Virtuoso: High Resource Utilization and μs-scale Performance Isolation in a Shared Virtual Machine TCP Network Stack

Matheus Stolet
Max Planck Institute for Software Systems

Liam Arzola
Max Planck Institute for Software Systems, Cornell University

Simon Peter
University of Washington

Antoine Kaufmann
Max Planck Institute for Software Systems

Abstract

Virtualization improves resource efficiency and ensures security and performance isolation for cloud applications. To that end, operators today use a layered architecture that runs a separate network stack instance in each VM and container connected to a separate virtual switch. Decoupling through layering reduces complexity, but induces performance and resource overheads that are at odds with increasing demands for network bandwidth, communication requirements for large distributed applications, and low latency.

We present Virtuoso, a new software networking stack for VMs and containers. Virtuoso performs a fundamental re-organization of the networking stack to maximize CPU utilization, enforce isolation, and minimize networking stack overheads. We maximize utilization by running one elastically shared network stack instance on dedicated cores; we enforce isolation by performing central and fine-grained per-packet resource accounting and scheduling; we reduce overheads by building a single-layer data path with a one-shot fast-path incorporating all processing from the TCP transport layer through network virtualization and virtual switching. Virtuoso improves resource utilization by up to 50%, latencies by up to 42% compared to other virtualized network stacks without sacrificing isolation, and keeps processing overhead within 11.5% of unvirtualized network stacks.

1 Introduction

The cloud leverages virtualization to improve resource utilization while ensuring isolation for security and performance. Guests running as virtual machines and containers enable deployment of separate applications from multiple tenants onto shared physical hosts. The hypervisor and operating system allocate and manage the shared physical resources such as processor cores, memory, and the network link. For network communication, each VM and container runs a separate network stack instance that sends and receives packets through a central virtual switch. VMs send packets from their OS stack through virtual NICs to the hypervisor, while containers run in isolated network name spaces also generating raw guest network packets, then forwarded through a virtual interface to a kernel or userspace virtual bridge or switch. The operator configures the virtual switch to implement network virtualization features, such as tunneling, bandwidth limits, and security checks, and pass packets to and from the physical network.

This results in a layered architecture where packets pass through a series of different separate loosely-coupled components, such as the guest transport layer, network layer, virtual NIC, virtual switch, etc. This layered architecture has worked well but incurs significant performance and resource overheads. On one hand, decoupling through layering simplifies development, configuration, and management. Decomposition into separate layers also allows for a certain level of performance isolation, as guest stacks may be isolated by dedicating a number of CPU cores to each guest. On the other hand, demands for increasing network bandwidths and for low latency communication are expensive to meet in this architecture. 100 Gbps links are commonplace and 400 Gbps are already available. At the same time, modern cloud applications demand μs-scale network latencies [51]. Coupled with the slow down of Moore's Law, any wasted CPU cycles due to network processing—incurred either due to underutilized dedicated CPU resources or inefficiencies in network stack processing—are particularly problematic.

In this paper, we argue that the existing layered virtual network stack architecture unnecessarily sacrifices resource efficiency and performance, in particular for isolation. The typical static CPU allocation for guests (VMs or containers) requires users to provision cores for peak traffic. However, guests do not serve peak traffic at all times, leading to poor CPU utilization because of idle capacity allocated for network processing in non-peak times. Additionally, the layered architecture providing network virtualization and isolation on packet streams from per-guest network stacks add significant overhead to the networking datapath. The hypervisor individually mediates every packet sent or received by the

guest, increasing CPU overhead and communication latency. Up to 60% of the time spent transmitting a packet can be from transferring packets from the guest virtual interface to the physical interface [26, 27].

We argue that these overheads are not inherent to network virtualization, but are an artifact of the existing architecture. To that end, we propose a fundamental re-organization of the full virtual network stack architecture. We present Virtuoso, a new, shared software networking stack for virtual machines and containers that maximizes CPU utilization, while minimizing network stack processing overheads and enforcing isolation. Virtuoso provides drop-in compatibility for sockets and the TCP protocol. Our evaluation shows that Virtuoso can improve resource utilization by up to 50% while increasing throughput by up to 51% over optimized layered stacks, while still ensuring performance isolation at μs-scale tail latencies. Virtuoso also achieves high absolute throughput, incurring only a 14% throughput penalty compared to state-of-the-art bare-metal network stacks.

The first Virtuoso key idea is to use only *one network stack instance* in the hypervisor, *shared* by all guests. Sharing improves CPU utilization for bursty workloads, as the shared stack elastically allocates CPU resources just-in-time, rather than statically provisioning CPU bandwidth for each guest's peak. To provide microsecond-scale performance isolation in a shared network stack, we leverage *fine-grained per-packet resource scheduling*. Virtuoso accounts CPU cycles and network bandwidth spent by each processed packet to the respective guest resource budget, scheduling each guest on a per-packet basis. These fine-grained mechanisms incur minimal performance overhead but enable performance isolation even for microsecond-scale latencies. Finally, a *coalesced data path* combines all virtual network processing from transport down to virtual switching. The coalesced data path in Virtuoso collapses all layers in the stack, minimizing processing by avoiding overheads for intermediate queuing, implementing the same functionality as conventional layered stacks considerably faster and with fewer processor cycles. We further split the data path into a *fast-* and a *slow-path*. Virtuoso processes common packets of established connections on the fast-path in *one-shot*, further reducing necessary state and simplifying performance isolation through short, predictable code paths. Uncommon cases are handled on the slow-path at a small performance penalty.

Our contributions are the following:

- The design of a new shared TCP network stack for virtual environments that improves resource utilization and leverages fine-grained scheduling for isolation.
- One-shot network virtualization fast-path incurring minimal virtualization overhead.
- Virtuoso prototype implementation for Linux and QEMU.
- Performance analysis of Virtuoso prototype to quantify the resource utilization improvement and overhead reduction, and confirming performance isolation and low tail latency.

We will release Virtuoso as open-source software.

2 Background

Network communication faces unique challenges in virtualized environmets. In this section, we first discuss the conceptual requirements and goals, then move on to the conventional implementation and its shortcomings, and finally to prior approaches addressing a subset of these shortcomings.

2.1 Network Virtualization Concepts

Virtualization aims to facilitate management and consolidation of host and network resources. Multiple guest VMs or containers with separate network addresses share a single physical host, network controller, and link. Similarly, multiple separate virtual networks share the same physical network. Multiple separate tenants can manage containers, VMs, and virtual networks, while yet a separate operator maintains the shared physical infrastructure. Tenants expect to flexibly configure and use their virtual infrastructure, e.g. using custom addressing, routing, and protocols. Infrastructure operators must multiplex the physical resources providing isolation to produce the illusion of completely separate infrastructure to mutually non-trusting tenants.

On the hosts this requires virtual switching, moving packets between the various guests and the shared physical network in a controlled and safe way, and enforcing all necessary processing for security and isolation. In the network, virtualization requires tunneling protocols such as VXLAN, [42], NVGRE [12], and GENEVE [14] to encapsulate packets, enabling use of separate protocols and routing on the physical network. Virtual switching also assigns appropriate physical network addresses based on virtual network addresses.

Operator Goals. The complete network communication infrastructure aims to meet the following operator goals:

- **Performance isolation.** Guests must not use more than their allocated resources, and must not affect throughput or latency of other guests.
- **High resource utilization.** Resources are expensive, so network virtualization must avoid idle resources. This also implies that host infrastructure must scale to many guests per physical server to utilize all available resources.
- **Security.** Tenants must not be able to see or tamper with other tenant's packets. Similarly the physical network must be protected from tampering by tenants.

