rate are highly dynamic and hence is the efficiency of the Hylics memory space usage. As a next step, we propose an *online self-adaptive control scheme* to meet users' SLO while keeping a moderate memory usage.

5.1 Online Self-Adaptive Control Scheme Design

To provide a robust control scheme, we combine queueing modeling and adaptive control together in this work. The reason why we need such a design is twofold. First, we learn from Google's trace analysis [23], [24] that typical job inter-arrival time exhibits an exponential distribution. Although Google's trace data cannot include all possible hybrid workload types, the level of detail and mixture of workload types in this trace is unprecedented [24]. Meanwhile, queueing model is also widely applied in the cloud context to provide simplification on the system that has a bottleneck stage [25], [26]. Second, the adaptive control loop can build the residual error model and enhance the controller performance. It can reduce the inaccuracies in the queueing model and handle the sudden change of hybrid workloads in a dynamic fashion. The combination of queueing model predictor and adaptive control provides a better performance regulation under a wide range of workload conditions.

Our abstraction of a Hylics-enabled VM is an M/G/1 processor sharing queue (M/G/1/PS). With the network-related tasks offloaded to the hypervisor layer, we assume that the VM is now exclusively working on the computation tasks with the assistance of the shared in-memory file system. To begin the self-adaptive control scheme design, we first introduce the notations in the queueing model (summarized in Table 1). We denote by $R(x)$ the average response time of a computation task whose service time is x. According to the queueing model definition, the service time is an i.i.d random variable in an M/G/1/PS system, denoted by X, of which the probability distribution function is $P(X)$, with an average $E[X]$. The load and arrival rate of the queue are denoted by ρ and λ, respectively. The average response time for all tasks on the VM, is then calculated by

$$R = \int_0^\infty R(t)dP(t) = \frac{E[X]}{1-\rho}. \tag{1}$$

Let the size of the in-memory file system space allocated for the VM be m. In this work, we assume that the average service time of a task is a variable subject to the file system size m. Then we have

$$R(t) = \frac{E[X|m]}{1-\lambda(t)E[X|m]}. \tag{2}$$

4. http://wiki.qemu.org/Features/VirtioVsock

TABLE 1
Summary of Notations

Notations	Definitions
$R(x)$	average response time of a task
$P(x)$	PDF of service time
$E[X]$	average service time
ρ	load of a queue
λ	arrival rate of a queue
m	current allocated in-memory file system space
τ	reference delay
k	sequential number of a measurement pair
A, B, a, b, q	general control model parameters
n_A, n_B, d	control model orders
$y(k)$	kth control output
$u(k)$	kth control input
$\theta(k)$	kth parameter vector
$\phi(k)$	kth input-output pair
$F(k)$	kth adaption gain matrix
I	identity matrix

The goal of the adaptive control is to adjust the average response time of tasks $R(t)$ as close to the reference delay τ as possible. The reference delay τ indicates users' SLO. Suppose the queueing model is accurate, then from Equation (2), we can then directly set the file system size m to get the ideal average service time, so that we can further get the steady response time to be exactly τ.

As the next step, we present the adaptive feedback loop design. In this context, the purpose of the adaptive control loop is to correct the "residual errors" of the response time ($\Delta\tau$) by dynamically tuning the adjustment of Hylics file system space (Δm). Considering the overall performance of the system, we apply direct adaptive control for its efficiency and simplicity. We then consider the system as a discrete time model with adjustable parameters estimated by a recursive least squares (RLS) estimator [27], which is an online version of the well-known least-square estimator. Such online parameter estimation can provide us with real-time feedback. The parameters are updated during each control interval to minimize the queueing model error. Then the control law is calculated by setting the adjustment of Hylics space (Δm) to diminish the residual errors ($\Delta\tau$).

To be more specifically, in order to construct the control law, the adaptive controller first needs to estimate the residual error model for the system whose parameters can be used in the controller. In the following discussion, we describe the scheme with a general model

$$A(q^{-1})y(k) = q^{-d}B(q^{-1})u(k), \tag{3}$$

where

$$\begin{aligned} A(q^{-1}) &= 1 + a_1 q^{-1} + \cdots + a_{n_A} q^{-n_A}, \\ B(q^{-1}) &= b_0 + b_1 q^{-1} + \cdots + b_{n_B} q^{-n_B}, b_0 \neq 0, \end{aligned} \tag{4}$$

and $y(k)$ is the control output, $u(k)$ is the control input. In the Hylics context, $y(k)$ corresponds to the residual error of the response time $\Delta\tau(k)$. and $u(k)$ corresponds to the memory space adjustment $\Delta m(k)$. Due to the digital implementation of the controller, the effect of the control command determined on time interval k can only be measured in interval $k+1$, then we set the delay order as $d = 1$. The model parameter of Equation (4) are estimated with the RLS estimator. Let

$$\begin{aligned} \theta(k) = [&\theta_1(k), \ldots, \theta_{n_A}(k), \theta_{n_A+1}(k), \\ &\theta_{n_A+2}(k), \ldots, \theta_{n_A+n_B+1}(k)]^T, \end{aligned} \tag{5}$$

and

$$\begin{aligned} \phi(k) = [&y(k), y(k-1), \ldots, y(k-n_A+1), u(k), \\ &u(k-1), \ldots, u(k-n_B)]^T, \end{aligned} \tag{6}$$

