QuartzV: Bringing Quality of Time to Virtual Machines

Sandeep D'souza and Ragunathan (Raj) Rajkumar
Carnegie Mellon University
sandeepd@andrew.cmu.edu, rajkumar@andrew.cmu.edu

***Abstract*—Cyber-physical systems are increasingly interconnected and distributed. Examples range from factory-scale industrial robotics to regional-scale smart grids. Therefore, to enable dynamic coordination at scale among geo-distributed physical endpoints, the *intelligence* behind these systems will often be hosted in the cloud. However, most CPS applications are inherently safety-critical, and require low-latency responses. Hence, a hierarchy of edge *cloudlets* and the cloud can be used to offload computationally and data-intensive workloads. While low latency is key, a shared sense of time with the added notion of *Quality of Time* (QoT) is useful for fault detection, and enables fault-tolerant coordinated action in distributed CPS [1][2]. Given that most public clouds and cloudlets provide multi-tenancy using *virtualized* units of computing, we aim to introduce the notion of QoT to virtual machines. The use of virtual machines entails the use of a hypervisor, which adds additional timing uncertainty due to relatively higher jitter in clock-read and timer-interrupt latencies. Hence, the use of virtualization presents a challenge in terms of observing and guaranteeing the QoT delivered to an application. To meet these challenges, we present the QuartzV extension to the QoT Stack for Linux [1], to make virtual machines QoT-aware. We utilize the open-source QEMU-KVM [3] hypervisor, and illustrate the *para-virtual* design choices that are key for delivering near-native levels of *timing* performance in virtual machines. We also demonstrate the utility of QuartzV by using it in the development of an industrial-automation application. Experimental evaluations also show the efficacy of QuartzV with respect to the native and fully-virtualized cases.**

I. Introduction

Emerging Cyber-Physical Systems (CPS) involve the control and manipulation of the environment using multiple distributed entities. These systems range from factory-scale industrial robotics [4] and city-scale connected vehicles [5] to regional/national-scale power grids [6]. To operate efficiently, all of these systems require coordinated *actuation* (or action) among numerous nodes spread out over possibly large geographical regions. The nature of this coordination is usually dependent on the analysis of *sensed* data by an intelligent *computational* entity, which may utilize techniques ranging from signal processing to machine learning. The data-intensive nature of CPS makes the cloud well-suited to host this computational *intelligence* [2]. Alternatively, a hierarchy of edge cloudlets [7] and the cloud may be utilized to meet the low-latency requirements of CPS.

Time plays a key role in enabling coordination among multiple entities [8]. This coordination occurs at different timescales ranging from human coordination at the order of seconds, to the sub-nanosecond level coordination among Global Positioning System (GPS) satellites. A non-exhaustive list of such coordinated systems includes sensor networks [9], swarm robotics [10], tele-surgery [11], distributed databases [12], and large-scale scientific experiments [13]. Therefore, maintaining a shared notion of time is critical to the performance and reliable operation of many distributed systems.

Clock-synchronization technologies such as GPS, Network Time Protocol (NTP) [14], Precision Time Protocol (PTP) [15], and hardware timestamping [15] have made it possible to provide distributed systems with a reliable and accurate shared notion of time. However, other technology trends have made it harder for applications to benefit from these advances. For example, asymmetric networking delays [14] and abstractions like virtual machines introduce greater timing uncertainty [16]. This timing uncertainty can affect the quality and reliability of coordination [2]. The level of uncertainty acceptable to an application often depends on the time granularity at which coordination occurs, as well as the coordination policy [2].

Fault-tolerant time-based coordination can be enabled by using the notion of *Quality of Time* (QoT) [1], which represents the end-to-end uncertainty in the notion of time delivered to an application by the system. Thus, QoT represents the uncertainty bounds corresponding to a timestamp, with respect to a clock reference. From an application perspective, if these uncertainty bounds exceed an acceptable limit, the application can enter a graceful-degradation mode, and thus be fault-tolerant. Based on the notion of QoT, [1] also introduced a reference QoT Architecture along with its corresponding implementation for Linux, called the QoT Stack for Linux.

To enable scalable time-based cyber-physical coordination, it is essential that we engineer a QoT-aware cloud/edge-cloudlet infrastructure [2]. However, to maintain application isolation, most public clouds and cloudlets provide multi-tenancy using *virtualized* units of computing. These maybe Virtual Machines (VMs) [17] or application containers [18]. Additionally, the use of virtualization for consolidation of multiple real-time systems on a single platform is also of increasing interest [19]. Motivated by these needs, this work focuses on bringing the notion of QoT to the dominant virtualization technology, namely virtual machines. We design and implement the QuartzV extension to the QoT Stack for Linux for introducing the notion of QoT to Linux VMs running atop the open-source QEMU-KVM [3] hypervisor.

The contributions of this paper are as follows:

1) Elucidating the challenges and subsequent architectural choices in bringing QoT to Virtual Machines.
2) Introducing the QuartzV extensions for Linux VMs which support para-virtual clocks.
3) Porting the QoT Stack for Linux to VMs and hypervisors which do not support para-virtual clocks.

4) Evaluating and comparing the performance and scalability of the para-virtual QuartzV approach against the native and fully-virtualized scenarios.

The rest of the paper is organized as follows. Section II provides a background of virtualization technologies, and time-based coordination using QoT. Section III provides insight into developing time-based coordinated applications using QuartzV. Section IV introduces the QuartzV extension to the QoT Stack for Linux. Section V provides experimental evaluations, and Section VI concludes the paper.

II. Background and Related Work

We now introduce the background relevant to this work.

A. Virtualization

Virtualization is often used to share physical hardware resources among multiple users, while providing the illusion that every user has access to his/her own machine [20]. To support this illusion, it is important that, (i) virtualized units are well-isolated from other users [20], and (ii) the overhead of virtualization is low [20]. These objectives are often conflicting, and virtualization technologies generally trade-off one of the objectives in favor of the other. For example, hypervisor-based virtual machines [3][21] offer strong isolation by trading off some performance due to the overhead of the hypervisor. On the other hand, operating-system level virtualization [18] (also known as *containerization*) trades off some level of isolation for performance by eliminating the hypervisor.

In this paper, we focus on hypervisor-based virtual machines (VMs). Modern hypervisors generally take advantage of *hardware-accelerated* virtualization, based on hardware extensions like Intel VT-x [22] and AMD-V [23]. These technologies enable VMs to execute unprivileged CPU instructions natively, while privileged instructions are serviced using the trap and emulate mechanism [22]. On the other hand, *para-virtualization* [20] enables low-latency access to peripherals and I/O devices, such as network interfaces, disks and clocks, also delivering near-native performance. This access is made possible by para-virtual drivers [20] which can directly perform protected access to the hardware through the hypervisor. For systems which do not support hardware acceleration or VMs which lack para-virtual drivers, all CPU instructions or peripheral-device access must be emulated in the hypervisor. This is also referred to as *full virtualization* [20]. In practice, modern hypervisors generally utilize a mixture of para-virtualization and hardware-accelerated virtualization [3][21] to provide near-native levels of performance.

B. Time and Virtualization

The use of hypervisor-based VMs introduces an additional abstraction layer between applications and the hardware. This translates to additional timing uncertainty, due to higher jitter in clock-read and interrupt-servicing latencies [16]. Therefore, in [16], the authors experimentally characterize the timekeeping properties of the Xen hypervisor [20]. Their work highlights the weaknesses of the existing timing solution in Xen,

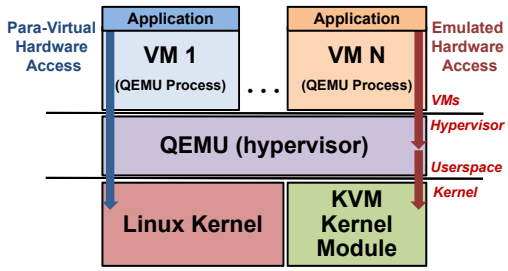


Fig. 1. The QEMU-KVM Hypervisor. VM 1 supports para-virtual peripheral access, while VM N utilizes peripheral emulation (full virtualization)

which uses independent NTP [14] synchronization sessions for each guest VM. They refer to this as the *independent clock* paradigm, where each VM independently performs clock synchronization. The authors note that this practice is wasteful of system resources, and degrades synchronization accuracy. Additionally, the authors also find the practice of keeping clock-synchronization state in the VM detrimental for live migration. Hence, the authors propose a *dependent clock* solution based on the *RADclock* [24] feed-forward synchronization algorithm. Each VM has a dependent clock, which is sourced from the hardware clock on the host machine. Hence, each VM has access to the para-virtualized hardware clock exposed in the host OS. This clock is disciplined in the host OS, and thus only one synchronization service is required per host machine. Apart from being resource-efficient, as VMs now do not contain synchronization state, the dependent-clock paradigm also aids VM live migration. Hence, if a VM is migrated, it does not need to re-synchronize its clock. Rather, it can derive the time from the hypervisor at the new host.