Tenant Goals. In addition, there are two tenant goals:

- **Low latency.** Modern applications require low common-case and tail latency communication on the order of μs.
- **High throughput.** Network virtualization must achieve high throughput and keep up with rising network speeds, on the order of hundreds of Gbps.

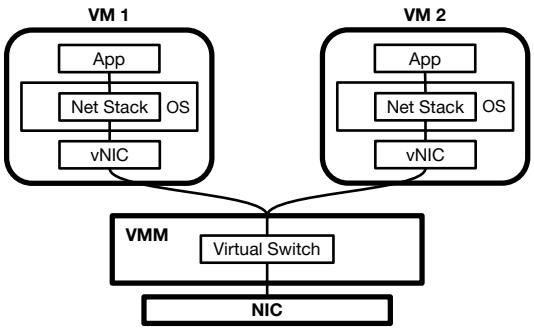

Fig. 1: Layered and independent virtualized network stacks.

2.2 Status Quo: Layered Silos

Traditionally, network virtualization is implemented as a deeply layered architecture (Fig. 1). Packet processing is divided between independent network stacks in each guest (managed by the tenant), multiplexed by the virtual switch running as the lowest layer on the host (managed by the operator). Guests send and receive raw network packets through their conventional OS network stack via the assigned virtual NIC (vNIC) just as in a native deployment. Containers run in separate network namespaces typically communicating with the outside through `veth` pairs [22], while VMs run separate OS instances and use virtual NICs such as `virtio-net` [45] implemented by the hypervisor. The virtual switch (vSwitch) takes packets sent on the guests' vNICs, routes and encapsulates them for the physical network, and then sends them out through the host's physical NIC. Receiving packets works symmetrically: packets arrive on the physical NIC, the vSwitch inspects and decapsulates them to determine the virtual network, and then looks up and passes them to the corresponding vNIC.

This architecture largely evolved organically. Guests completely re-use the existing network stack originally built for physical systems. vNICs optimized for virtualization replaced emulated physical NICs to improve performance, but otherwise the stack remains unchanged.

Advantages. Separately developed, maintained, and operated virtual switches simplify implementation and enable flexible deployment with different implementations. Typically data centers run guest VMs and containers on dedicated cores. Thus, independent per-guest stacks effectively silo applications, minimizing inter-VM performance interference. Independent stacks can easily account for the CPU time and network bandwidth used by each VM. Much of the protocol processing happens in the guest and is thus automatically accounted for and isolated. Early demultiplexing also prevents priority inversion for more complex scheduling policies, as packet priorities are only known after demultiplexing [29].

Disadvantages. On the other hand, independent network stacks often over-provision resources. For the typically bursty workloads, tenants have to provision VMs with enough resources (especially cores) for peak bandwidth. While VMs can share resources such as CPU cores, this is not compatible with μs-scale latency requirements because of long and expensive VM context switches. Cloud VMs, in particular, typically exclusively provision processor cores, with only exceptions for the smallest and cheapest instance types. VMs typically only rarely operate at peak traffic, frequently leaving resources underutilized.

Layered network stacks also incur overheads increasing latency and wasting precious CPU cycles [20, 34]. Each layer adds indirection, often through queues or other data transfer mechanisms. For example, the TCP layer might generate segments that then queue up lower in the stack because of vSwitch-enforced bandwidth limits. Further, independent layers often repeatedly look up similar information for packets [29] in multiple layers in different data structures, e.g. guest IP routing, ARP lookup, vSwitch MAC to physical IP translation, physical IP to physical MAC.

Finally, even layered stacks include significant processing before multiplexing points that is not performance isolated, e.g. in the vSwitch. This is a source of performance cross-talk between guests and tail latency [44].

Summary. Layered stacks face two key challenges:

First, there is a *trade-off between isolation and resource utilization*. Multiplexing resources early with independent stacks facilitates isolation but fails to capitalize on finer-grained resource-sharing opportunities because resources are siloed from the start and cannot be pooled at lower levels.

Second, *layering provides modularity but leads to overheads* in packet processing. These overheads are exacerbated by rising network speeds, microsecond tail-latency requirements [4], and the large scale of datacenter applications.

2.3 Prior Work

Prior work has investigated these challenges, but fails to satisfy all goals. In particular, providing high resource utilization with minimal interference for μs-scale workloads remains a challenge.

Reducing layering overheads. A range of work seeks to avoid layering overheads and reduce indirection in packet processing in specific layers. These solutions span kernel bypass [3, 7, 18, 34, 51], kernel offload via eBPF [13, 37, 52], zero-copy methods [3, 21, 24, 34], new NIC interfaces [8, 35, 39, 47], and one-shot unlayered fast-paths [20, 40]. These approaches do not completely solve performance overheads or isolation across the virtualized stack.

Hardware offload. Offloading different parts of network virtualization processing can significantly reduce overheads. Recent data center NICs support offload for VXLAN, GENEVE, and NVGRE [31] en-/de-capsulation. Complete offload

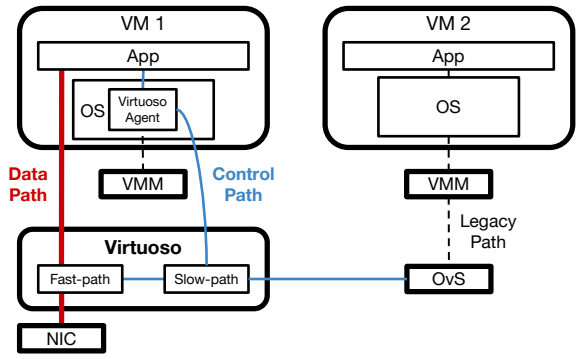

Fig. 2: TX and RX operations go through the fast-path and control operatons go through the slow-path. Legacy applications go through the layered legacy path.

of virtual switching fast-paths can reduce overheads [2, 10, 25, 32] further and enable direct HW NIC access for guests through SR-IOV [2, 10]. However, these approaches either rely on fixed hardware, limited to specific protocols and features [10], or themselves leverage multi-core SmartNICs [2,25,32], resulting in another layered architecture with the challenges discussed above. Hardware offload also gives rise to other performance isolation challenges with shared hardware resources, such as the NIC, PCIe interconnect, and IOMMU [1]. Finally, software solutions are relevant because of the comparative flexibility and simple deployability.

3 Virtuoso Approach

Virtuoso (Fig. 2) eliminates the tradeoff between resource efficiency and isolation by sharing a network stack among guests and implementing isolation in a single layer. The shared stack allows Virtuoso to elastically pool resources and increase utilization, while fine-grained resource accounting and scheduling ensure performance isolation. Virtuoso uses a multi-threaded data fast-path with dedicated cores for common case send and receive operations, and a separate slow-path for data path exceptions and control operations. The fast-path combines all network virtualization and packet processing layers up to and including the TCP transport layer, minimizing the path between the guest application and the host NIC. The fast-path implements en-/de-capsulation and de-multiplexing, and combines all common-case processing. Only the sockets interface remains in the guest, but is tightly integrated with guest applications in guest userspace through a dynamic link library.

3.1 Design Principles

To achieve our goals, we employ the following principles:

Shared network stack for elastic resource utilization. Instead of partitioning network processing to multiple guest silos and the hypervisor, Virtuoso places one shared network stack instance in the hypervisor. Externalizing network processing allows guests to serve the same workload with fewer cores; we instead re-allocate some of these cores for the shared stack. This resource consolidation particularly improves utilization for bursty workloads by being elastic; the larger shared pool of cores can absorb bursts better than multiple static per-guest pools [49].