The target function of the RLS estimator is defined as

$$\min_{\hat{\theta}(k)} J(k) = \sum_{i=1}^{k} [y(i) - \hat{\theta}^T(k)\phi(i-1)]^2. \tag{7}$$

The term $\hat{\theta}^T(k)\phi(i-1)$ in Equation (7) corresponds to $\hat{y}[i|\hat{\theta}(k)]$. This is the prediction of the output at instant i ($i \leq k$) based on the parameter estimated at instant k, which is obtained by using k measurements. The RLS algorithm works by calculating the adaption gain matrix F and updating the model parameter θ

$$\begin{aligned} F(k-1) = F(k-2) - &[1 + \phi(k-1)^T F(k-2)\phi(k-1)]^{-1} \\ &F(k-2)\phi(k-1)\phi(k-1)^T F(k-2), \end{aligned} \tag{8}$$

$$\begin{aligned} \theta(k) = \theta(k-1) + &F(k-1)\phi(k-1)[y(k) \\ &- \phi(k-1)^T\theta(k-1)], \end{aligned} \tag{9}$$

then the control law for the memory space adjustment is given by solving

$$\phi(k)^T\theta(k) = y^*(k+1), \tag{10}$$

where $y^*(k+1)$ is the residual response time error at instant $k+1$. In this case, since we are considering to let the "residual errors" to diminish, then we need to set $y^*(k+1) = 0$. The above algorithm begins with initial condition $F(-1) = f_0 I$ and $f_0 > 0$.

Algorithm 1. Hylics File System Size Control

```
1: Pre-define R_thresh and m_thresh
2: Pre-define ci_minthresh and ci_maxthresh
3: Initialize gain matrix F = f0 I and parameter vector θ
4: while control interval (ci) expires do
5:    Acquire average response time R(t) from profiler
6:    if ΔR ≥ R_thresh then
7:       Update λ(t) and queueing model
8:       Update m̂ by solving Equation (2)
9:       m ← m̂
10:   end if
11:   Update adaption gain matrix F by Equation (8)
12:   Update parameter vector θ by Equation (9)
13:   Acquire Δm by solving Equation (10)
14:   if Δm ≥ m_thresh then
15:      m ← m + Δm
16:      ci ← max{ci/2, ci_minthresh}
17:   else
18:      ci ← min{ci * 2, ci_maxthresh}
19:   end if
20: end while
```

As a summary, we list the key steps of the Hylics memory space control algorithm in Algorithm 1. As shown in line 1

and line 2 of the algorithm, we first need to define the threshold of the response time changes (R_thresh) and memory space changes (m_thresh). This is to avoid the adjustment oscillations during the control loop. $ci\_maxthresh$ and $ci\_minthresh$ are used for setting the upper bound and lower bound of the control interval, respectively. The control loop is executed whenever the control interval (ci) expires. Note that the control interval is also adaptively set to avoid oscillations, which is shown in line 16 and line 18.

5.2 Case Study and Parameters Selection

We also use the hybrid workload "transcoding and streaming" as an example to show how the proposed scheme works. We first leverage the queueing model predictor to approximate the transcoding performance. Then, the proposed adaptive control loop compares the response time of the transcoding task with the referenced transcoding delay at each control interval. The controller then adjusts the usage of the in-memory file system to approach the desired performance. Whenever there is a large spike in the response time, the queueing model predictor is recalculated. If there is only a minor change in the transcoding response time, we update the in-memory file system size merely with the adaptive control loop. The video data are then transferred between the in-memory file system and the default file system hosted in the virtual disk based on the control law.

Next, we discuss how the model parameters are selected in this work. The average service time $E[X|m]$ given specific m for different workloads used in the queueing model predictor is measured offline. To this end, we first initiated a lightweight workload on the VM. We then varied the utilized in-memory file system size and measured the response time. The measured average response time approximates the average service time since there is no queueing delay during the experiment. We conducted the measurement test multiple times (≥ 100) and then averaged the measured time to get $E[X|m]$. As the next step, we need to determine the model orders n_A and n_B to complete the adaptive controller design. The model orders n_A and n_B are also measured in an offline fashion. We used the following method to determine these parameters: we first disabled the adaptive controller and collected offline data $\Delta\tau$ by using only the queueing model predictor and a white noise input of Δm. Under different combinations of n_A and n_B, we used a direct model identification method—a default least square estimator to get the corresponding model parameters, then we tested which θ acquired from these combinations has good fitting performance. This means using the θ, a new data group of collected data pairs (Δm, $\Delta\tau$) produces high r^2 value [27]. r^2 denotes the percentage of variations that can be explained by the model. In the Hylics system design, we found the parameter choice with $n_A = 1$ and $n_B = 0$ has the best fitting result.

6 Performance Evaluation

In this section, we conducted experiments to understand the performance of the proposed design. The criteria include file system read and write performance, computation performance, networking performance, and energy efficiency. First, we introduce the testbed configuration, together with the hardware and software environments.

6.1 Experiment Configuration

Our experiments are conducted on a typical rack server Dell PowerEdge R430 Server 1U. It is equipped with two Xeon E5-2630 v3 2.4 GHz octa-core CPUs and 256 GB RAM. To understand the networking performance, we deployed another rack server with the same hardware configurations. These two servers are installed with Intel Corporation Ethernet Controller X540-AT2, and the interfaces are connected by a Netgear XS712T switch with 10 Gbps Ethernet link.