The authors in [16] conclude that the para-virtualized dependent clock is useful for VMs. However, the authors do not consider the utility of exposing timing uncertainty information. Additionally, the recent evolution of hypervisors and the advent of hardware-accelerated virtualization offers a fresh opportunity to re-visit the problem of time and virtualization.

C. Kernel-Based Virtual Machine (KVM)

We focus on one commodity open-source hypervisor, namely Kernel-based Virtual Machine [3], also referred to as QEMU-KVM. However, the core concepts of this work are applicable to other commercial and open-source hypervisors. Figure 1 shows the organization of the QEMU-KVM hypervisor, and illustrates how virtual machines interact with its components. QEMU-KVM consists of two core components:

1. QEMU Emulator functions as a hypervisor, and each VM runs as a QEMU process. QEMU can be used for full virtualization (all instructions emulated), or hardware-accelerated virtualization (only privileged instructions emulated). In addition, QEMU also provides VMs with para-virtualized access to host peripherals (such as disks, I/O devices, network interfaces and clocks). For VMs which do not support para-virtualization, QEMU also provides peripheral-device emulation.

2. KVM Loadable Kernel Module enables QEMU to interface with the Linux kernel. This allows QEMU-KVM to use existing kernel functionality for resource management

(such as scheduling and isolation). Additionally, the KVM kernel module also enables the hypervisor to take advantage of hardware extensions like Intel VT-x and AMD-V.

In terms of clock support, QEMU-KVM provides the para-virtual `KVM-clock` [25] to Linux VMs. This allows a para-virtual guest VM to access the host's monotonic system clock (`CLOCK_MONOTONIC`) and real-time clock (`CLOCK_REALTIME`). On the x86 architecture, `KVM-clock` uses the Time-Stamp Counter (TSC) [26], and a memory page mapped into the VM's virtual-memory space to provide low-latency clock reads. Whenever the VM is scheduled, the hypervisor writes the current time (monotonic and real-time), and corresponding TSC value into this page. The VM can then use this timestamp along with reading the current TSC value to calculate the current time. Given that both reading the TSC (`rdtsc`) and accessing a memory address are non-privileged operations, a para-virtual guest VM can perform low-latency clock reads. For VMs which do not support para-virtualization, QEMU-KVM provides access to emulated timers [3].

D. Quality of Time Architecture

The knowledge of QoT enables applications to adapt and be fault-tolerant [2], while allowing the system to manage resources efficiently [1]. Building on QoT, the QoT Architecture [1], centered around a shared notion of time, allows applications to specify their timing requirements, while delivering the required QoT and exposing timing uncertainty to applications.

In [1], the authors also introduce the *timeline* abstraction, which abstracts away low-level synchronization details from applications. This allows the underlying framework to orchestrate the clock-synchronization service to ensure that QoT requirements are met, while making the achieved QoT visible to the application. Additionally, a timeline is not necessarily tied to any standard timing reference, and in the context of coordination, serves as a "narrow waist". This enables developers to easily develop distributed time-based applications on heterogeneous infrastructure, using a timeline-based API.

Consider an application that needs to perform coordinated actions by its distributed endpoints. All of these components bind to a common timeline, each specifying its respective QoT requirements. The QoT Architecture supports multiple timelines. This enables different coordinating sub-groups with varying QoT requirements to each have its own time reference and co-exist on the same infrastructure [1].

The QoT Architecture consists of three key components:

1) **Clocks** are used to expose the time-stamping and time-keeping capabilities of the underlying hardware. Clocks also expose their intrinsic parameters like accuracy, precision and drift, which enables uncertainty estimation.
2) **System Services** are responsible for clock synchronization, distributing timeline meta-data, message passing and quantifying timing uncertainties.
3) **QoT Core** acts as a bridge between QoT-aware applications, system services and the OS. It maintains synchronization and timeline state, and is also responsible for event scheduling.

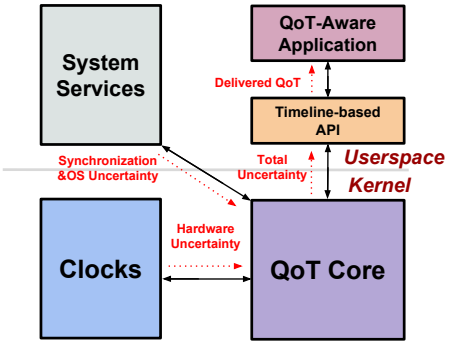


Fig. 2. Timeline-based QoT Architecture [1]

The main components of the QoT Architecture are illustrated in Figure 2. Based on this architecture, a prototype open-source QoT Stack for Linux was introduced in [1]. This preliminary stack focused on implementing necessary functionality over a LAN. The current version of our stack supports Linux-based platforms (ARM and x86 architectures), with clock-synchronization support for PTP [15] and NTP [14] over Ethernet. The stack consists of kernel modules and system services, and does not modify the Linux kernel. The extensibility of our stack along with the ubiquity of Linux make it usable on a wide variety of platforms.

The use of virtualization in cyber-physical applications presents a challenge in terms of both guaranteeing a certain level of QoT, as well as observing the QoT delivered to an application. Hence, we explore solutions to efficiently introduce the notion of QoT to hypervisor-based virtualization.

III. Time-Based Applications using QoT

Before describing QuartzV, we motivate its utility by describing an application which can be enabled by using virtualization and QoT. Although the application described is from the industrial-automation domain, the core concepts can be adapted for other coordinated distributed application domains.

Consider an industrial-automation application, where multiple robotic arms are used to collaboratively assemble a mechanical assembly. Collaborative manufacturing is often required to: (i) speedup assembly (e.g. performing parallel assembly), and (ii) perform joint tasks which maybe too large for a single robot to operate on (e.g. cooperatively picking and placing a large part onto the main assembly). To successfully perform collaborative manufacturing, we need to ensure that the robotic arms are coordinated such that, (i) when performing parallel tasks, they do not interfere with the proper functioning of each other, or operate in/on the same physical space, and (ii) when performing joint tasks, they coordinate their actuations (actions) to successfully complete the task. While these coordination scenarios can be carried out using extensive message passing, the overhead involved is high. This message-passing overhead prevents a system from scaling to multiple endpoints. Often, industrial systems are over-engineered or hard-coded to achieve such tasks, which limits the capabilities and flexibility of the system.

An alternative approach is to use a shared notion of time as a primitive for coordination [1][8]. In this case, an intelligent

centralized/distributed task planner with a view of the entire system can dynamically generate action commands with a corresponding action timestamp, based on shared time. The endpoints of the system can then execute these actions at the planned time points. However, given that industrial systems are often safety-critical, a fault-detection primitive such as QoT is needed to handle the case of clock-synchronization failure [2].

By specifying the required QoT, each component in the system knows the maximum level of uncertainty tolerable to perform successful coordination. Since each node independently maintains its own notion of QoT with respect to the reference, a node can enter a graceful-degradation [29] mode when the level of uncertainty exceeds the tolerable limit. Additionally, if a coordination message is delayed or arrives too late, all a node needs to do is compare the action timestamp against the current time on its local clock [8]. Based on this timestamp, the endpoint can adapt or enter a graceful degradation mode. Given that modern oscillators drift slowly, the probability of clock-synchronization failure is much lower than the probability of CPUs, networks or disks failing [12]. Therefore, using a shared notion of time with QoT can enable scalable and fault-tolerant coordination [2].

Based on the philosophy of QoT, Figure 3 illustrates a collaborative-assembly scenario using two robot arms. The system consists of (i) a centralized task planner running in a VM (using QuartzV) hosted on a server machine, (ii) two embedded-grade arm controller nodes (using the QoT Stack for Linux), and (iii) two robot arms with sensors (to determine state), an end effector, and real-time motor controllers.

We now describe this system in a top-down fashion:

1) **Task Planner in a VM:** is able to receive timestamped sensory input from the sensors on the robotic arms, and can be based on techniques including signal-processing, machine learning [30], or model-based artificial intelligence [31]. Based on the system objective, sensory inputs, and the state of the system, the task-planner can generate receding-horizon-based [32] timestamped action commands for the robotic arms to perform the collaborative assembly. In the context of the example application, the action commands can be in the form of (i) position of the end-effector, and (ii) "pick" or "place" actions of the end-effector. These action commands are received by the "controller" nodes.

2) **Controller Nodes:** are responsible for generating a time-parametrized feasible motion plan for their respective robotic arms based on the action commands, and the sensory inputs from the arm. This requires converting the coarse-grained end-effector trajectory into feasible fine-grained joint-motion trajectories or end-effector actions, such that collisions are avoided. These embedded controller nodes also contain I/O ports which enable them to directly interface with their respective robotic arms with low latency.

3) **Robot Arms:** contain on-board low-level real-time motor controllers which are responsible for carrying out the motion plan received from their respective controller nodes.