Fine-grained per-packet scheduling for isolation. Instead of coarse-grained resource management via cores dedicated to guests, we employ central and fine-grained resource accounting and scheduling for each individual packet to ensure isolation in the shared network stack. Virtuoso precisely accounts processor cycles and network bandwidth spent for each processed packet to the respective guest resource budget. Virtuoso leverages global visibility across all guests combined with accurate resource accounting to implement fine-grained per-packet scheduling that enforces tight isolation policies. Scheduling is implemented centrally at a single layer in the system, minimizing crosstalk [23, 44]. This nimble mechanism incurs minimal performance overhead but enables performance isolation even for microsecond-scale latencies.

Single-layer data path. Instead of layered processing, Virtuoso leverages a single-layer data path, coalescing all network processing from the TCP transport layer down to network virtualization and virtual switching, for receive and transmit. Guest applications interact directly with the data path through efficient shared memory queues, by linking a dynamic link library in guest userspace that provides the TCP sockets API. This allows Virtuoso to implement the same functionality as conventional layered stacks considerably faster and with fewer processor cycles by communicating directly with the single-layer data path.

One-shot fast-path. We further streamline the Virtuoso data path via a one-shot fast-path. For each TCP connection, *one-shot processing* pre-computes rarely changing processing state, such as guest and physical IP routing and tunnel state, storing it in the fast-path, reducing per-packet processing overhead for common packets of established connections. Handling a limited number of common cases in the fast-path also simplifies performance isolation through short and predictable code paths. For example, when sending a TCP segment from the guest on a virtual TCP connection, the fast-path can directly create a physical packet with all relevant virtualization headers and send it via the host NIC in a short operation. Uncommon cases are handled on a separate slow-path at a small performance penalty.

4 Detailed Virtuoso Design

In the Virtuoso network stack, a multi-core fast-path polls guests for new packets and parses and generates headers in a single layer for low-overhead packet transmission and

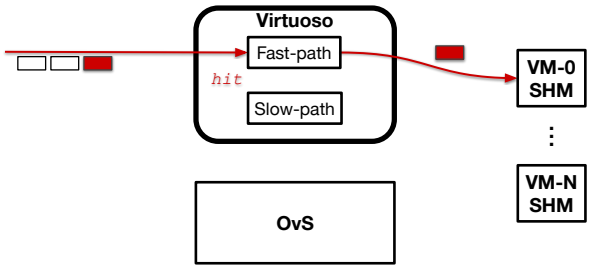

Fig. 3: Virtuoso uses cached OvS state in the fast-path to steer packets to the proper VM's payload buffer.

reception. The fast-path accesses each guest library's shared memory region and aggregates packets from multiple guests into a batch to increase utilization. A separate slow-path core handles control operations and exceptions. Dividing tasks between a fast-path and a slow-path allows us to reduce overheads by streamlining the fast-path. For initialization of these shared memory channels between applications and Virtuoso during startup, we leverage a modified hypervisor for guests and the host OS for containers. We describe these implementations later (§5.3, §5.2). After shared memory regions are set up, there are no differences for the application-Virtuoso data path between virtual machines and containers. Hence, we refer to them collectively as *guests*.

In this section, we give detailed descriptions of the different components of Virtuoso. We start by describing how we integrate efficient network virtualization in a multi-core fast-path (§4.1). We then describe how we track each guest's resource usage on individual fast-path cores through a per-guest budget (§4.2), followed by a discussion of how we use this information to coordinate guest resource budget allocation across fast-path cores centrally through our slow-path (§4.3). We then describe how we build fine-grained scheduling based on the per-guest resource budgets (§4.4). Finally, we describe how we protect the Virtuoso network stack on the host when sharing it among guests (§4.5).

4.1 One-shot, Single-layer Transport

Single-layer transport. To improve performance and minimize overheads, Virtuoso combines all processing from the TCP transport layer all the way down to network virtualization into a streamlined one-shot fast-path (Fig. 3), for send and receive. Separating out minimal common case processing in a minimal fast-path enables performance optimization, while a separate slow-path ensures that less frequent cases can still be handled. Regular data transfer packets for established TCP connections exclusively use this optimized data path, while packets for unknown connections or other protocols pass through the slow-path. In the case of new connection requests, the slow-path then sets up one-shot fast-path state for the connection so future packets remain on the fast-path (Fig. 4).

On receive, the fast-path parses the packet according to the expected format, configured by default to TCP over IPv4 on the guest side, encapsulated in GRE [9] over UDP, IPv4, and Ethernet on the physical network. The fast-path then leverages the corresponding connection identifiers, TCP ports, guest IPs, and tunnel ID to look up the consolidated flow state. After validating the packet against the state, the fast-path directly stores the TCP payload in the guest flow buffer and enqueues a notification in the corresponding guest receive queue. Finally, if necessary, we reformat the packet by swapping addresses and tweaking the TCP header into a response TCP acknowledgement.

Similarly for transmit, once Virtuoso schedules a flow to transmit a packet, the flow state directly provides all necessary state to directly assemble the complete packet with all headers for immediate transmission via the NIC. Headers are divided between inner and outer headers. The inner TCP and IP headers includes the source and destination IP address and port on the guest (virtual) network. The outer headers include the GRE encapsulation with the key field to identify the network [6] wrapped by the outer UDP and IP header and corresponding physical network source and destination IP addresses and ports, finally wrapped by Ethernet and the necessary peer MAC address. Virtuoso stores these key fields in the consolidated flow state.

One-shot fast-path processing. We implement this processing as straight-line code with minimal control flow (other than exceptions for rare cases) and no packet modifications until acknowledgements [20]. Virtuoso processes packet in one shot without intermediate queuing or access to complex data structures other than the consolidated flow state. We skip some steps for conventional network virtualization such decapsulation completely. Other steps we combine with already necessary related steps previously in other layers, such as combining the virtual switching table lookup with the TCP flow state lookup.

Our state consolidation optimizations rely on most of this state, such as guest routing, tunneling, host addressing and routing, remaining typically unchanged over the life of a connection. Thus it can be pre-computed and stored with all other necessary state when a new connection is established. This is related to other fast-path caches for virtual switching state in systems such as Open vSwitch [33]. Except Virtuoso explicitly and eagerly manages this state, adding it, updating it, and removing it as necessary instead of relying on misses and invalidations. This also implies that changes to this state are more expensive in Virtuoso than in other systems, as many changes to individual connection state instances on the fast-path may be required for an individual change to the underlying state.

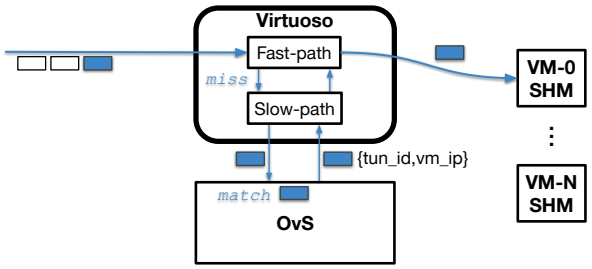

Fig. 4: Virtuoso retrieves tunneling information from OvS when there is a miss.

Slow-path for remaining processing. The Virtuoso slow-path handles all packets not handled by the fast-path. This includes packets that are not TCP data packets, TCP control packets to open and close connections as well as non-TCP packets. For the network virtualization slow-path and control plane we leverage the existing Open vSwitch [33]. First off, Virtuoso leverages OvS as a virtual switch for slow-path packets, by first passing them, truncated to the headers, to OvS. OvS then asynchronously sends back the packet with the correct destination guest and necessary state update for the virtualization related state for future fast-path processing. The Virtuoso slow-path then processes the switched packet, and if it is a new TCP connection, combines the received virtualization fast-path state with the necessary TCP state, Non-TCP packets are forwarded to guests through legacy interfaces (vNICs or veth) for processing in the legacy stack.