The operating system running on the host machine and the guest VM is Ubuntu 16.04.2 LTS. The kernel version of the host machine is Linux 4.4.0-78. For the guest VM, we have upgraded the guest kernel (version 4.7.0+) to support the virtio-vsock interface. We then configured qemu-kvm 2.5.0 on this testbed machine. Based on the typical VM configurations in public clouds, we set the default number of accessible vCPU to be 8 for the guest VM. The VM is then equipped with 32 GB RAM. We used `ifstat` to measure the network throughput with the probing interval set as one second. To collect the detailed system information, we captured the virtual CPU utilization using `top`, which is a standard resource monitoring tool integrated into the Linux distribution. Furthermore, we used the Linux hardware performance analysis tool `perf` to collect such system level statistics as CPU cycles and context switching information. To avoid randomness in our data, we ran each experiment 100 times and calculated the average and standard deviation. We used a Linux library function `gettimeofday()` to calculate the running time of computation tasks. The granularity is one microsecond. In terms of energy efficiency, the CPU power consumption is captured by RAPL counters in Intel's Sandy Bridge processors.

To comprehensively understand the performance of Hylics, we also selected representative benchmark tools (`dd` and `sysbench`) and real-world applications (`LIVE555`, `FFmpeg`, `pigz`, and `Lighttpd`), which are introduced in Table 2. In detail, the transcoding task conducted by `FFmepg` is a typical transsizing task. Transsizing consists of operations on changing the picture size of video, which is commonly seen in cloud environments for streaming to different devices. Our video source is multiple 1440P (2560×1440) video segments with 60 FPS frame rate. The length of one individual video segment is 30 seconds. In our experiments, we performed transsizing tasks to convert the video segments into different resolutions. The file compression task is performed by `pigz` with the default compression ratio, and we set the number of compression threads to be exactly the same number of the allocated vCPUs.

6.2 Benchmark Test

As the first step, we used benchmarking tools `sysbench` and `dd` to understand the file I/O performance of the shared in-memory file system. Fig. 12 compares the read throughput when the VM is reading from the virtual block device (dev) and the Hylics in-memory file system (Hylics). In this experiment, we disabled the operating system block cache and varied the block size to thoroughly understand the overall performance. As shown by our results, Hylics file system performs better in both sequential read and random read scenarios. This is mainly because the read operations are now executed in memory space. Fig. 13 presents the

TABLE 2
Benchmark Tools and Real-World Applications

Tool	Reference	Description
dd	https://wiki.archlinux.org/index.php/benchmarking	Benchmark tool dd is used to test the speed of sequential file write.
sysbench	https://wiki.postgresql.org/wiki/SysBench	Benchmark tool sysbench is used to test the speed of sequential and random file I/O.
LIVE555	http://www.live555.com/	The LIVE555 libraries are designed for multimedia streaming, suitable for various cloud-based streaming applications.
FFmpeg	https://www.ffmpeg.org/	FFmpeg is a common multimedia framework that is able to decode, encode, transcode, mux, and demux multimedia files.
pigz	https://zlib.net/pigz/	pigz is a compression utility that exploits multiple processors and multiple cores to the hilt when compressing file data.
Lighttpd	https://www.lighttpd.net/	Lighttpd is a fast and flexible web server implementation with a low memory footprint.

sequential write and random write throughput. Similarly as the read benchmark tests, the Hylics file system outperforms the native block device and has a more stable throughput. During the above experiments, we can find that the in-memory file system provision especially favors for the random read and write operations of small data blocks. Fig. 14 shows the comparison when the VM is writing null characters with the benchmarking tool dd. To ensure the fairness, we also disabled the writing cache and required physically writing the output data before the finish. This is performed by setting the dd parameter "conv=fsync". We found that, the in-memory file system achieves better write throughput than the default virtual disk device in all experiments. Meanwhile, we are also curious about the read and write performance of the in-memory file system at the hypervisor level. We therefore performed the same sysbench tests at the hypervisor layer. The results are listed in Figs. 15 and 16. As a matter of fact, the read and write throughput at the hypervisor level can both achieve gigabit-level performance.

6.3 Real-World Application Performance

We further demonstrate the performance gain brought by the Hylics design with real-world applications. we also used the same hybrid workloads as in the preliminary measurement studies: 1) transcoding and streaming services, 2) file compression and delivery services. In Fig. 17, we show the improvement of the transcoding task completion time brought by the Hylics design. As we have discussed in Section 2, when the transcoding tasks and streaming tasks simultaneously run on the VM (labelled as "Interference" in the figure), we can observe the longest task completion time. Meanwhile, the completion time of stand-alone transcoding tasks is labelled as "No Interference", which serves as the baseline. The completion time, when we offload network I/O modules and socket level operations to the hypervisor layer, is labeled in the figure as "Hylics-module" and "Hylics-socket", respectively. In all the experiments, we can observe that the Hylics design shortens the transcoding task completion time. Furthermore, the performance is fairly close to the "No interference" case.