In the described system, the task-planner node (VM) and the controller nodes are inter-connected using a switched

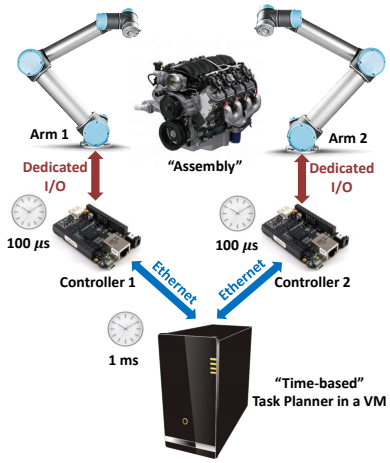


Fig. 3. Time-based Coordinated Industrial Automation

Ethernet network. All these three nodes use the QoT Stack functionality to bind to a common timeline with their specified QoT requirements (+/-1 ms for the task-planner node, and +/-100 μs for the controller nodes). The synchronization service (based on PTP [15]) can then discipline the clocks to meet the specified QoT requirements. Notably, there is no need for the robot arms to directly join the timeline. This is because the on-board motor controllers of the robot arm can perform real-time control with deterministic latency [33][34]. Additionally, sensor values can also be accessed with deterministic latency. This assumption holds true for most industrial robots. Thus, due to the presence of dedicated controllers and interfaces, the robots can carry out the motion plan specified by the embedded controller in deterministic fashion. Additionally, using the dedicated I/O interface, the embedded-controller node can read the robot's sensors with deterministic latency, and hence assign timestamps using its own local timeline reference.

For this paper, our objective is to use QuartzV in the design of scalable and fault-tolerant coordinated applications, like the above, using a shared notion of time and QoT.

IV. QuartzV Extension to the QoT Stack

In this section, we describe the design choices involved in bringing QoT to hypervisor-based virtualization. Subsequently, we present the QuartzV extension to the QoT Stack for Linux to provide QoT awareness for para-virtual guest VMs running atop the QEMU-KVM hypervisor. QuartzV adds extensions to the QEMU-KVM hypervisor, in order to provide *clock-synchronization-as-a-service* to para-virtual guests. This enables applications running in a guest VM to specify their QoT requirements, while a host service tries to meet the specified requirements, and feeds back the achieved QoT to the application running in the guest VM.

We first discuss the applicability of QuartzV in a para-virtualized setting, and later discuss how our implementation works in an environment where clocks are fully virtualized (emulated), or the hypervisor cannot provide clock-synchronization extensions. While QuartzV has been implemented for QEMU-KVM, the concepts are readily applicable to other commodity open-source hypervisors like Xen [21].

A. QoT and Virtualization

While designing QuartzV for a para-virtual setting, the following design considerations need to be taken into account:

1) Specifying QoT Requirements: To provide QoT awareness in the virtualization context, applications need to be able to specify their QoT requirements. Hence, we need to develop a mechanism to allow applications running in a VM to convey their QoT requirements to a service running on the host OS. Since specifying QoT requirements is not on the critical path of most applications, we can afford a somewhat higher-latency communication mechanism for this purpose.

2) Exposing QoT to Applications: To expose the notion of QoT to applications, every timestamp read should contain its associated uncertainty. Reading timestamps from a clock is often on the critical path of most applications. Hence, we must provide a timestamp along with its associated uncertainty, with low latency. For this purpose, we require an efficient low-latency mechanism which can transfer a timestamp, along with the achieved QoT from the host to the guest VM.

3) Supporting Multiple VMs: The key idea of virtualization is to consolidate multiple VMs on a single physical machine. Hence, it is imperative that our implementation scale to support multiple VMs without any impact on performance.

4) Maintaining VM Isolation: While workload consolidation is key, isolation between different VMs is essential. Hence, our implementation should prevent malicious VMs from affecting the correct operation of other VMs.

5) Portability: We aim to implement our system such that no modification is made to the hypervisor source code. Instead, we use existing hypervisor functionality to implement extensions. This ensures that our implementation is portable across different versions of the QEMU-KVM hypervisor.

B. QuartzV: Design and Implementation

Based on these considerations, we now present the design of QuartzV, which builds upon the previously proposed QoT Stack for Linux [1], to bring the notion of QoT to VMs. The key components of QuartzV are as follows:

1) QoT Application Library: Also known as `qotlib`, it provides QoT-specific functionality to user-space applications. This library exposes timeline-based distributed coordination APIs, that are independent of the platform and OS. The APIs enable applications to (i) bind/unbind from a timeline, (ii) specify/update their QoT requirements, (iii) schedule events based on shared time, (iv) timestamp events, and (v) support publish/subscribe messaging for coordination [2]. All API calls return the QoT actually delivered to the application, providing the ability to adapt to changes in QoT [1]. This library can be configured with a compilation flag to enable para-virtual guest-related functionality. This allows native applications to be ported to a VM without any changes to the source code.

2) QoT Core Kernel Module: It acts as a bridge between the components of the QoT Stack for Linux [1], and is responsible for timeline management, clock management and time-based event scheduling. Applications and system services interact with the QoT Core using an `ioctl` interface exposed over the `/dev/qotusr` character device. Both the host and guest VMs contain their own QoT Core module. The QoT core's scheduling interface is policy-agnostic [1], and is responsible for moving tasks from the scheduler wait-queue to the run-queue at the specified time. This provides developers the flexibility to choose the appropriate real-time scheduling policy based on application priorities and requirements.

3) QoT Clocks: These are useful for maintaining a shared notion of time, and also aid in performing clock synchronization over a network [1]. The core clock [1] is used to maintain a monotonic free-running (drift not disciplined) notion of time. Each timeline-reference clock `/dev/timelineX` (where X is the timeline id) is mapped from the core clock (on the host) using the parameters tl_{skew} (drift correction), $core_{last}$ (the core-clock timestamp at the last synchronization event) and tl_{last} (timeline-reference timestamp at the last synchronization event). Using the current core timestamp, $core_{now}$, the timeline-reference time, tl_{now}, can be projected as follows:

$$tl_{now} = tl_{last} + tl_{skew} * (core_{now} - core_{last}) \quad (1)$$

4) Synchronization Service: This is deployed on the host, and synchronizes the local timeline clock, derived from the local monotonic clock source, with the global timeline reference. We use feed-back synchronization to discipline the clock on a per-timeline basis. This service polls the timeline clock over `/dev/timelineX` (where X is the timeline id) to detect any updates to application QoT requirements. Based on these application requirements, the service disciplines the timeline-reference clock by modifying its parameters (drift and offset) to achieve the desired levels of QoT. In doing so, the synchronization service periodically updates the clock mapping parameters and associated timestamp uncertainty lower and upper bounds, ϵ_l and ϵ_h, on a per-timeline basis.

5) Inter-VM Shared-Memory Server: Also known as `ivshmem_server`, this is deployed on the host, and creates a POSIX shared-memory region which is used to distribute clock parameters and timestamp-uncertainty information to applications running in guest VMs.

6) QoT Virtualization Service: Also referred to as `qot_virtd`, this is deployed on the host, and aggregates application-specific QoT requirements from different guest VMs hosted on the host machine. It creates a Unix socket, and acts as a server, while the guest VMs are its clients. Applications running in a VM can send their timeline information and QoT requirements to `qot_virtd` using the created socket. Additionally, whenever the synchronization service updates the clock parameters and estimated timing uncertainty of a given timeline reference, `qot_virtd` is responsible for conveying these changes back to the application, using the shared-memory region created by `ivshmem_server`.

Using the above components, we now describe their interactions which facilitate the transfer of QoT and timeline-related information between the applications deployed inside guest VMs and the services running on the host.

1) Specifying QoT Requirements (Guest VM to Host): To transfer application-specific QoT requirements from the

guest to the host, we utilize the para-virtual VirtIO-serial interface, also referred to as `virtserial` [27]. VirtIO-serial provides bi-directional serial communication between applications running inside guest VMs with a host service. This interface is exposed to the guest application through a QEMU character-device driver front-end in the VM. Using an API, guest applications can read from or write messages to the character-device front-end. Since each VM runs as a QEMU process, the QEMU backend can forward guest application messages to a specified service on the host over a Unix socket. When an application in a VM binds to a timeline, the information is sent to `qot_virtd` using `virtserial` via the socket interface. The daemon then creates a version of the timeline on the host (using the QoT Core kernel module [1]), and registers the QoT requirements of the application with the host OS. `qot_virtd` also sends an acknowledgment to the guest application to indicate if the request was successfully accepted. Figure 4 highlights this interaction of each guest VM application with `qot_virtd`, and illustrates the transfer of application QoT requirements from a guest (VM 1) to the host using VirtIO-Serial. Although our stack supports multiple VMs, for the purpose of illustration, we show only one VM.