4.2 CPU Resource Accounting

Core-local resource accounting. The first step towards achieving isolation is to accurately account for resource use. We are particularly concerned with processor cycles spent on network stack processing on behalf of each guest. Each fast-path core tracks resources available to and used by individual guests through a local budget table, storing each guest's resource budget on that core.

Batch processing in three main tasks. The Virtuoso fast-path performs three main CPU-intensive tasks for guests: receiving packets (RX), polling guest transmit queues (POLL), and packet transmission (TX). RX dequeues incoming packets from the NIC, parses the packets, and implements the necessary TCP processing before forwarding the payload to the guest. POLL checks outgoing queues from guest applications to the fast-path for new transmission request. TX assembles complete network-virtualized TCP packets and enqueues them in the NIC. For efficiency, these tasks execute in batches, generally from multiple guests. The batch size is a compile-time parameter and primarily depends on the system's cache hierarchy; we chose 16 empirically as the value that yielded the highest throughput for our setup.

Lightweight accounting with TSCs. Virtuoso measures CPU consumption by taking CPU time stamp counter (TSC) readings at the start and end of processing for each batch. Reading the TSC is lightweight and precise. Virtuoso breaks down the TSC total to separate guests, based on each guest's number of packets. As per-packet processing costs are generally similar, this represents a reasonable trade-off between overhead for accounting and accuracy, as we will show later (§6.2, §6.3). Virtuoso then subtracts the cycles consumed from the respective guest's resource budget. To avoid frequent synchronization and coherence overhead, we maintain separate guest budget tables on each fast-path core.

4.3 Central Resource Allocation

A separate slow-path core periodically replenishes the per-core budgets on the fast-path, leveraging its global view. Separation into a parallel de-centralized fast-path and a central slow-path enables scalable and efficient coordination of the very frequently accessed per-core budgets. The slow-path replenishes the total budget in periodic 1 ms intervals and distributes the new budget to each guest. The distribution among guests is controlled by a guest weight w_g, configured by the operator. By default each guest has the same weight.

We compute budget updates u_g for guest g by recording the timestamp t' for the current update and the timestamp t for the previous update. The allocator scales the elapsed time $t'-t$ by a constant boost B. B compensates for any fast-path CPU cycles not explicitly accounted to any guest by Virtuoso to avoid over-committing processor cycles. We found the fraction of accounted cycles to be 94% (and set $B = 0.94$), with minimal processing not related to specific guests. This includes functions, such as scaling fast-path cores up and down and checking if a core can block. We multiply the product of the boost and elapsed cycles by the guests's w_g, divided by the sum of the weights of all n guests.

$$u_g = \frac{B(t'-t)w_g}{\sum_{k=1}^{n} w_k} \tag{1}$$

Preventing guest from accumulating budget. The operator also configures a budget cap C for all guests. Capping the budget prevents guests from accumulating arbitrary budgets during long periods of low utilization and starving other guests in bursty periods of activity. C restricts the number of CPU cycles Virtuoso can spend on behalf of a guest per fast-path core between replenishments. Thus, the Virtuoso slow-path finally calculates the updated per-core budget b'_{gc} for guest g on core c as

$$b'_{gc} = \min\{C, b_{gc} + u_g\} \tag{2}$$

Minimizing synchronization overheads. We update the fast-path value by performing an atomic add to the guests

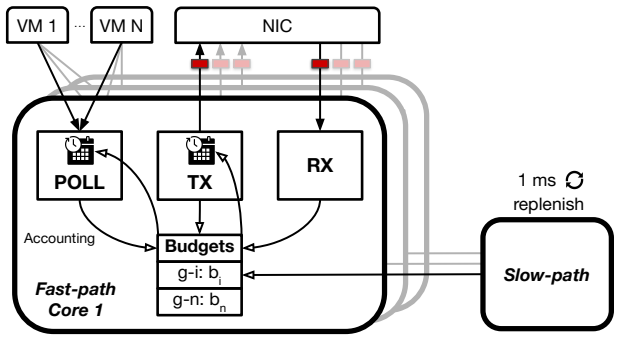

Fig. 5: Fast-path cores spend a guest's local budget as they execute processing tasks. All tasks account for resource consumption, while POLL and TX tasks schedule based on guest budgets. The slow-path periodically refills budgets.

entry in the budget table. This avoids the need for synchronization on the fast-path, and only occurs periodically.

4.4 Fine-grained Scheduling

Virtuoso performs fine-grained scheduling to enforce performance isolation based on per-guest CPU cycle budgets (Fig. 5). The scheduler performs hierarchical scheduling, first it chooses which guests to perform work for, and for sending packets determines which of the guest's flows get to transmit next. On the fast-path, before starting a task on behalf of a guest, the core consults the guest's budget, and if it is zero or negative moves on to do work for a different guest. The central resource allocator on the slow-path periodically replenishes guest budgets (§4.3).

Bounded fast-path simplifies scheduling. The observation at the center of our approach is that all individually scheduled tasks are strictly bounded, on the order of 200–500 cycles depending on packet sizes. This provides us with two key advantages. First, *preemption is not necessary*, as individual packet processing tasks complete very quickly. Second, *fine-grained batch scheduling and accurate accounting enable low tail latency and isolation*, even without knowing concrete task lengths. Tasks are all similarly sized, and, after each task completes, the next scheduling decision can compensate based on the updated budget. Even if a task overruns the budget, it will only be by a small amount of cycles and Virtuoso still precisely accounts for this with negative budgets, akin to deficit round-robin scheduling [41].

These two insights enable fine-grained scheduling for low tail latency without preemption overhead in Virtuoso. In conventional layered network stacks, switching between guests requires context switches, which disrupt CPU pipelines, cause cache overheads, and demand expensive state saving and restoration [15]. In Virtuoso, all necessary processing state for each connection is stored in the corresponding flow-state data structure in the shared stack. As a result, switching

Algorithm 1 POLL Scheduler

```
function POLL_VMS(vms)
    n ← batch_size
    for vm in vms do
        if vm.budget > 0 and n > 0 then
            x ← POLL_CONTEXTS(vm.contexts, n)
            n ← n - x
function POLL_CONTEXTS(contexts, n)
    i ← 0
    for context in contexts do
        if i < n then
            x ← FETCH_PACKETS(context.tx_head, n - i)
            i ← i + x
```

to processing a packet for a different guest incurs minimal overhead, as it does not require any context switch.

Hierarchical scheduler controls guests and flows. Virtuoso uses a two-level hierarchical scheduler. The first level decides which guest should be serviced next and the second level decides what flow (TX) or transmit queue (POLL) from the selected guest should be scheduled, using different policies. This allows us to control resource allocation between guests and between guest's flows and transmit queues. For RX, Virtuoso only performs resource accounting, as the specific guest is not known before initial processing of the packets, preventing scheduling. The following paragraphs dive into detail on scheduling Virtuoso processing tasks.

Batch-scheduling POLL. Virtuoso polls guest transmit queues for new connection send requests. The fast-path polls each guest in a batched round-robin fashion to balance efficiency and low tail latency (Alg. 1). First, the fast-path core selects the next guest and then starts polling the guest's transmit queues, until the batch is full or all the guest's queues are empty, or the guest resource budget is used up. If the batch is not full, the scheduler moves on to the next guest.