As a next step, we varied the number of vCPUs assigned to the VM and the total number of threads for the transcoding tasks to investigate the impact of computation power. The results presented in Fig. 18 indicate that Hylics also achieves nearly ideal performance. Note that in the (8 vCPUs, 8 threads) and (4 vCPUs, 4 threads) cases, since the transcoding task runs on all the available vCPUs, such a configuration causes more severe self interference inside the VM, as well as the worst computation performance. The Hylics design, however, has a negligible increase in the task completion time. In these experiments, Hylics exhibits a 7.8-31.2 percent computation performance enhancement when compared with the "Interference" case. We also tested the performance when file compression tasks and file delivery tasks were co-located on the VM. In this case, we varied the size of the target file. The results are shown in Fig. 19. Similarly as in the first experiment, the Hylics design achieves almost the same computation performance as the

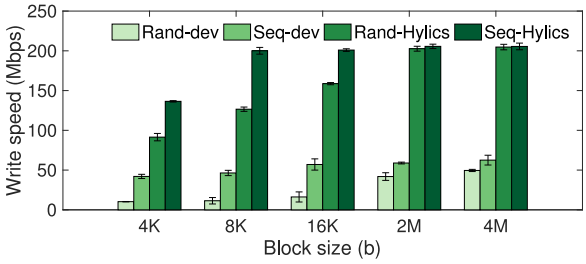


Fig. 13. Sysbench VM write performance.

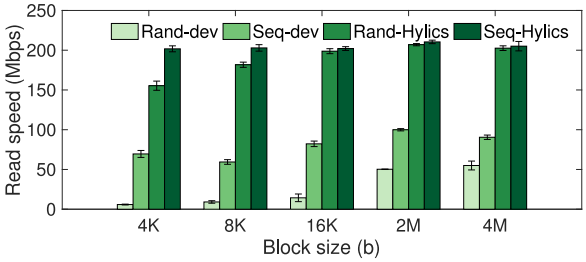


Fig. 12. Sysbench VM read performance.

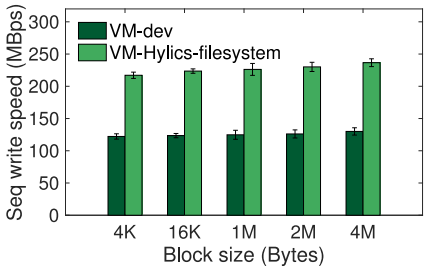


Fig. 14. dd VM write performance.

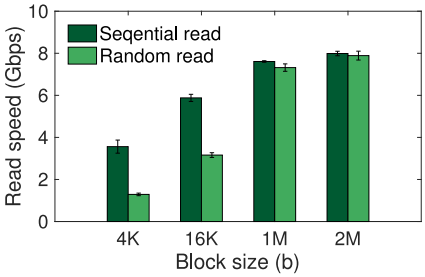


Fig. 15. Host read performance.

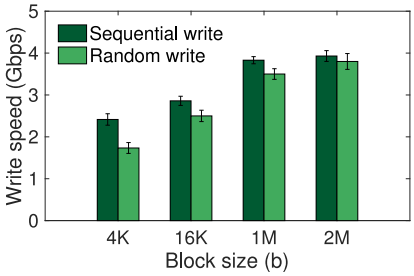


Fig. 16. Host write performance.

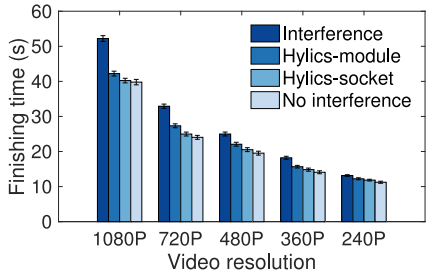


Fig. 17. Video trancoding performance.

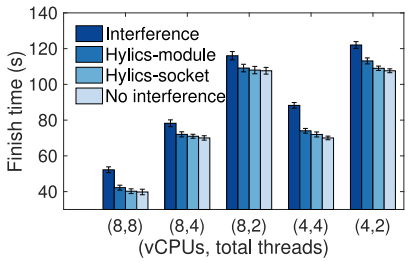


Fig. 18. vCPUs and threads.

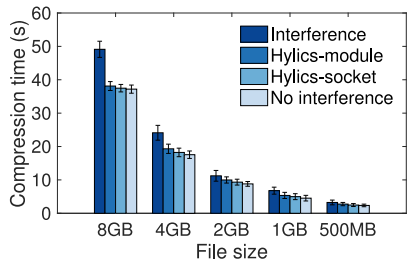


Fig. 19. File compression performance.

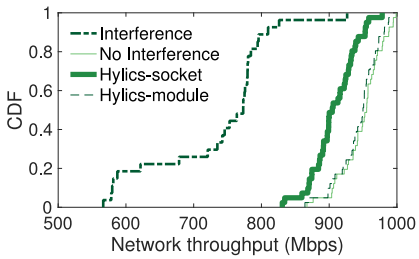


Fig. 20. Streaming server performance.

"No Interference" case. The compression time is improved by up to 26.2 percent in our experiments.

As for the networking performance, in Fig. 20 we present the CDF of the video streaming throughput. In this figure, we use "No interference" to label the stand-alone streaming performance inside the VM. The "Interference" curve describes the streaming throughput when we added concurrent transcoding tasks. By comparing the results between "Interference" and "No Interference", we can observe a 21.4 percent network throughput degradation. In this figure, "Hylics-module" and "Hylics-socket" label the streaming performance when the actual network data transfer is offloaded to the hypervisor level. To summarize, when we applied the Hylics architecture, the average throughput reached up to 931 Mbps, which is close to the "No interference" case. It is noted that the performance of offloading socket level operations to the hypervisor level is slightly worse than directly offloading network I/O modules. This is because the socket level offloading needs to return the sending result to the VM more frequently. In general, Hylics resolves the self interference and achieves a 27.8 percent network throughput enhancement. We also tested the networking performance with the file compression and delivery workload, which is shown in Fig. 21. The comparison also shows the performance enhancement achieved by Hylics.