2) Facilitating Low-Latency Clock Reads: In QuartzV, we utilize the para-virtualized dependent-clock paradigm and perform clock synchronization on the host on a per-timeline basis. Hence, we maintain a monotonic free-running core clock on the host, and compute its disciplining parameters (drift and offset), with respect to a global timeline reference. These disciplining parameters allow us to project the monotonic core clock to a global timeline reference. Therefore, to compute the current *timeline* time reference, a guest application needs to access a monotonic counter (on the host), and apply the clock-discipline parameters to this monotonic clock. Additionally, the synchronization service also computes the achieved QoT. This estimated QoT enables an application to read a timestamp with its associated uncertainty.

To enable low-latency reads of the timeline reference, we need to provide low-latency access to:

1) the monotonic core clock,
2) the timeline clock-projection parameters and,
3) the estimated QoT

We solve problem (1) by utilizing the para-virtual `KVM-clock`, which provides low-latency access to the host's real-time (`CLOCK_REALTIME`) and monotonic (`CLOCK_MONOTONIC`) clocks. Of these two clocks, `CLOCK_MONOTONIC` provides a monotonic clock source, and hence can be used as a core clock. Thus, `KVM-clock` allows the host OS and the guest VMs to, in practice, share the same core clock. Therefore, timeline clock-projection parameters calculated with respect to the host core clock can be applied (using Equation 1) inside the VM as well.

To solve problems (2) and (3), we use the inter-VM shared-memory (`ivshmem`) [28] interface to memory-map a shared-memory region containing the timeline clock parameters and uncertainty information into the guest VM application's virtual-memory space. Therefore, reading a timeline-reference

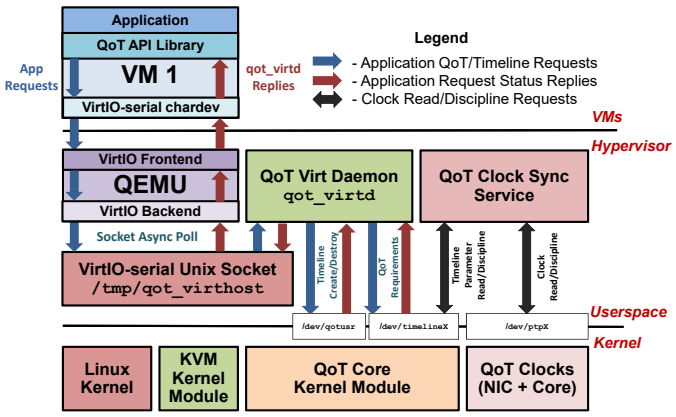


Fig. 4. Specifying QoT information from guest applications to host service `qot_virtd` using VirtIO serial

timestamp involves reading `KVM-clock` and applying the timeline-projection and uncertainty parameters from shared memory. Since, these instructions are all unprivileged, the timeline-reference time can be read with low latency.

When a VM boots up, it registers with `ivshmem_server` over the Unix socket `/tmp/ivshmem_socket` (created by `ivshmem_server`). The server replies with a read-only file descriptor to the POSIX shared-memory region `/dev/shm/ivshmem` (created by `ivshmem_server`). `ivshmem` exposes this shared-memory region as a PCI device to the guest. When a guest application binds to a timeline, it interacts with this PCI device to memory-map the shared-memory region with read-only access into its own virtual-memory space. The fact that this shared-memory space is potentially shared across multiple VMs makes it necessary that VMs have read-only access. This provides isolation between different VMs while enabling low-latency clock reads.

In our implementation, we launch the `ivshmem_server` service on the host. This service provides a guest VM the right to access a read-only shared-memory region, which contains the timeline clock parameters, and estimated uncertainty information. In addition to the guest VMs, the QoT Virtualization Service, `qot_virtd`, also memory-maps this shared-memory region with read-write access into its own virtual-memory space. Therefore, whenever the synchronization service updates the clock parameters and uncertainty information corresponding to a given timeline, `qot_virtd` writes these parameters to the shared-memory region which is memory-mapped into a guest application's virtual-memory space. Figure 5 highlights this interaction of guest VM applications with `ivshmem_server` and `qot_virtd`, and illustrates the sharing of per-timeline clock parameters and uncertainty information from host to guest (VM 1) using the memory-mapped shared-memory region created by `ivshmem_server`.

C. QoT and Full Virtualization

For guest VMs which do not support para-virtualized clocks, or hypervisors which do not permit extensions, the notion of QoT can still be supported. Our latest implementation of the QoT Stack for Linux allows all of its components: QoT core, QoT clocks, and clock-synchronization service (both

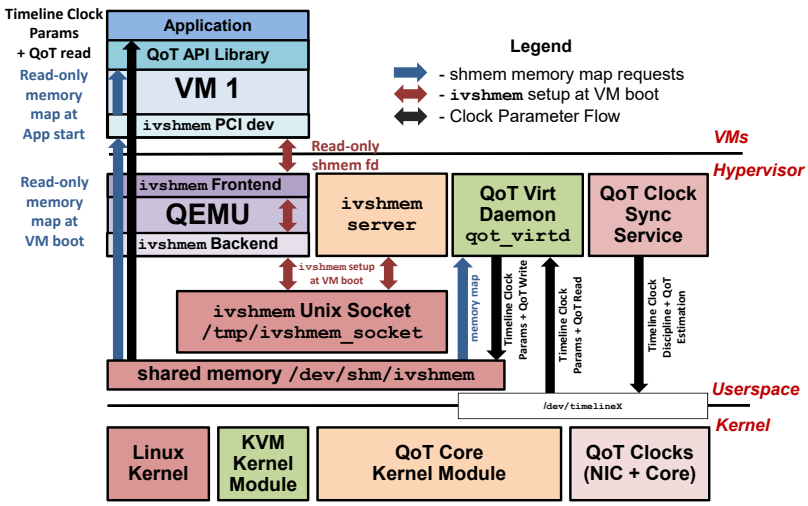


Fig. 5. Sharing clock parameters and QoT information from host service `qot_virtd` to guest VM applications using `ivshmem`

NTP-based and PTP-based with software timestamping), to run inside a VM which does not support para-virtualization. However, the overhead of emulated hardware timers (full virtualization) will cause a loss in application performance, due to higher clock-read latency. Additionally, the overhead of an emulated network stack and lack of hardware-timestamping support (for PTP) can degrade the achieved synchronization accuracy and QoT. Figure 6 illustrates the components of the QoT Stack for Linux, deployed in a QEMU-KVM Linux VM (VM 1), which does not support para-virtualized clocks.

To support a core clock based on `CLOCK_MONOTONIC`, our latest implementation of the QoT Stack for Linux implements an architecture-independent QoT core clock driver [1], which allows the entire QoT stack to be deployed on any Linux-based platform including VMs. This implementation provides a monotonic clock based on `CLOCK_MONOTONIC`, and provides time-based scheduling by using the existing Linux high-resolution timer (`HRTIMER`) interface. Our stack is open source, and the source code and installation instructions can be found at *https://bitbucket.org/rose-line/qot-stack*.

D. QoT-based Industrial Automation using QuartzV

We now describe a simple test prototype to realize the industrial-automation application described in Section III. We utilize the same structure as the described application and the main components are as follows:

1) **The Task Planner** running in a para-virtual VM with QuartzV is responsible for generating time-parametrized tasks. The VM is deployed atop QEMU-KVM on the desktop *Onyx* running Ubuntu 14.04 with a quad-core Intel i7 processor.

2) **Two Controller Nodes** each deployed on a Beaglebone Black [35] embedded platform (*Agate* and *Citrine*) with the QoT Stack for Linux, are responsible for generating and executing motion plans based on the time-based task plans.

3) **Simulated Robot Arms** receive motion plans from the controller nodes using the ROS-based [36] publish-subscribe mechanism. Since we did not have ready access to real robots, we utilize ROS Indigo [36] with the Gazebo simulator [37] to simulate two Universal Robotics UR5 [33] robot arms along

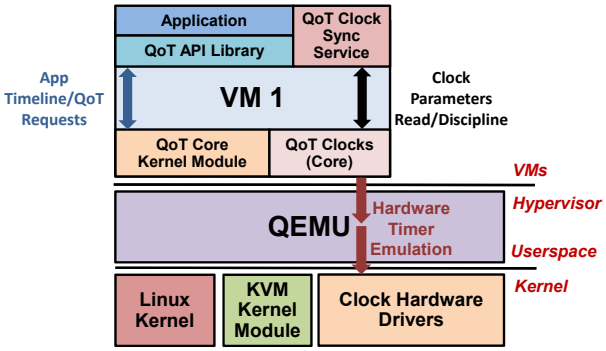


Fig. 6. QoT Stack for Linux in a fully-virtualized guest VM

with their motion controllers (using `ros-control` [38]). The simulation is performed on the desktop machine *Jasper* running Ubuntu 14.04 with a quad-core Intel i7 processor and an Nvidia GT620M GPU.

We consider a simple scenario where two robots collaboratively pick and place a component synchronously. However, our testbench can be used to develop and test more complex application scenarios. Additionally, the use of ROS enables the same application code to be deployed directly on a real robot. A video showing our prototype application can be found at *https://youtu.be/7NoxnZEWDrM*.