Pulling multiple transmit requests from a queue in one batch significantly reduces per-request overheads for queue access. Consolidating tasks for a specific guest within a batch also increases resource accounting accuracy as work from fewer guests is aggregated into the same batch. But even across guests, processing requests in batches enables Virtuoso to improve efficiency by avoiding cache misses on key memory accesses through group prefetching [19]. Group prefetching processes individual requests in a batch in phases, breaking before potentially expensive memory accesses, such as shared queue slots or flow state, and issuing prefetch instructions. By the time one phase has been executed for all requests, prefetch instructions for the first packet will hopefully have completed, thereby avoiding the cost of demand misses. During the processing of these transmit requests, the Virtuoso transport layer schedules the corresponding flows for packet transmission through TX tasks.

Algorithm 2 TX Scheduler

```
function SCHEDULE_VMS(vms)
    n ← batch_size
    for vm in vms do
        if vm.budget > 0 and n > 0 then
            x ← SCHEDULE_FLOWS(vm.flows, n)
            n ← n - x
function SCHEDULE_FLOWS(flows, n)
    i ← 0
    for flow in flows do
        if i < n then
            x ← SCHEDULE_PACKETS(flow.packets, n - i)
            i ← i + x
```

Scheduling TX for per-guest fairness. Virtuoso also schedules TX tasks with a similar batched hierarchical approach (Alg. 2). The scheduler first chooses the next guest, and then the guest's next flow to transmit a packet. In the first level of the scheduler the round-robin algorithm decides which guest should send next. The second level instead schedules flows according to a priority queue that tracks the earliest time when each flow should send next. The TCP processing logic determines these timestamps with a split fast-path/slow-path congestion control scheme [20]. These timestamps also automatically ensure that a guests' flows are serviced fairly. Guests without available budget are skipped until the budget is replenished.

Scheduling RX is avoided. Virtuoso does not schedule RX tasks and instead only performs resource accounting. RX performs complete receive processing, starting with pulling a packet from the NIC receive queue. As NIC receive queues do not distinguish guests, Virtuoso cannot know which guest a packet will be for until significant processing, including the flow state lookup, has been performed. At this point, re-enqueuing the packet on a separately scheduled per-guest queue would incur resource overhead comparable to just finishing processing.

Guests can still receive packets with a depleted budget, but Virtuoso keeps track of the cycle deficit accrued when later replenishing the budget. Guest that deplete their budget on RX tasks as a result have fewer cycles available for POLL and TX tasks. For the workloads we tested, this results in the overall system self-correcting, as senders that do not receive replies will stop sending.

An alternative is to drop received packets once the guest has been identified and is found to have no budget for processing the packet. However, while this might improve scheduler accuracy, it comes at the cost of efficiency. Getting to the point of dropping the packet, already incurs significant processing cycles. TCP will later re-transmit any packet that has been dropped, again requiring resources for processing this re-transmitted packet.

4.5 Secure Shared Stack

Virtuoso processes packets from multiple guests and applications in the same network stack. Resource accounting and scheduling mechanisms provide performance isolation. For this we rely on shared memory queues between individual application cores and the fast-path, as well as a separate slow-path for more expensive control operations, akin to TAS [20] and SNAP [43]. However, we also need security enforcement while enabling efficient direct communication between applications and the Virtuoso stack.

Protecting memory regions. Virtuoso ensures security isolation for the guest and application interface through memory isolation. We allocate different guests' queues and connection buffers in separate shared memory regions only mapped into a single guest and the fast-path. To avoid leaks due to dynamic remapping and ensure resource isolation, Virtuoso statically pre-allocates the complete shared memory region when the guest starts. Virtuoso also has a narrow shared memory interface comprising just guest receive and send queues along with connection payload buffers. This narrow interface provides no other attack vectors, such as complex data structures that could interfere with Virtuoso.

5 Implementation

Virtuoso runs in host userspace as a separate service and provides all features of a typical TCP stack to guest applications. Virtuoso maintains TCP protocol and sockets API drop-in compatibility. For fast NIC access, we use DPDK [16]. We build our prototype using TAS [20] as a basis. We heavily modify and extend the TAS fast and slow-path, but retain the sockets emulation library unmodified. Virtuoso supports guest VMs as well as guest containers. The Virtuoso prototype comprises 20,918 lines total, 4,669 lines for the fast-path, 5,536 lines for the slow-path, 2,437 lines for the hypervisor integration, and 1,029 lines of modification to OvS.

5.1 Support for Multiple Guests

To extend TAS to securely connect to multiple guests, we modify it to use multiple separate shared memory regions and Unix sockets. Virtuoso creates separate shared memory regions on the host for each guest. These regions contain transmit request and receive notification queues, as well as per-flow RX and TX circular payload buffers. During guest initialization, Virtuoso needs to pass the newly created shared memory region to the guest. As in TAS we implement this using Unix domain sockets, that carry a handshake along with the shared memory file descriptor. Unlike TAS, Virtuoso exposes separate listening Unix sockets for each guest, allowing it to securely identify which guest is connecting, assuming sufficient access control on the host.

5.2 Virtuoso with Containers

Given that guest containers share the same host operating system as Virtuoso, connecting applications in guest containers is simply a matter of mapping the respective guest Unix socket into the container. After this any container guest application can interact with Virtuoso just as a native application with exactly the same performance.

5.3 Virtuoso with Virtual Machines

For virtual machine guests, initialization is more complex as unix sockets and file descriptors are by definition local to the host. Virtuoso instead integrates with the hypervisor to directly map shared memory regions via a dummy PCI device. In the VM, a Virtuoso guest agent implements a user space driver for this dummy device and translates the interface into a compatible Unix socket and shared memory file descriptors again, avoiding the need for applications to distinguish between native, container, and VM operation.

We leverage QEMU [36] with KVM and its Inter-VM Shared Memory Device (IVSHM) [17] to implement the hypervisor part of the Virtuoso integration. IVSHM already allows an external application to send a shared memory file descriptor and eventfds for bi-directional interrupts to QEMU that are then exposed as a PCI device with the memory region as a directly memory mapped BAR. For ease of integration, we implement this as a separate host proxy process that connects to QEMU and Virtuoso.

In the guest we implement a Virtuoso guest agent, that leverages the vfio-pci [46] kernel module to implement a user space driver for the dummy PCI device. vfio-pci provides a file descriptor that the application can mmap for access to the BAR, along with eventfds for interrupts. The guest agent then creates a listening unix socket, and during the handshake passes the file descriptors directly to applications.

This results in a directly shared memory region between Virtuoso on the host, and applications in the guest VM. As a result, fast-path interactions with Virtuoso incur no additional overheads compared to containers or native applications.

5.4 OvS Slow-path

We use OvS for virtualization management, to identify tunnelling information, and to determine the destination VM for a flow. To that end, we modify OvS to exchange packets and control information with Virtuoso. In OvS we implement custom transmit and receive netdev-provider ports. The receive port polls Virtuoso for new packets and passes them to OvS. OvS then performs internal matching based on the packet metadata and directs it to a transmit netdev-provider port. The transmit port holds tunnelling information for a packet and establishes a message queue with Virtuoso to dispatch this information. This message queue exchanges inner and outer IP addresses for encapsulated packets, tunnel IDs from GRE headers, and the appropriate ID for the destination VM.