6.4 Memory Control

To demonstrate the effectiveness of the memory control scheme design, we applied the proposed file system space control algorithm when running the transcoding and streaming experiments. To maintain a reasonable stress on the VM, we selected multiple 10-second 720P video segments as the input. We used two sets of workloads in our tests. The first set, workload #1 is a simple workload which has exponentially distributed inter-arrival time with an average of 8 seconds. The second set, workload #2 is a more complicated workload which periodically changes the average inter-arrival time from 8 seconds to 4 seconds. We consider the following running scenarios: If one video segment has not been requested in the last one minute, it will be evicted from the tmpfs to the standard file system inside the virtual disk space. We further make the assumption that if the tmpfs is temporarily full, then all the arrived transcoding requests will be handled by the file system hosted in the virtual disk space. Based on our real-world measurement in the target VM, when the transcoding task is handled by the in-memory file system, the average task completion time is 3.32 seconds, if the task is handled by the default file system inside the virtual block device, the average task completion time is 4.02 seconds. We then used an offline measurement method to identify the correlation between the file system size m and the average service time $E[X|m]$. The correlation can generally be fitted by an extended inversely proportional function in this case.

To make a fair comparison, we used three different memory control schemes, namely, "adaptive only", "queuing prediction" and the combination of the both. The test lasted for one hour and we set the referenced response time τ to be 10 seconds since it equals the length of the video segments. In addition, we set the parameter $ci\_minthresh$ to be 30 seconds, and set $ci\_maxthresh$ to be 4 minutes. The average response time of the transcoding tasks in this experiment is shown in Figs. 22 and 23. In these two figures, we can see that: (1) At the beginning stage, the adaptive only method needs to gather enough measurement inputs to identify the system performance, which leads to a slow convergence speed; (2) The queueing model provides an approximation of the real system performance. However, there still exists gaps between the queueing model prediction and the real system performance; (3) When combined with the adaptive control scheme, the queueing model helps to set a better starting point to identify the real system performance and hence provides better performance as well as fast convergence. We further present the aggregated response time errors with different settings in Table 3. The results show that combining the queueing model prediction and the self-adaptive control can achieve the least aggregated response time error. Furthermore, with the memory usage control and adjustment, Hylics can operate with a moderate memory usage under dynamic

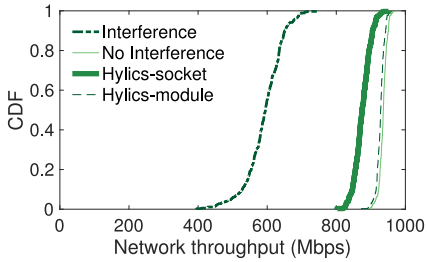


Fig. 21. Web server performance.

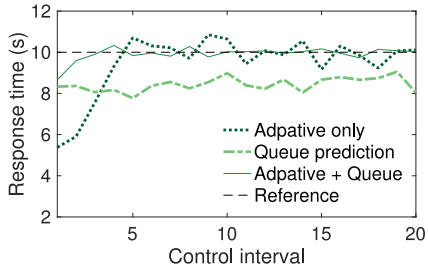


Fig. 22. Hybrid workload #1.

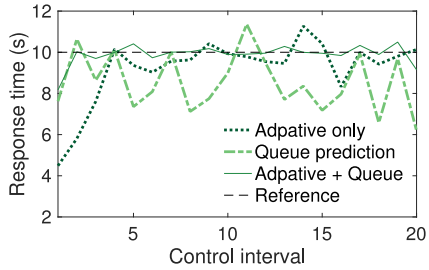


Fig. 23. Hybrid workload #2.

workload stress. The maximum amount of memory used for the file system when handling workload #1 and workload #2 are 245 and 388 MB, respectively.

We also pinpoint the power savings achieved by the Hylics design. We closely measured the CPU power consumption while running the tests. In particular, Fig. 24 shows the results when we only introduced the transmission tasks (video streaming or file delivery) on the VM, with no computation tasks. In this experiment, Hylics is running with the offloaded network I/O module. The power consumption is therefore stable throughout the comparison. The results indicate that, the power consumption of Hylics architecture is much lower when compared with the transmission initiated in the VM. As further shown in Fig. 25, after we initiated the hybrid workloads, the power consumption of the Hylics system is also better than the native virtualized system.

6.5 In-Depth Investigation

To reveal more details, we conducted an in-depth investigation to pinpoint where exactly the gain is from. We collected low-level profiling benchmarks by `perf` when running the stand-alone streaming tasks. We tested four cases: streaming in VM, using network I/O module offloading, using socket level offloading, and the bare-metal case. The results are presented in Table 4. The first benchmark *context-switches per second* refers to the operation when the scheduler determines to run another process or when an interrupt triggers a routine's execution (such as handling network buffer). The measurements of these four cases are 762,991, 1,979, 2,423, and 1,856, respectively, which demonstrates a significant improvement brought by the Hylics design. An explanation on this is that when handling high-volume streaming traffic inside the VM, the streaming process and the traffic handling process keep interrupting each other. Consider if we add the co-located CPU intensive operations and disk I/O operations, the result can get much worse. As a comparison, the Hylics design offloads the actual network data transfer to the hypervisor level, and initiates the data transfer at memory space. As a consequence, the frequent interrupts no longer exist and as is the self interference. The second and third benchmark presented are *stalled cycles at the frontend/backend stage*. A CPU cycle is stalled when the instruction pipeline does not advance during this cycle. In particular, the "frontend stages" are a group of stages during which the instructions are fetched and decoded. During the "backend stages", the instructions are then accordingly executed. From our measurement results, the number of stalled cycles in the frontend has only a 2.14-2.64 percent difference. The number of stalled cycles in the backend, however, has a 15.41-15.62 percent difference. This shows that the streaming process running inside the VM keeps waiting to be scheduled to send out the data. Yet it is often interrupted by the traffic handling process for sending the network buffer. Furthermore, the comparison on *instructions per cycle* and *stalled cycles per instruction* also shows that it is more efficient to shift the I/O intensive operations to the hypervisor layer.