V. Experimental Evaluation

We now present some experimental results to benchmark the performance of QuartzV using as metrics (i) clock-synchronization accuracy, and (ii) clock-read latencies. We use the QoT Stack for Linux deployed natively as the baseline. Before stating the results, we describe our experimental setup.

A. Experimental Setup

Figure 7 illustrates the different nodes in our clock-synchronization test-bed. All the nodes are interconnected by an IEEE 1588 (PTP)-compliant Ethernet switch [40].

Our evaluations are performed on a quad-core (8 virtual cores) x86-64 Intel i7-based desktop *Onyx*, which hosts the QoT-based benchmarking applications. *Onyx* utilizes Ubuntu 14.04 with the Linux 4.4 kernel and also contains version 2.8

of the QEMU-KVM hypervisor. This enables *Onyx* to host VMs utilizing QuartzV. The Intel i7 CPU contains a constant-invariant TSC which always maintains a steady frequency, and thus can be used as a reliable clocksource. Additionally, *Onyx* is equipped with an IEEE 1588 (PTP)-compliant Intel 82574L network interface [39] which supports hardware timestamping at the PHY layer. The presence of hardware timestamping allows us to perform accurate clock synchronization using the QoT Stack for Linux's PTP-based synchronization service.

We utilize the Beaglebone Black node *Citrine* as our clock reference. The Beaglebone Black ARM-based TI AM335x chipset [35] contains an IEEE 1588-compliant network interface which supports hardware timestamping at the PHY Layer.

To measure the accuracy of clock synchronization on *Onyx*, with respect to the reference node *Citrine*, we utilize the nodes *Amethyst* and *Agate*. To measure synchronization accuracy, we need to take (near) simultaneous timestamps of a common event on both the reference (master) and the target (slave). By comparing these timestamps over a period of time, we can compute the synchronization accuracy. Therefore, we use (i) the node *Agate* (Beaglebone Black) to periodically (every second) generate UDP-multicast packets which serve as common-reference events providing timestamping opportunities, and (ii) the node *Amethyst* to generate reference timestamps (equivalent to *Citrine*) for the UDP datagrams.

Amethyst has an x86-64 Intel i7 processor, running Ubuntu 14.04 with the Linux 4.12 kernel, and is equipped with an Endace 7.5G2 DAG card [41]. The DAG card contains two ports which intercept all packets flowing between *Agate* and *Onyx*. This card also provides 7.5 nanosecond resolution timestamping [41], and processing of packets at line rate. Therefore, all the UDP packets from *Agate* can be accurately timestamped with no significant delay introduced by the DAG card. The same UDP packets can subsequently be timestamped on *Onyx* (using socket/hardware timestamping [42] on the host and guest VMs). Hence, if we assume (for now) that the DAG card on *Amethyst* can provide equivalent timestamps as the reference *Citrine*, then by comparing these timestamps with those (nearly) simultaneously generated on *Onyx*, we can compute the clock-synchronization accuracy of *Onyx* with respect to *Citrine*. Note that the introduction of the DAG card adds noise to our measurements, as there is some latency between the DAG timestamp and the timestamp on *Onyx*. However, given that *Onyx* and the DAG card share a dedicated link, the latency is low.

To accurately synchronize the DAG card on *Amethyst* to the reference *Citrine*, we utilize its inbuilt PPS (Pulse-per second) input. We use *Citrine* (Beaglebone Black) to generate a reference PPS signal over a GPIO pin (using a hardware timer), which is fed to the PPS input of the DAG card. The DAG card can use this signal along with *Amethyst's* system clock (`CLOCK_REALTIME`) to precisely synchronize its clock with <10ns accuracy. Hence, to achieve precise synchronization (using PPS), we also need to synchronize *Amethyst's* system clock (`CLOCK_REALTIME`) to the reference clock on *Citrine*, with an accuracy within 1s. This can be done using Linux

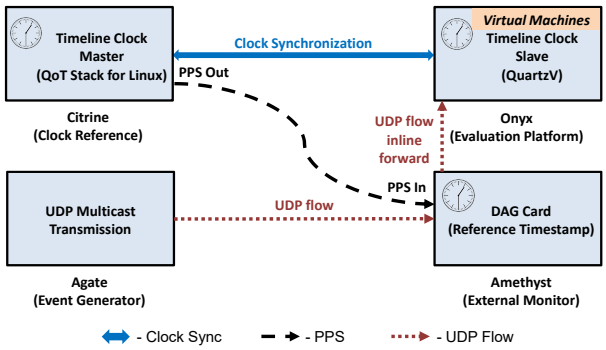


Fig. 7. QuartzV Clock-Synchronization Test-bed

PTP's [43] two-stage system-clock synchronization (*ptp4l* and *phc2sys*). *Amethyst* is also equipped with an IEEE 1588-compliant Intel 82574L network interface [39] which supports hardware timestamping at the PHY layer. Hence, using Linux PTP, we can synchronize `CLOCK_REALTIME` to the reference clock on *Citrine* to an accuracy on the order of microseconds, which is more than sufficient compared to the requirement of within 1s. Along with PPS, this allows us to achieve DAG clock synchronization with accuracy on the order of a few nanoseconds. Therefore, this setup allows us to externally measure the synchronization accuracy of QuartzV.

B. Synchronization Accuracy

We now compare the clock-synchronization accuracy (or error), with respect to the reference *Citrine*, achieved by (i) QuartzV for a Linux VM with para-virtual-clock support, (ii) the QoT Stack for Linux deployed natively, and (iii) the QoT Stack for Linux deployed in a VM with a fully-virtualized clock. Note that, in cases (i) and (ii), clock synchronization happens on the host OS, while in case (iii) clock synchronization happens inside the VM. We use Ubuntu 14.04 VMs, each configured to use 2 Virtual CPU cores and 2 GB of memory.

Figure 8 shows the histogram of the measured clock-synchronization accuracy, and Figure 9 shows a box-plot of the clock-synchronization accuracy for the mentioned scenarios. The measurements were taken over a period of six hours. Notice that the accuracy distribution achieved by the QoT Stack natively (Figure 8(a)) and QuartzV for para-virtual VMs (Figure 8(b)) is nearly identical with a mean of 24.28μs and 26.12μs respectively, and standard deviation of 5.05μs and 5.12μs respectively. This is because QuartzV performs clock synchronization on the host and transfers the clock-projection parameters to the guest VM. On the other hand, the accuracy achieved by the fully-virtualized QoT Stack inside a VM (Figure 8(c)) is lower with a mean of 70.23μs and standard deviation of 128.28μs. This is due to the additional packet-timestamping uncertainty introduced by the virtualized networking stack.

The ability to provide QoT bounds also enables fault detection. Figures 10(a) and 10(b) plot the upper and lower QoT bounds calculated by para-virtual QuartzV, and the fully-virtualized QoT Stack deployed inside a VM respectively. Observe that the computed bounds always bound the accuracy measured by the experimental test-bench. From these results,

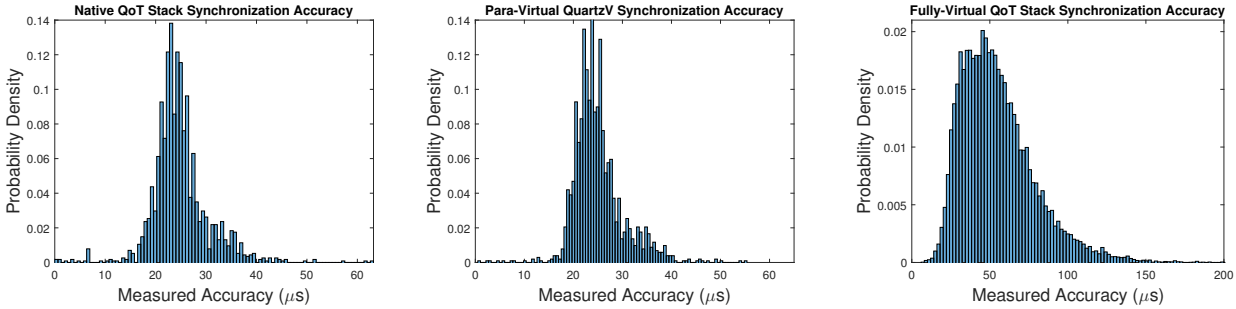


Fig. 8. Measured Clock-Synchronization Error Distributions. The y-axis represents the probability density, and the x-axis the measured error

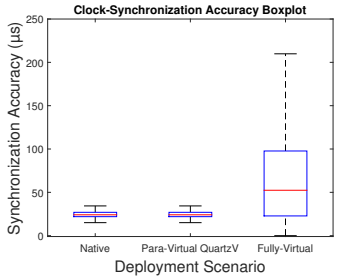


Fig. 9. Clock-Synchronization Accuracy Boxplot. The center 'red' line represents the median accuracy, the inner whiskers the 25th and 75th percentile accuracy, and the outer whiskers the minimum and maximum error observed.