6 Evaluation

In this section we evaluate how well Virtuoso addresses the goals outlined in §2. To that end, our evaluation aims to answer the following questions:

- Does sharing the stack improve resource utilization? (§6.1)
- Can fine-grained scheduling and resource accounting ensure isolation of tenants despite sharing resources? (§6.2)
- How close can optimized one-shot virtualization performance get to native un-virtualized stacks? (§6.3)
- Does Virtuoso scale to serve many guests? (§6.4)

Testbed. We configure two identical machines as client and server. They are directly connected with a pair of 100 Gbps Mellanox ConnectX-5 Ethernet adapters. Both machines have two Intel Xeon Gold 6152 processors at 2.1 GHz, each with 22 cores for a total of 44 cores and 187 GB of RAM per machine. We run Linux kernel 5.15 with Debian 11.

Baselines. We compare Virtuoso against a number of baseline configurations. For these we use two existing network stacks, the default in-kernel Linux network stack, and the optimized TAS TCP stack. Depending on the configuration, we run these bare metal, or in QEMU/kvm virtual machines with virtio-net vNICs connected to OvS. We configure OvS with the DPDK backend and use vhost-user between QEMU and OvS to get the best baseline performance. For containers, we directly mount the respective Unix socket into the containers for Virtuoso.

Focus on VMs. For most of our evaluation we focus on Virtuoso with virtual machine guests, rather than container. The dominating fast-path interaction performance is identical between Virtuoso VM guests and container guests, while some slow-path interactions in the VM case are more expensive than for containers (§6.3). At the same time we found Virtuoso to provide higher relative benefits when comparing to existing container stacks than compared with VM stacks. Thus, Virtuoso with VMs provides a conservative evaluation and performance comparison.

6.1 Sharing the Stack Increases Utilization

We start off by measuring resource utilization with bursty guest workloads. For this we provision four guest VMs with echoservers responding to RPCs. Clients generate bursty high-low traffic, separately saturating each guest during peaks, and sending 2.5 M requests per second and guest during low periods. We then vary the degree of overlap, i.e. how many of the guests burst concurrently, from 50% all the way to 100%. At 50% burst overlap we have two guests bursting

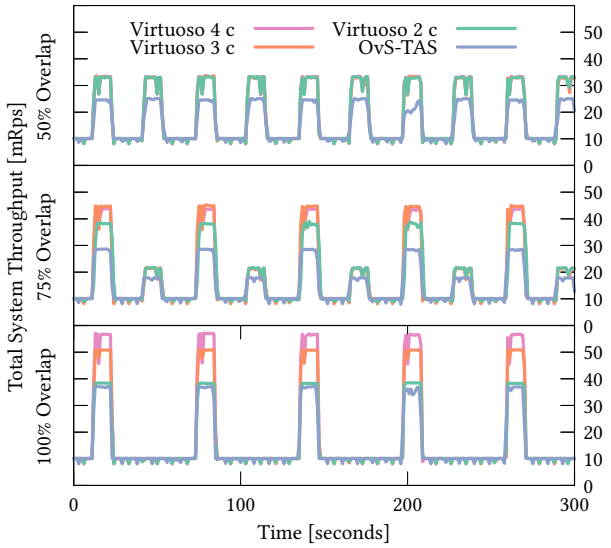

Fig. 6: Resource efficiency with bursty guests and a varying number of Virtuoso fast-path cores. Virtuoso achieves higher throughput with the same or less resources.

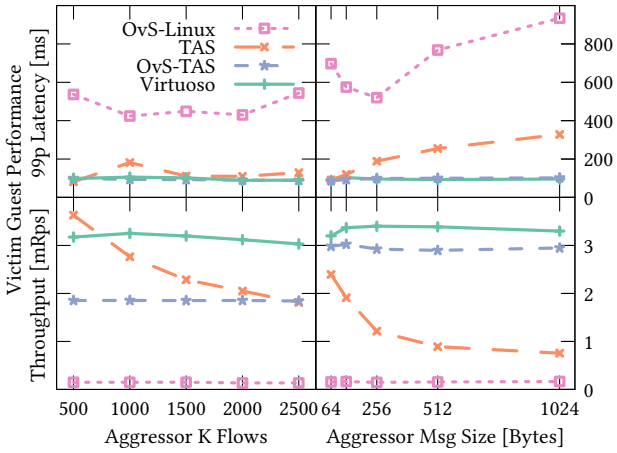

Fig. 7: Measuring performance isolation of a victim guest from an aggressor guest with varying flows and messages sizes. Virtuoso achieves tail latency on par with the siloed OvS-TAS, and significantly higher throughput.

every 50 seconds for 10 seconds. At 75% overlap three guests burst at the same time and one guest bursts by itself. At 100% overlap all guest burst at the same time.

As a baseline, we use separate TAS network stacks in each guest connected to OvS on the host (OvS-TAS). For the baseline we provision each guest with 5 cores, and configure the TAS instances to use one fast-path core. For Virtuoso we instead provision each guest with 4 cores, and 2, 3, or 4 of the saved cores as shared fast-path cores for Virtuoso.

Fig. 6 shows the measured aggregate RPC throughput across all guests. When 50% of the guests burst at the same time, Virtuoso can handle the bursts with only two cores instead of the four cores for OvS-TAS and achieves 30% higher throughput. With 75% overlap, three fast-path cores are sufficient and during peak obtain 51% higher throughput compared to OvS-TAS. At 100% burst overlap, we need all 4 cores are necessary to achieve the maximum throughput, 44% higher than OvS-TAS.

Our results show sharing the stack allows Virtuoso to pool resources and thereby significantly improve resource utilization and overall system efficiency. Virtuoso achieves significantly higher throughput with fewer cores than the baseline.

6.2 Fine-grained Scheduling Isolates VMs

Next, we evaluate Virtuoso ability to isolate guests despite sharing a network stack and underlying resources. To that end, we evaluate two main performance metrics, latency and throughput, for a "victim" guest while a separate aggressor guest attempts to introduce performance interference.

We evaluate two different forms of interference, by separately varying the number of aggressor connections and the size of the aggressor messages. For the former the aggressor uses a fixed message size of 64 bytes, and for the latter a fixed number of 500 connections. The victim uses one connection with 64 B messages for the latency measurements, and 500 connections with 64 B messages for the throughput measurement. The victim uses a single core VM, while we provision the aggressor VM with a core for every 500 connections. Both victim and aggressor use the RPC echoserver.

We compare this workload across different system configurations. Virtuoso with two fast-path cores, OvS-TAS with one additional fast-path core per guest VM for the TAS instance, the guest Linux stack with OvS with no additional guest cores. Finally, we also compare to native TAS by running victim and aggressor as separate processes on the host connecting to the same TAS instance with two fast-path cores. We use the timely [28] congestion control algorithm in Virtuoso, OvS-TAS, and TAS. In all cases VMs, processes, and network stacks are pinned to dedicated cores.

Fig. 7 shows the results. At a high level, the results confirm that Virtuoso's fine-grained isolation retains tail latencies at the same level as siloed OvS-TAS, while significantly improving victim throughput. TAS without isolation increases tail-latency significantly as the aggressor's message size increases. Flow rate-limiting is able to mitigate rising tail-latencies for TAS when an aggressor increases the number of flows, but it still incurs tail latencies above Virtuoso and OvS-TAS. For example, at 2500 aggressor connections Virtuoso achieves a 99p latency of 90 μs, OvS-TAS's 89 μs,and TAS's 129 μs. The benefits of fine-grained scheduling also hold when comparing average latencies of the baselines. The Virtuoso victim achieves 59 μs 50p latency when the aggressor VM sends 1024 B messages, while the TAS and OvS-TAS clients achieve 241 μs and 79 μs average latencies.