TABLE 3
Aggregated Response Time Error

Workloads	vCPUs	RAM	Adaptive	Queue	A+Q
Type#1	4	16 GB	14.2 s	27.3 s	8.8 s
Type#1	4	32 GB	10.2 s	21.4 s	7.5 s
Type#1	4	64 GB	20.4 s	25.9 s	9.4 s
Type#1	8	16 GB	10.1 s	28.5 s	7.6 s
Type#1	8	32 GB	18.2 s	31.3 s	5.1 s
Type#1	8	64 GB	15.7 s	40.5 s	8.4 s
Type#2	4	16 GB	26.0 s	34.9 s	12.3 s
Type#2	4	32 GB	22.1 s	33.5 s	15.4 s
Type#2	4	64 GB	18.8 s	45.8 s	14.9 s
Type#2	8	16 GB	28.1 s	37.5 s	12.1 s
Type#2	8	32 GB	21.1 s	33.2 s	11.4 s
Type#2	8	64 GB	32.1 s	41.5 s	16.0 s

7 Related Work

In Section 2.2, we have reviewed some recent works on improving the I/O performance in virtualized environments. In addition to those works, there have been other studies in the related fields.

Detecting and Resolving VM Interference. The interference discussed in previous papers comes from the multi-tenancy nature in virtualized cloud environments. The extent of such

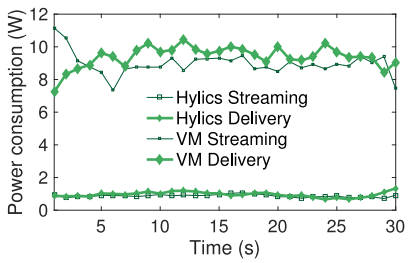


Fig. 24. Transmission tasks.

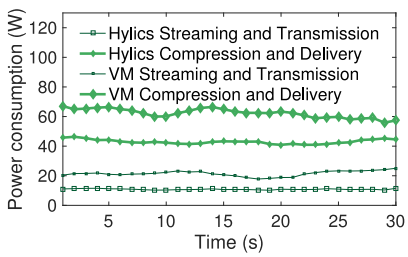


Fig. 25. Hybrid workloads.

TABLE 4
Perf Profiling

Perf statistics	VM stream	Hylics-module	Hylics-socket	Bare-metal
Context switches per second	$762,991 \pm 122$	$1,979 \pm 153$	$2,423 \pm 208$	$1,856 \pm 130$
Stalled cycles frontend	$(67.54 \pm 1.26)\%$ idle	$(70.18 \pm 1.25)\%$ idle	$(69.68 \pm 1.02)\%$ idle	$(72.44 \pm 0.97)\%$ idle
Stalled cycles backend	$(51.93 \pm 0.58)\%$ idle	$(36.31 \pm 0.52)\%$ idle	$(39.52 \pm 0.77)\%$ idle	$(35.52 \pm 0.33)\%$ idle
Instructions per cycle	0.61 ± 0.05	0.97 ± 0.02	0.91 ± 0.03	0.97 ± 0.01
Stalled cycles per instruction	1.10 ± 0.05	0.72 ± 0.03	0.79 ± 0.03	0.72 ± 0.02

interference heavily depends on the combination of workloads running on co-located VMs [28]. Effective profiling and prediction tools have been developed to keep track of such interference [29], [30]. There have been recent efforts towards identifying the impact and pattern of the interference among co-located workloads, as well as developing efficient workload handling strategies [11], [31], [32], [33]. It is noted that these methods are designed for cloud providers, which require detailed runtime information and a global view on the resource management. The techniques include smart scheduling, live migration, and resource containment. Service reconfiguration is also suggested to mitigate such interference from cloud consumers' point of view [13], [14].

Energy Efficiency of Cloud Workloads. Mastelic et al. [34] provided a comprehensive analysis on the energy efficiency of the underlying cloud infrastructure. The survey covers energy efficiency in server domain, supporting management system domain, and appliance domain. There have been studies focusing on the energy consumption of network transactions [35], [36]. We have also seen studies targeting on the optimization of power consumption for specific cloud workloads, e.g., reducing power consumption of data centers through the placement of energy-hungry jobs/VMs [37], [38]. Other pioneer works have explored solutions to measure and cap the energy consumption of VMs running in cloud environments, e.g., *Joulemeter* [39]. The authors also discussed different power consumption models to infer power consumption from resource usage. The power consumed during VM migration is discussed in [40], [41].

Our Hylics design is inspired by these pioneer studies. Yet it targets the self interference inside the VM in cloud environments and also improving the overall energy efficiency when handling hybrid workloads. Our work expands the offloading operations with hypervisor-level in-memory file system sharing. As such, it works well for a wide range of applications, particularly for those involving both intensive data transmission and real-time processing, which can hardly be accommodated by existing tools.