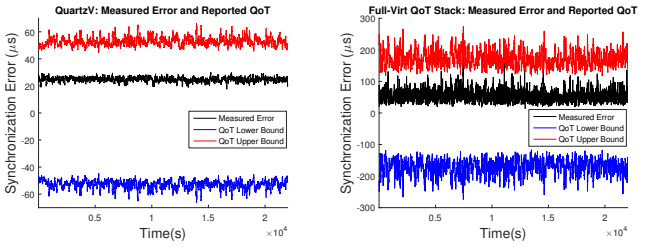


Fig. 10. QoT Bounds: (a) para-virtual QuartzV, (b) fully-virtual QoT Stack

we can conclude that QuartzV can provide near-native clock-synchronization accuracy to applications running in VMs.

C. Clock-Read Latency

To compare clock-read latencies, we consider the following scenarios: (i) the QoT Stack for Linux deployed natively, with the x86 Time-Stamp Counter (TSC) clocksource, (ii) a para-virtual Linux VM using QuartzV with the `KVM-clock` [3] clocksource, and (iii) the QoT Stack for Linux deployed in a VM with an emulated (fully-virtualized) x86 High-Precision Event Timer (HPET) clocksource. For each of these cases, we measure the latency of reading a timeline reference, which is calculated by applying the projection parameters to the QoT core clock, `QOT_CORE` (based on `CLOCK_MONOTONIC`). To measure a clock's read latency, we read the clock in a continuous loop, and take the difference between adjacent readings. For the sake of comparison, we also present latency measurements for the clocks exposed by Linux (`CLOCK_MONOTONIC` and `CLOCK_REALTIME`) along with the x86 TSC.

The clock-read latency data can be found in Table I. We present the minimum, average and standard deviation of the

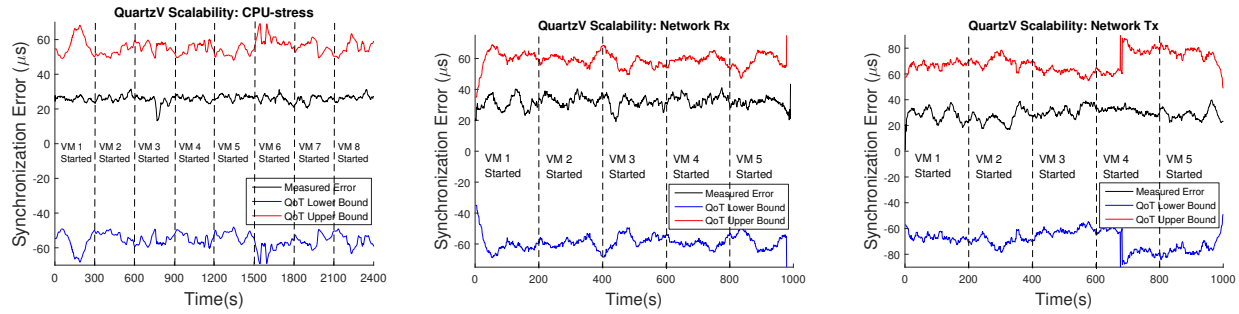
TABLE I
CLOCK-READ LATENCY (NANOSECONDS)

Scenario	Clock	Min	Average	Std. Dev
Native QoT Stack (x86 TSC)	`TSC`	4	7.41	59.55
	`REALTIME`	13	26.19	172.44
	`MONOTONIC`	13	18.41	95.74
	`QOT_CORE`	16	32.01	123.69
Para-virtual QuartzV (`KVM-clock`)	`TSC`	4	8.28	88.75
	`REALTIME`	31	40.46	246.47
	`MONOTONIC`	31	34.79	233.83
	`QOT_CORE`	54	60.71	242.34
Fully-virtual QoT Stack (Emulated HPET)	`TSC`	4	8.19	95.18
	`REALTIME`	1785	2038.02	9721.72
	`MONOTONIC`	1786	2022.13	8912.25
	`QOT_CORE`	1892	2435.64	9512.45

latency measurements for all of the clocks being compared. The data is averaged across 1000 experiments, each consisting of 1 million consecutive clock reads. Observe that, for `CLOCK_MONOTONIC`, `CLOCK_REALTIME` and `QOT_CORE`, the average and minimum clock-read latency observed in the para-virtual guest VM is roughly twice (˜2x) that observed in the native environment. This reflects the overhead introduced by using the para-virtual `KVM-clock` as a clocksource. Compare this with the fully-virtualized case which has latencies that are 3 orders of magnitude (>100x) greater than the native setting. This is due to the overhead of emulating the HPET clocksource. On the other hand, reading the TSC (using the `rdtsc` instruction) has nearly the same latency in all three scenarios. This is because `rdtsc` is an unprivileged instruction and can be executed natively [22].

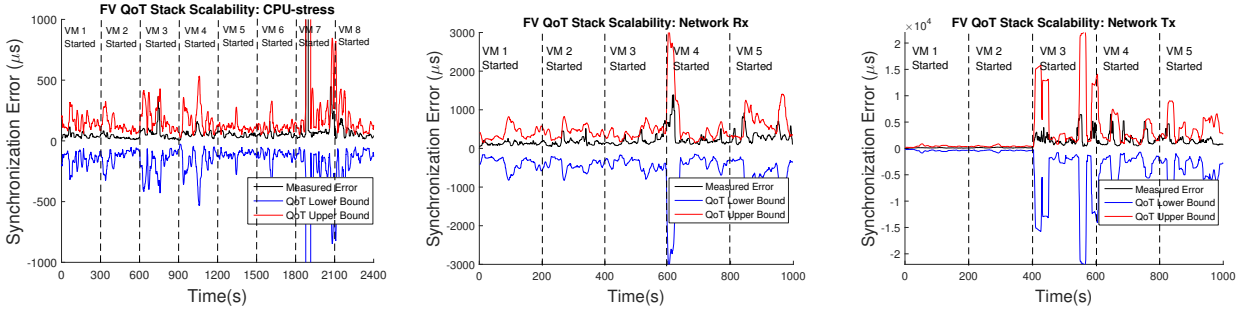
`QOT_CORE` (QoT core clock) is implemented as a wrapper around `CLOCK_MONOTONIC`. Observe that, in all the three cases, the observed latency in reading `QOT_CORE` is slightly greater than `CLOCK_MONOTONIC`. This is because of the additional overhead of applying the timeline clock-projection parameters. For the para-virtual scenario using QuartzV, the `QOT_CORE` latency is ˜1.8x that of `CLOCK_MONOTONIC`. This is due to the overhead of accessing the shared-memory region exposed by `ivshmem`. However, this overhead is minimal and does not affect the order of magnitude of the clock-read latency, as compared to `CLOCK_MONOTONIC`.

If we compare the two virtualization scenarios based on standard deviation, we can observe that reading the para-



(a) (b) (c)

Fig. 11. QuartzV Synchronization Scalability Results. The dashed lines represent moments in time where a new VM was spawned



(a) (b) (c)

Fig. 12. Fully-Virtual QoT Stack Synchronization Scalability Results. The dashed lines represent moments in time where a new VM was spawned

virtual `KVM-clock` clocksource provides approximately 40x lower standard deviation (clock-read latency variability) than an emulated clocksource. This lower variability translates to better QoT. Additionally, note that the QuartzV implementation of the `QOT_CORE` clock has similar standard deviation as `CLOCK_MONOTONIC`. Therefore, we conclude that QuartzV provides minimal loss in timing performance (latency and uncertainty) compared to the native case, while allowing services on the host to expose the notion of Quality of Time to applications running in guest VMs.

Notice that for both the para-virtual and fully-virtual scenarios, the clock-synchronization error is an order of magnitude (>10x) higher than the clock-read latency. Thus, the network residency and timestamping uncertainties are the bottlenecks for achieving good QoT for an application in a VM.

D. Clock-Synchronization Scalability

We now analyze the scalability of (i) QuartzV for Linux VMs with para-virtual clock support, and (ii) the QoT Stack for Linux deployed in a VM utilizing full virtualization. Our experiments measure the clock-synchronization accuracy achieved in the presence of competing VMs present on the same host. To test the limits of both approaches, we consider scenarios involving competing VMs with CPU and network-intensive workloads.

Figures 11 and 12 provide the scalability results for the para-virtual QuartzV setup, and the fully-virtual QoT Stack respectively. In both figures, subplots (a) provide results in the presence of competing VMs with CPU-intensive workload, subplots (b) provide results in the presence of competing VMs with network data-reception-intensive workload, and subplots (c) provide results in the presence of competing VMs with network data-transmission-intensive workload. Each plot shows the measured clock-synchronization accuracy and reported QoT bounds. The x-axis denotes the progression of time in seconds, and the y-axis indicates the measured synchronization error in microseconds. Please note that each sub-plot has a different scale for the y-axis.