Virtuoso Features			Cycles	
Scheduling	VMs	GRE	/ RPC	Overhead
✗	✗	✗	1 152	
✓	✗	✗	1 181	+ 2.5%
✓	✓	✗	1 263	+ 9.6%
✓	✓	✓	1 285	+ 11.5%

Tab. 1: Request processing times for different Virtuoso features relative to the native baseline (TAS).

We also measure similar results with the victim's throughput. TAS sees a decrease in throughput as the aggressor VM increases the number of connections or message size. With Virtuoso, the victim maintains similar throughput as the aggressor attempts to acquire more resources, in all cases higher than OvS-TAS. For 2500 aggressor connections, Virtuoso achieves 65% higher throughput than OvS-TAS.

6.3 One-shot Processing Reduces Overhead

Now we evaluate how one-shot processing for network virtualization affects overhead, latency, and throughput.

Virtualization overhead. First, we seek to measure and break down the overheads of adding network virtualization features to the network stack. For this, we start with the TAS fast-path as the baseline and profile the number of processor cycles required to process an RPC request including sending the response. The application workload is saturating a single Virtuoso core with 64 B RPCs. We then successively add Virtuoso features, starting with scheduling, then VM integration, and finally GRE tunneling.

Tab. 1 shows the results. Fine grained scheduling and resource accounting adds around 30 cycles or 2.5% to each RPC. Enabling VM integration adds 82 cycles, and finally GRE tunneling adds another 22 cycles per RPC. In total, the additional functionality in Virtuoso only adds a total of 133 cycles or 11.5% of overhead. We conclude that one-shot processing is effective for avoiding expensive overhead for significant additional network virtualization functionality.

Latency. These minimal overheads should translate to minimal latency increase for virtualized guests in Virtuoso compared to TAS. We now measure the small 64 B RPC latency, both for long-lived connections and short-lived connections that only carry one RPC before closing and re-opening, also including latency for establishment and tear-down. We record latency distributions for all our system configurations and report the results in Fig. 8.

With long flows Virtuoso achieves mean latencies of 5 μs compared to 4 μs with bare-metal TAS, and 7 μs 99p latency compared to 5 μs for TAS. OvS-TAS only achieves median latencies of 11 μs and a 99p latency of 16 μs, both about a factor of two higher than Virtuoso. Native Linux and Linux VMs

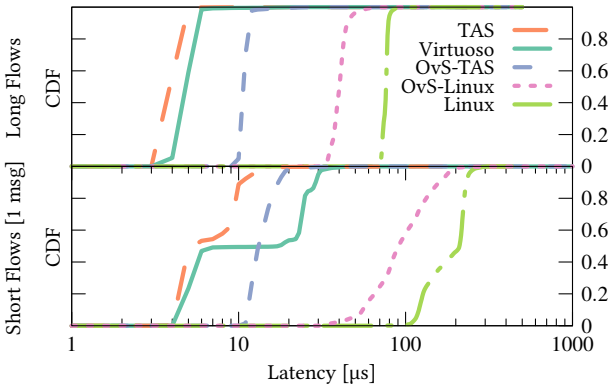

Fig. 8: RPC latency distribution across different network stacks. For long-lived connections Virtuoso adds minimal overhead relative to TAS, while tail latency for short-lived connections, the Virtuoso worst-case, remains competitive.

with OvS are both significantly worse, although interestingly we found that the DPDK drivers in OvS seem to reduce overheads for compared to the native in-kernel drivers, thereby surprisingly lowering the latency.

Short-lived connections are Virtuoso' Achilles heel, as one-shot connection state management optimizes for fast access to established state at the cost of overhead for adding and removing connections. The extreme case of connections that send only one RPC before being torn down again, factoring in complete time for establishment and tear-down, probes this. For average latency Virtuoso is again only marginally slower than TAS without OvS, at 8 μs compared to 6 μs and far below OvS-TAS' 14 μs. But in the tail Virtuoso shows 99p latencies of 36 μs compared to TAS' 15 μs. Tail latency is even moderately higher than with OvS-TAS, although we suspect that this is due to inefficiencies in the Virtuoso connector in OvS that may be less optimized than the vhost-user port we use for OvS-TAS. Linux is again far slower.

We conclude Virtuoso enables virtualized networking with minimal latency overhead compared to unvirtualized stacks.

Throughput. One-shot virtualization processing also allows Virtuoso to achieve high throughput, comparable to bare metal performance. In Fig. 9 we dedicate the same number of cores to the networking stack in a client and server machine running an RPC echo server. Server and client applications run on 12 cores each, and we dedicate 5 cores to Virtuoso and TAS. We again measure throughput for short and long-lived connections.

We first vary the number of messages per flow and measure throughput. The more expensive Virtuoso slow-path is apparent for short-lived connections, but as more messages are sent per connection the gap between Virtuoso and bare-metal solutions decreases. Virtuoso also achieves throughput competitive with TAS for long-lived connections. For 1024 B messages Virtuoso reaches throughput only 14% lower than TAS, while OvS-TAS shows a performance drop of

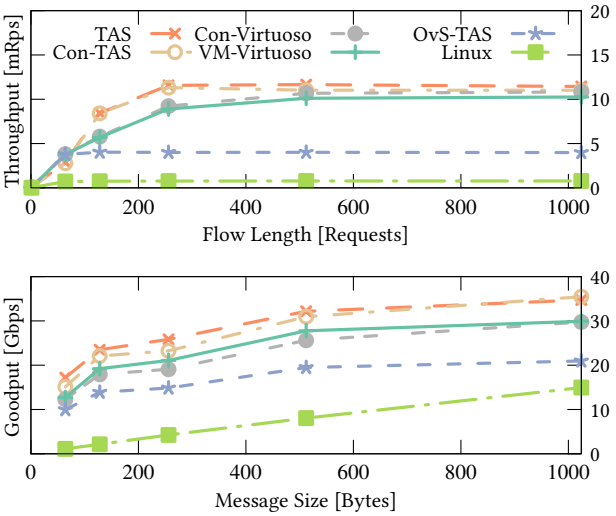

Fig. 9: Throughput with varying flow lengths and message size. Virtuoso offers throughput close to the un-virtualized TAS baseline, and much higher than other virtualized stacks.

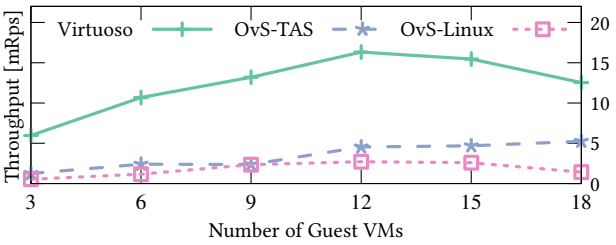

Fig. 10: Virtuoso significantly outperforms alternative stacks even with many guests, and scales for cloud setups.

40%. Baselines with containers show that there is little to no overhead between running Virtuoso in VMs (VM-Virtuoso) as opposed to containers (Con-Virtuoso). Linux is again not competitive. We conclude one-shot processing also enables high-throughput virtualized network communication.

6.4 Virtuoso Scales to Many Guests

Finally, we evaluate guest scalability in Virtuoso. For each run we provision two cores for each guest VM and measure the aggregate throughput as the number of guests increases. Each VM runs an RPC echoserver loaded 100 connections sending 64 B messages. We use five fast-path cores for Virtuoso with one polling core in OvS. In the OvS-TAS and OvS-Linux baselines we assign six polling cores to OvS.