8 Conclusion and Future Work

In this paper, we closely examined the self interference from real-world applications in virtualized environments. To jointly optimize performance and energy efficiency for hybrid workloads in cloud environments, we designed and developed *Hylics*, a novel protocol-independent solution that leverages hypervisor-level in-memory file system sharing. We implemented a prototype of Hylics in KVM and evaluated the overall performance with real-world workloads. The experiment results indicate that such a design can largely improve I/O performance and accelerate computation tasks in the presence of the self interference. The energy efficiency of the underlying server is also enhanced. In the future work, we plan to implement Hylics-based solutions on other virtualization platforms. Since Hylics significantly minimizes the self interference, we will also revisit the existing VM resource allocation strategies to help cloud providers to achieve better service performance and cost efficiency.

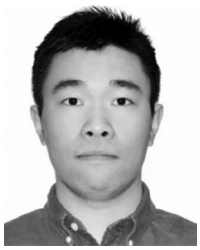
Acknowledgments

This work is supported by an Industrial Canada Technology Demonstration Program and an NSERC Discovery Grant.

References

[1] A. J. Younge, R. Henschel, J. T. Brown, et al., "Analysis of virtualization technologies for high performance computing environments," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2011, pp. 9–16.
[2] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *Proc. IEEE INFOCOM*, 2014, pp. 1285–1293.
[3] A. Gordon, N. Amit, N. Har'El, et al., "ELI: Bare-metal performance for I/O virtualization," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 411–422, 2012.
[4] N. Har'El, A. Gordon, A. Landau, et al., "Efficient and scalable paravirtual I/O system," in *Proc. USENIX Annu. Techn. Conf.*, 2013, pp. 231–242.
[5] J. Hwang, K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manage.*, vol. 12, no. 1, pp. 34–47, Mar. 2015.
[6] S. Gamage, D. X. R. Kompella, and A. Kangarlou, "Protocol responsibility offloading to improve TCP throughput in virtualized environments," *ACM Trans. Comput. Syst.*, vol. 31, pp. 1–34, 2013.
[7] S. Gamage, C. Xu, R. R. Kompella, and D. Xu, "vPipe: Piped I/O offloading for efficient data movement in virtualized clouds," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.
[8] C. Xu, B. Saltaformaggio, S. Gamage, R. R. Kompella, and D. Xu, "vRead: Efficient data access for hadoop in virtualized clouds," in *Proc. ACM Annu. Middleware Conf.*, 2015, pp. 125–136.
[9] A. Gordon, M. Ben-Yehuda, D. Filimonov, and M. Dahan, "VAMOS: Virtualization aware middleware," in *Proc. 3rd Conf. I/O Virtualization*, 2011, pp. 3–3.
[10] C. Xu, S. Gamage, H. Lu, et al., "vTurbo: Accelerating virtual machine I/O processing using designated turbo-sliced core," in *Proc. USENIX Annu. Techn. Conf.*, 2013, pp. 243–254.
[11] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for QoS-aware clouds," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2010, pp. 237–250.
[12] X. Chen, L. Rupprecht, R. Osman, et al., "CloudScope: Diagnosing and managing performance interference in multi-tenant clouds," in *Proc. IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2015, pp. 164–173.
[13] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "Mitigating interference in cloud services by middleware reconfiguration," in *Proc. ACM Int. Middleware Conf.*, 2014, pp. 277–288.
[14] A. K. Maji, S. Mitra, and S. Bagchi, "ICE: An integrated configuration engine for interference mitigation in cloud services," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2015, pp. 91–100.
[15] C. Xu, X. Ma, R. Shea, H. Wang, and J. Liu, "MemNet: Enhancing throughput and energy efficiency for hybrid workloads via paravirtualized memory sharing," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2016, pp. 980–983.