Figures 11(a) and 12(a) present scalability results when multiple VMs with CPU-intensive workload are present. To test the limits of our approach, we consider a maximum of 8 VMs, each with 1 virtual core and 2 GB of memory, as our test-node *Onyx* has 8 virtual cores and 16 GB of memory. Each VM runs a simple QoT-aware application which binds to a timeline, and reads the timeline clock in a tight loop with real-time priority. In addition, each VM also utilizes the *stress* tool [44] to spawn a single CPU-intensive thread, without real-time priority. This ensures that any CPU capacity left-over by the QoT-aware application, will be consumed by the stress tool. We spawn a new VM every 300 seconds, and the dashed lines in the plot represent points in time where a new VM was spawned. In practice, we observe that our test-bed system's CPU is fully utilized with 6 CPU-intensive VMs. This is due to the use of some processing capacity by the host OS, the graphics sub-system, and QEMU-KVM.

Observe that, for the para-virtual QuartzV case (Figure 11(a)) there is no significant change in synchronization accuracy as new CPU-intensive VMs are spawned. This is because clock-synchronization is performed in the host OS, and as long as the synchronization service has sufficient resources, the

accuracy remains unaffected. Additionally, the use of hardware timestamping (available only on the host), ensures that the packet-timestamping uncertainty is unaffected by CPU load. On the other hand, for the fully-virtualized QoT Stack (Figure 12(a)), as the VM count grows higher, the synchronization accuracy degrades, and greater instability can be observed in the obtained accuracy. This is because clock synchronization is performed inside the VM, and the networking stack is emulated by the hypervisor. Thus, greater CPU load increases the uncertainty in the software timestamping of synchronization packets, and makes the synchronization service unstable. This in turn degrades accuracy. Specifically, after the addition of the 7th VM, the system is overloaded, and there are durations where the synchronization accuracy is significantly degraded (>5 times the case without overload). The QoT bounds returned by the system reflect this instability in the fully-virtual synchronization service.

Figures 11(b)(c) and 12(b)(c) present scalability results when multiple VMs with network-intensive workloads are present. In these experiments, we consider a single VM running a QoT-aware application, and a maximum of 5 competing VMs, each with 1 virtual core and 2 GB of memory, and no per-VM bandwidth restrictions. We spawn a new VM every 200 seconds, and the dashed lines in the plot represent points in time where a new VM was spawned. Each VM uses the *iperf* tool [45] to send/receive TCP packets to/from another machine on the LAN, such that the available network bandwidth is saturated. We observed that, without bandwidth regulation, a single VM is able to nearly saturate the network. Further adding new VMs causes the load to grow incrementally until the 4th VM, after which the bandwidth is fully saturated. This is because, in our setup, the 100 Mbps industrial PTP switch [40] is the network bottleneck, as compared to the 1 Gbps Ethernet card on the host *Onyx*.

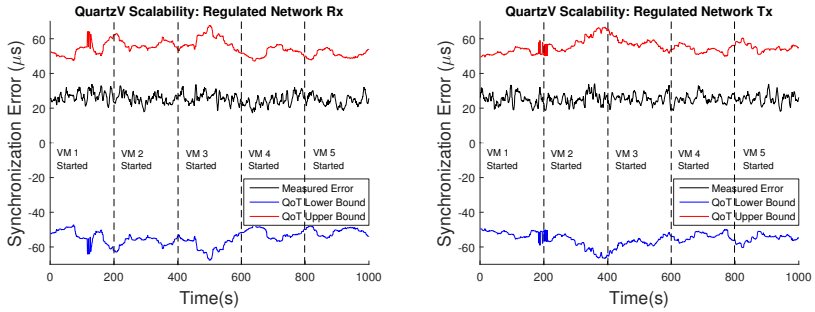
Notice that, for the para-virtual QuartzV case, with network data-reception-intensive workload (Figure 11(b)), the achieved synchronization accuracy and uncertainty (variance) degrades by ˜1.2x, as compared to the load-free scenario shown in Figure 10. However, this degradation is minimal and does not significantly change as new competing VMs are added. This is because clock synchronization is performed on the host, and uses hardware timestamping. Similarly, for the para-virtual QuartzV case with network data-transmission-intensive competing workload (Figure 11(c)), the achieved synchronization accuracy is similar to the network data-reception-intensive case. However, the synchronization uncertainty (variance) degradation is higher by ˜1.3x, as compared to the previous case. This observation especially holds true when more competing VMs are present (> 3), and is reflected by the increase in the reported QoT bounds.

On the other hand, for the fully-virtualized QoT Stack (Figure 12(b)(c)), as the competing network-intensive VMs increase, on average the synchronization accuracy degrades significantly (˜1.8x-4x in different regions). Moreover, at the instances where new VMs are added, greater instability can be observed in the obtained accuracy. Also, observe that, for the network data-reception-intensive case, the accuracy significantly degrades on the addition of the 4th VM, and for the data-transmission intensive case, this can be observed at the point of addition of the 3rd VM. The accuracy degradation is one order-of-magnitude worse for the network data-transmission-intensive case, and this is reflected by the QoT bounds returned by the system, which are, in the worst-case, about ˜10x of those reported in the presence of network data-reception-intensive load.

For both the para-virtual and fully-virtual scenarios, the accuracy degradation observed is greater in the presence of data-transmission-intensive network load. This is because, for the network-reception case, as the incoming traffic increases, there is more congestion at the PTP switch, as the switch is the bottleneck. On the other hand, for the network-transmission case, as the switch becomes congested, packets start getting dropped, and there are more re-transmission attempts made at the host Ethernet card (due to TCP), thus causing greater congestion at the host. However, the degradation of both the measured accuracy and computed QoT bounds observed while using para-virtual QuartzV is minimal, as compared to the significant degradation observed while using the fully-virtualized QoT Stack inside a VM. This is explained by the fact that, during overload, the overhead of using an emulated networking stack creates greater uncertainties and delays in handling and timestamping synchronization packets.

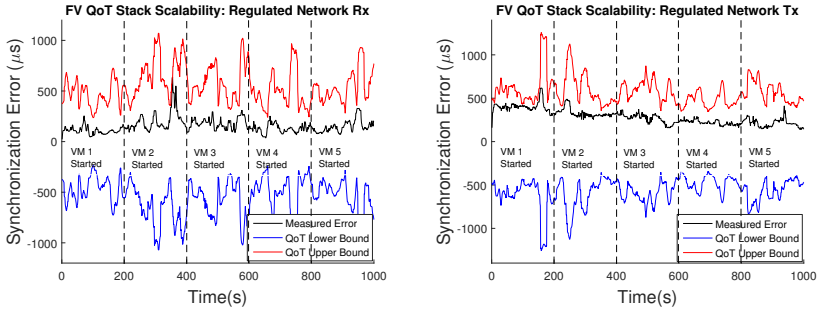
In summary, our scalability experiments indicate that, for the para-virtual QuartzV approach, CPU-intensive VMs do not significantly affect clock-synchronization accuracy when: (i) adequate hard CPU reservations are used (already guaranteed by default in all hypervisors), (ii) Virtual Machine over-commit is avoided (i.e., not allowing more VMs than available resources), and (iii) by ensuring that the clock-synchronization service has sufficient resources. However, the same cannot be said for the fully-virtualized QoT Stack deployed inside a VM. For the network-intensive scalability experiments, we have observed that, for both QuartzV and the fully-virtual QoT Stack, heavy network load does affect the clock-synchronization accuracy and the reported QoT bounds. The degradation in the observed accuracy is significant for the fully-virtual QoT Stack while being minor for the para-virtual QuartzV approach. This degradation is caused due to added uncertainty in network timestamping and packet residency delays, and can be avoided by restricting the network bandwidth available to a VM, based on a user-specified limit. Such functionality is available in most hypervisors including QEMU-KVM.

Figures 13 and 14 present scalability results in the presence of bandwidth-restricted network-intensive VMs, for para-virtual QuartzV and the fully-virtual QoT Stack respectively. We consider a maximum of 5 competing VMs, each of which has its transmission and reception bandwidth restricted to 2 MB/s (16 Mbps). In both figures, subplots (a) provide results in the presence of competing VMs with network data-reception-intensive workload, and subplots (b) provide results in the presence of competing VMs with network data-transmission-intensive workload. For both the para-virtual QuartzV scenario



(a) (b)

Fig. 13. QuartzV Synchronization Scalability Results with per-VM Network Reception/Transmission Bandwidth restricted to 2 MB/s



(a) (b)

Fig. 14. Fully-Virtual QoT Stack Synchronization Scalability Results with per-VM Network Reception/Transmission Bandwidth restricted to 2MB/s

and the fully-virtual QoT Stack, the plots indicate that restricting the bandwidth of competing VMs can prevent significant degradation of synchronization accuracy, as compared to the scenario with no bandwidth restrictions.