Fig. 10 shows the results. Virtuoso sees a steady increase in throughput up to 12 guest VMs, while OvS-TAS and OvS-Linux do not scale as well. At 12 VMs, Virtuoso achieves 260% higher throughput than OvS-TAS and 500% higher throughput than OvS-Linux. OvS-TAS needs at least one core for the slow-path and one core for the fast-path, so in this setup TAS cores inside a VM compete with the application for resources, resulting in a smaller performance gain when compared to OvS-Linux. We expect the small throughput drop past 12 guests is due to a sub-optimal polling implementation in the Virtuoso fast-path that incurs overheads when polling too many queues. We leave optimization of this as future work. We conclude that at scale Virtuoso provides throughput significantly above the alternatives and scales to the number of guests on typical cloud servers.

7 Related Work

Shared hypervisor-level network stack. NetKernel [30] proposes a fundamental re-design of the software stack for virtual network processing, by extracting the network stack from VMs and sharing it between multiple virtual machines on the host. Nonetheless, NetKernel keeps virtual switching and network virtualization separate separate from the rest of the network stack. It also provides limited isolation and does not evaluate support for microsecond latencies. Unfortunately NetKernel is not available for comparison.

Container overlay networks. Container overlay networks [5, 11, 48] provide network virtualization features and allow containers to communicate using their own independent IP addresses. They often have large overheads because packets have to traverse virtual switches and the network stack twice. Slim [53] optimizes network virtualization for containers and does not require packet transformations in the data plane. However, it does so by completely avoiding protocol-level network virtualization and instead simply sends packets on the physical network only translating address info in socket calls. As a result, Slim only works for networks that exclusively use Slim. Slim also does not provide additional mechanisms over Linux for performance isolation and lacks support for VM guests.

Mixed granularity scheduling. The Scout OS [29] defines a path abstraction for data flow between a source and end device. This abstraction can schedule work at different path granularities, allowing schedulers to drop work if the deadline for a path is not going to be met and separating between high and low priority work in the beginning of a path. Nonetheless, Scout has not been designed for challenges of the μs-scale modern workloads targeted by Virtuoso.

High-speed packet processing. Prior work has investigated reducing overheads in software packet processing by minimizing layer crossings. TAS [20] also splits operation between a streamlined fast-path with dedicated cores and a slow-path for control operations, but does not provide network virtualization or performance isolation. Arrakis [34] creates virtualized devices and lets applications conduct I/O through these virtual devices without involving the kernel, but does not provide the benefits of fully fledged network virtualization. StackMap [50] and mTCP [18] both reduce overheads from the socket API, but only one application can

access a NIC port, thus forgoing opportunities for sharing resources. Netmap [38] reduces the costs of moving traffic between the hardware and the host stack. It can run inside a virtual machine to reduce overhead, but does not address the inneficient datapath packets traverse after leaving VMs.

8 Conclusion

With Virtuoso we have shown that network processing for virtual machines and container environments can be implemented efficiently in software. By sharing resources and using fine-grained scheduling for isolation Virtuoso achieves resource utilization far above other alternatives. And one-shot network virtualization enables implementation of the necessary virtualization functionality with minimal overhead over optimized bare metal stacks. We expect that our techniques can generalize to other protocols and implementation on other architectures, such as SoC-SmartNICs.

Acknowledgments

We thank Florian Bauckholt and Mehrshad Lotfi for proof-of-concept prototypes of the Virtuoso VM and OpenVSwitch integration respectively. We also thank Rajath Shashidhara for his contributions in the long running discussions over the course of this project.

References

[1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *21st ACM Workshop on Hot Topics in Networks*, HotNets, 2022.

[2] Amazon Web Servics. AWS Nitro system. https://aws.amazon.com/ec2/nitro/.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[4] Jeffrey Dean and Luiz André Barroso. The tail at scale. *ACM Transactions on Computer Systems*, 56(2):74–80, February 2013.

[5] Docker overlay. https://docs.docker.com/network/.

[6] G. Dommety. Key and sequence number extensions to GRE, September 2000. RFC 2890.

[7] Peter Druschel, Larry Peterson, and Bruce Davie. Experiences with a high-speed network adaptor: A software perspective. In *1995 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 1995.

[8] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference*, ATC, 2019.

[9] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic routing encapsulation (GRE), March 2000. RFC 2794.

[10] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2017.

[11] Flannel. https://github.com/flannel-io/flannel.

[12] P. Garg and Y. Wang. Nvgre: Network virtualization using generic routing encapsulation, September 2015. RFC 7637.

[13] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2021.

[14] J. Gross, I. Ganga, and T. Sridhar. Geneve: Generic network virtualization encapsulation, November 2020. RFC 8926.

[15] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. A case against (most) context switches. In *18th Workshop on Hot Topics in Operating Systems*, HOTOS, 2021.

[16] Intel data plane development kit. http://www.dpdk.org/.

[17] Inter-VM shared memory device – QEMU documentation. https://www.qemu.org/docs/master/system/devices/ivshmem.html.

[18] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[19] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the bar for using GPUs in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2015.

[20] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *14th ACM European Conference on Computer Systems*, EuroSys, 2019.

[21] Hsiao keng Jerry Chu. Zero-copy TCP in Solaris. In *1996 USENIX Annual Technical Conference*, ATC, 1996.

[22] M. Kerrisk. veth - virtual ethernet device. https://man7.org/linux/man-pages/man4/veth.4.html, February 2023.

[23] I.M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996.

[24] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: datacenter sockets can be fast and compatible. In *2019 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2019.

[25] Yan Luo, Eric Murray, and Timothy L Ficarra. Accelerated virtual switching with programmable nics for scalable data center networking. In *2nd ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, VISA, 2010.

[26] Aravind Menon, Alan L Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *2006 USENIX Annual Technical Conference*, ATC, 2006.

[27] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *1st International Conference on Virtual Execution Environments*, VEE, 2005.

[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *2015 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2015.

[29] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *2nd USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 1996.

[30] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making network stack part of the virtualized infrastructure. In *2020 USENIX Annual Technical Conference*, ATC, 2020.

[31] NVIDIA. ConnectX-7 400G Adapters. https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds December 2022.

[32] NVIDIA. NVIDIA Bluefield-3 DPU. https://resources.nvidia.com/en-us-accelerated-networking-resource-libra datasheet-nvidia-bluefield?lx=LbHvpR&topic=networking-cloud, March 2023.

[33] Open vswitch. https://www.openvswitch.org/.

[34] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 33(4):11:1–11:30, November 2015.

[35] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with shared receive rings. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2023.

[36] QEMU – the FAST! processor emulator. https://www.qemu.org/.

[37] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian chin Wang, and K.K. Ramakrishnan. SPRIGHT: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *2022 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2022.

[38] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference*, ATC, 2012.

[39] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Enso: A streaming interface for NIC-Application communication. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2023.

[40] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2022.

[41] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *1995 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 1995.

[42] M. Mahalingam Storvisor, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks, August 2014.

[43] Weibin Sun and Robert Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In *9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2013.

[44] David L. Tennenhouse. Layered multiplexing considered harmful. In *Protocols for High Speed Networks I*, PfHSN, 1989.

[45] M. Tsirkin and C. Huck. Virtual i/o device (VIRTIO) version 1.2. https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html, July 2022.

[46] VFIO - "virtual function I/O". https://docs.kernel.org/driver-api/vfio.html.

[47] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *15th ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[48] Weave. https://www.weave.works/.

[49] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. The resource pooling principle. *SIGCOMM Computer Communication Review*, 38(5):47–52, September 2008.

[50] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference*, ATC, 2016.

[51] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel datapath OS architecture for microsecond-scale datacenter systems. In *28th ACM Symposium on Operating Systems Principles*, SOSP, 2021.

[52] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with ebpf. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2023.

[53] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a Low-Overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2019.