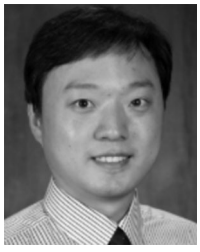
[16] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," *Elsevier J. Parallel Distrib. Comput.*, vol. 72, no. 11, pp. 1471–1480, 2012.
[17] J. P. Billaud and A. Gulati, "hClock: Hierarchical QoS for packet scheduling in a hypervisor," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2013, pp. 309–322.
[18] P. Snyder, "tmpfs: A virtual memory file system," in *Proc. Autumn Eur. UNIX Users Group Conf.*, 1990, pp. 241–248.
[19] R. Russell, "virtio: Towards a De-Facto standard for virtual I/O devices," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.
[20] V. Jujjuri, E. V. Hensbergen, A. Liguori, and B. Pulavarty, "VirtFS-A virtualization aware file system pass-through," in *Proc. Ottawa Linux Symp.*, 2010, pp. 109–120.
[21] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86," in *Proc. USENIX Annu. Techn. Conf.*, 2008, pp. 293–306.
[22] B. Yee, D. Sehr, G. Dardyk, et al., "Native client: A sandbox for portable, untrusted x86 native code," in *Proc. IEEE Symp. Secur. Privacy*, 2009, pp. 79–93.
[23] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster usage traces: Format + Schema," Google Inc., Technical Report, revised 2014-11-17 for version 2.1. Nov. 2011. [Online]. Available: https://github.com/google/cluster-data
[24] S. Di, D. Kondo, and F. Cappello, "Characterizing and modeling cloud applications/jobs on a Google data center," *J. Supercomput.*, vol. 69, no. 1, pp. 139–160, 2014.
[25] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proc. IEEE*, vol. 102, no. 1, pp. 11–31, Jan. 2014.
[26] D. Bruneo, A. Lhoas, F. Longo, and A. Puliafito, "Modeling and evaluation of energy policies in green clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 3052–3065, Nov. 2015.
[27] K. J. Åström and B. Wittenmark, *Adaptive Control*. Chelmsford, MA, USA: Courier Corporation, 2013.
[28] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao, "Who is your neighbor: Net I/O performance interference in virtualized clouds," *IEEE Trans. Serv. Comput.*, vol. 6, no. 3, pp. 314–329, Jul.–Sep. 2013.
[29] Q. Zhu and T. Tung, "A performance interference model for managing consolidated workloads in QoS-aware clouds," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2012, pp. 170–179.
[30] A. O. Ayodele, J. Rao, and T. E. Boult, "Performance measurement and interference profiling in multi-tenant clouds," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2015, pp. 941–949.
[31] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI^2: CPU performance isolation for shared compute clusters," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2013, pp. 379–391.
[32] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov, "Stay-away, protecting sensitive applications from performance interference," in *Proc. ACM Int. Middleware Conf.*, 2014, pp. 301–312.
[33] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: Transparently identifying and managing performance interference in virtualized environments," in *Proc. USENIX Annu. Techn. Conf.*, 2013, pp. 219–230.
[34] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J. M. Pierson, and A. V. Vasilakos, "Cloud computing: Survey on energy efficiency," *ACM Comput. Surveys*, vol. 47, no. 2, 2015, Art. no. 33.
[35] R. Shea, H. Wang, and J. Liu, "Power consumption of virtual machines with network transactions: Measurement and improvements," in *Proc. IEEE INFOCOM*, 2014, pp. 1051–1059.
[36] C. Xu, Z. Zhao, H. Wang, R. Shea, and J. Liu, "Energy efficiency of cloud virtual machines: From traffic pattern and CPU affinity perspectives," *IEEE Syst. J.*, vol. 11, no. 2, pp. 835–845, Jun. 2017.
[37] D. Kusic, J. Kephart, J. Hanson, K. Nagarajan, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2008, pp. 3–12.
[38] K. Ye, Z. Wu, C. Wang, B. B. Zhou, W. Si, X. Jiang, and A. Y. Zomaya, "Profiling-based workload consolidation and migration in virtualized data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 878–890, Mar. 2015.
[39] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya, "Virtual machine power metering and provisioning," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 39–50.
[40] H. Liu, C. Xu, H. Jin, J. Gong, and X. Liao, "Performance and energy modeling for live migration of virtual machines," in *Proc. ACM Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 171–182.
[41] Q. Huang, F. Gao, R. Wang, and Z. Qi, "Power consumption of virtual machine live migration in clouds," in *Proc. IEEE Int. Conf. Commun. Mobile Comput.*, 2011, pp. 122–125.



Chi Xu (S'14) received the BSc degree in software engineering from Xidian University, Xi'an, China, in 2013 and the MSc degree in computing science from Simon Fraser University, in 2016. His current research interests include computer and network virtualization, performance issues in cloud computing, and hardware support for big data applications. He is a student member of the IEEE.

Xiaoqiang Ma (S'12-M'16) received the BEng degree from the Huazhong University of Science and Technology, Wuhan, China, in 2010, and the MSc and PhD degrees from Simon Fraser University, Burnaby, BC, Canada, in 2012 and 2015, respectively. He is currently an assistant professor with the School of Electronic Information and Communication, Huazhong University of Science and Technology. His research interests include wireless networking, multimedia, cloud, and big data. He is a member of the IEEE.

Ryan Shea (S'08-M'16) received the BSc and PhD degrees in computer science from Simon Fraser University, Burnaby, BC, Canada, in 2010 and 2016, respectively. He is currently a University research associate with Simon Fraser University, where he also completed the Certificate in University teaching and learning. His research interests include computer and network virtualization and performance issues in cloud computing. He is a member of the IEEE.



Haiyang Wang (S'08-M'13) received the PhD degree in computing science from Simon Fraser University, Burnaby, BC, Canada, in 2013. He is currently an assistant professor with the Department of Computer Science, University of Minnesota Duluth, Minnesota. His research interests include cloud computing, big data, socialized content sharing, multimedia communications, and peer-to-peer networks. He is a member of the IEEE.

Jiangchuan Liu (S'01-M'03-SM'08-F'17) received the BEng (Cum Laude) degree from Tsinghua University, Beijing, China, in 1999, and the PhD degree from the Hong Kong University of Science and Technology, in 2003, both in computer science. He is currently a full professor (with University Professorship) with the School of Computing Science, Simon Fraser University, BC, Canada. He is a Steering Committee member of the *IEEE Transactions on Mobile Computing*, and associate editor of the *IEEE/ACM Transactions on Networking*, the *IEEE Transactions on Big Data*, and the *IEEE Transactions on Multimedia*. He is a co-recipient of the Test of Time Paper Award of IEEE INFOCOM (2015), ACM TOMCCAP Nicolas D. Georganas Best Paper Award (2013), and ACM Multimedia Best Paper Award (2012). He is a fellow of the IEEE and an NSERC E.W.R. Steacie Memorial fellow.