E. Clock-Read Scalability

Figure 15 plots the average clock-read latency of the para-virtual QuartzV approach with multiple VMs continuously performing simultaneous clock reads. Observe that for both `QOT_CORE` and `CLOCK_MONOTONIC`, the clock-read latency increases slightly for each new VM spawned. This is due to the unavoidable contention in reading the hardware counter to compute the time. Thus, as `qot_virtd` writes the clock-discipline parameters to a shared-memory region which all VMs can simultaneously read from, there is no bottleneck in our implementation, which allows QuartzV to easily scale and support multiple VMs.

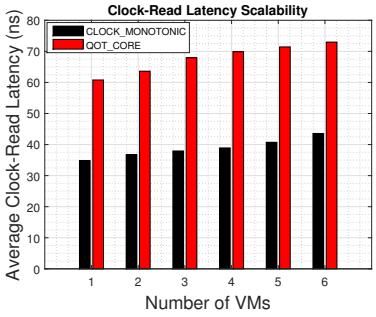


Fig. 15. QuartzV Clock-Read Scalability Results

VI. Conclusion and Future Work

The emergence of geo-distributed coordinated CPS makes it essential to utilize a shared notion of time along with QoT to enable fault-tolerant scalable coordination. The notion of QoT enables the calculation of timestamp uncertainty bounds with respect to a clock reference. If these supplied uncertainty bounds exceed an application-specified acceptable limit, the application can enter a graceful-degradation mode, and thus be fault-tolerant in the face of degraded QoT.

Given that virtualization is increasingly utilized in cyber-physical applications, we introduced the QuartzV extension to the QoT Stack for Linux to make VMs QoT-aware. QuartzV harnesses para-virtual clocks along with the dependent-clock paradigm [16] to provide near-native timing performance in VMs. We also demonstrated the utility of QuartzV by using it in a prototype industrial-automation application. This, in turn, illustrates that QoT-awareness makes it possible for intelligent CPS applications to dynamically take coordination decisions, based on a shared notion of time and the delivered QoT.

For VMs which do not support para-virtual clocks, or hypervisors which do not permit extensions, we extended the QoT Stack for Linux so that it can be entirely deployed in a VM. However, our experiments indicate that QuartzV's para-virtual implementation can achieve much higher synchronization accuracy, better scalability and timing performance.

In future work, we will extend our QoT Stack for Linux to support OS-level virtualization technologies (containerization). Additionally, we will focus on making our timeline-based coordination protocol scale across heterogeneous networking technologies and synchronization domains.

ACKNOWLEDGEMENTS

The authors would like to thank Fatima Anwar, Andrew Symington, Adwait Dongare, Anthony Rowe and Mani Srivastava for their efforts on the QoT Stack for Linux. This research is funded in part by the National Science Foundation under award CNS-1329644. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, or the U.S. Government.

REFERENCES

[1] F. Anwar, S. D'souza, A. Symington, A. Dongare, R. Rajkumar, A. Rowe and M. Srivastava, "Timeline: An Operating System Abstraction for Time-Aware Applications", in Proc. of *IEEE Real-Time Systems Symposium*, 2016

[2] S. D'souza and R. Rajkumar, "Time-based Coordination in Geo-Distributed Cyber-Physical Systems", in Proc. of *USENIX Workshop on Hot Topics of Cloud Computing*, 2017

[3] "Kernel-based Virtual Machine", http://www.linux-kvm.org/

[4] J. Enright and P. Wurman, "Optimization and Coordinated Autonomy in Mobile Fulfillment Systems", In Proc. of *AAAI Workshop*, 2011

[5] SAE J2735 Standard, *https://ntl.bts.gov/lib/51000/51100/51167/DE156ECC.pdf*

[6] M. Buevich, X. Zhang, D. Schnitzer, T. Escalada, A. Jacquiau-Chamski, J. Thacker and A. Rowe, "Microgrid Losses: When the Whole is Greater Than the Sum of Its Parts", In Proc. of *7th IEEE/ACM International Conference on on Cyber-Physical Systems*, 2016

[7] M. Satyanarayanan et al., "The Case for VM-Based Cloudlets in Mobile Computing", in *IEEE Pervasive Computing*, Vol. 8, Issue: 4, 2009

[8] B. Liskov, "Practical Uses of Synchronized Clocks in Distributed Systems", in Proc. of *ACM symposium on Principles of distributed computing*, 1991

[9] A. Rowe, R. Mangharam and R. Rajkumar, "FireFly: A Time Synchronized Real-Time Sensor Networking Platform", *http://nanork.org/attachments/148/nrk-chapter06.pdf*

[10] B. Regula, "Formation control of a large group of UAVs with safe path planning", in Proc. of *IEEE Mediterranean Conference on Control and Automation*, 2013.

[11] S. Natarajan and A. Ganz, "SURGNET: An Integrated Surgical Data Transmission System for Telesurgery", in *International Journal of Telemedicine and Applications*, Article ID 435849, 2009

[12] J. C. Corbett et al., "Spanner: Google's globally distributed database", in *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, 2013

[13] M. Lipinski, T. Wostowski, J. Serrano, and P. Alvarez, "White rabbit: A PTP application for robust sub-nanosecond synchronization", in Proc. of *Intl. IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication (ISPCS)*, 2011

[14] D. L. Mills, "Internet Time Synchronization: The Network Time Protocol," in *IEEE Transactions on Communication*, vol. 39, no. 10, 1991.

[15] K. Lee, J. C. Eidson, H. Weibel, and D. Mohl, "IEEE 1588-standard for a precision clock synchronization protocol for networked measurement and control systems", in *IEEE Instrumentation and Measurement Society Standard*, 2005

[16] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch, "Virtualize everything but time", in Proc. of *OSDI*, 2010

[17] J. Smith and R. Nair, "The Architecture of Virtual Machines", in *IEEE Transactions on Computers*, vol. 38-5, 2005

[18] Docker Containerization Platform, *https://www.docker.com/*

[19] H. Kim, S. Wang, and R. Rajkumar, "Responsive and Enforced Interrupt Handling for Real-Time System Virtualization", in Proc. of *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2015.

[20] P. Barham et. al, "Xen and the Art of Virtualization", in Proc. of *SOSP*, 2003

[21] The Xen Project, *https://www.xenproject.org/*

[22] Intel VT, *https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html*

[23] AMD Virtualization Solutions for Data Centers, *http://www.amd.com/en-gb/solutions/servers/virtualization*

[24] J. Ridoux and D. Veitch, "TSCclock Goes Live. A demonstration of a robust, accurate replacement to ntpd", in Proc. of *ACM SIGCOMM*, 2008

[25] KVM Paravirtual Clock, http://www.linux-kvm.org/page/KVMClock

[26] Intel 64 and IA-32 Architectures Software Developer's Manual, *https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf*

[27] VirtIO-Serial, *https://fedoraproject.org/wiki/Features/VirtioSerial*

[28] C. Macdonell, "Inter-VM shared memory PCI device", *https://lwn.net/Articles/380869/*

[29] J. Gertler, "Analytical Redundancy Methods in Fault Detection and Isolation", in *Preprints of IFAC/IMACS Symposium on Fault Detection, Supervision and Safety for Technical Processes*, 1991

[30] A. Agostini, C. Torras, and F. Worgotter, "Integrating Task Planning and Interactive Learning for Robots to Work in Human Environments", in Proc. of International Joint Conference on Artificial Intelligence, 2011

[31] Y. Li, J. Sun, J. Dong, Y. Liu and J. Sun, "Planning as Model Checking Tasks", in Proc. of the *35th Annual IEEE Software Engineering Workshop*, 2012

[32] T. Schouwenaars, E. Feron, and J. How, "Safe Receding Horizon Path Planning for Autonomous Vehicles", in Proc. of the *Annual Allerton Conference on Communication Control and Computing*, 2002.

[33] Universal Robotics UR-5, *https://www.universal-robots.com/*

[34] Kuka Robotics, *https://www.kuka.com*

[35] Beaglebone Black, *https://beagleboard.org/black*

[36] ROS Indigo Igloo, *http://wiki.ros.org/indigo*

[37] Gazebo Robot Simulation, *http://gazebosim.org/*

[38] ROS Control, *http://wiki.ros.org/ros_control*

[39] Intel Gigabit CT Adapter, *https://www.intel.com/content/www/us/en/products/network-io/ethernet/gigabit-adapters/ct-desktop.html*

[40] Moxa EtherDevice Switch EDS-405A-PTP, *https://www.moxa.com/product/EDS-405A-PTP.htm*

[41] Endace DAG 7.5G2, *https://www.endace.com/dag-7.5g2-datasheet.pdf*

[42] Linux Kernel Packet Timestamping, *https://www.kernel.org/doc/Documentation/networking/timestamping.txt*

[43] The Linux PTP Project, *http://linuxptp.sourceforge.net/*

[44] Ubuntu stress tool, *http://manpages.ubuntu.com/manpages/xenial/man1/stress.1.html*

[45] iperf: The TCP, UDP and SCTP network measurement tool, *https://iperf.fr/*