EXPLAINING AND EVALUATING THE USE OF RDMA IN HIGH PERFORMANCE CONTAINERS

by

Emerson Ford

A Senior Honors Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the

Honors Degree in Bachelor of Science

In

Computer Science

Approved:

Abstract

Containers are an increasingly popular packaging framework for complex applications, offering benefits such as lightweight isolation, portability, and ease of deployment. These benefits have the potential to solve a myriad of issues that have long been present in the high performance computing world, presenting a compelling narrative for their use in these environments. Unfortunately, standard container runtimes rely on relatively slow, virtualized network stacks to achieve network isolation and portability, which are incompatible with the kernel bypass networking technology RDMA. This has limited the widespread adoption of containers in high performance computing environments where RDMA-reliant applications are quite common.

Recent projects such as FreeFlow and Mellanox's RDMA hardware have created solutions for enabling RDMA inside of containers while maintaining varying degrees of the benefits of standard containers. However, despite the strong claims made by these solutions, many of their characteristics such as performance, isolation sacrifices, and scalability are either not well documented or simply not known; these characteristics being critical to assess for several high performance computing use cases. This thesis attempts to remedy this issue by identifying, explaining, measuring, and comparing these characteristics for the various solutions available today for enabling RDMA in containers; ultimately providing high performance computing end-users a more holistic perspective on which solutions may be most viable for their environment and use case.

1 Introduction

Containers offer a promising solution to a host of issues that have long plagued the high performance computing (HPC) world from dependency management to portable and reproducible environments to even lightweight sharing of hardware resources; all of which being achieved through the use of statically built images and standardized container runtimes [59]. For users, cluster administrators, and application developers, this would provide more freedom and ease in application development and deployment as it eliminates the burden of working around the hundreds of possible environments on which the application may be deployed. In addition, this also opens the door for easy migration and utilization of new, "container first" HPC platforms such as the cloud, Kubernetes, and SLATE that offer new compute and scheduling capabilities [3].

However, despite these numerous benefits, the adoption of containers in the HPC world has been notably slow [58]. Among other issues like the until-recent lack of rootless container runtimes, standard container network stacks do not support Remote Direct Memory Access (RDMA), a networking technology that provides extremely low latency and high throughput with minimal CPU overhead by offloading memory accesses from the CPU to NIC hardware (hence the term "kernel bypass networking") [52]. This lack of support heavily restricts the use of native containers in HPC environments as trends like disaggregated storage, multi-machine workloads, and increased use of parallel libraries such as MPI have necessitated the use of high performance networking like RDMA to maintain performance. To make the lack of RDMA support even worse, standard container network stacks by themselves have been shown to be both CPU intensive and significantly reduce container

networking performance [2, 20, 24, 57, 60], further worsening the networking downsides of containers in HPC environments.

Unfortunately, enabling support for RDMA in containers (and fast networking as a whole) is a nontrivial task. Container runtimes rely on "virtualized" network stacks for creating flexible network overlays, enforcing isolation, and providing portability [57]. These network stacks have, for the most part, only been expressible through software layers provided by the host kernel, adding significant software overhead to the processing of all network packets originating from and destined to containers. Due to this, these network stacks are fundamentally incompatible with RDMA which requires bypassing the host kernel entirely for its performance guarantees.

While it is possible to run containers without container networking by exposing the host's NIC directly to running containers (a mode called "host networking"), this comes with several notable disadvantages: (1) controlling fair sharing of the RDMA-capable NIC between multiple containers cannot be done programmatically, (2) containers are not relocatable/portable across multiple hosts, (3) common container orchestration tools like Kubernetes cannot be used, and (4) network isolation and network routing policies for containers is disabled. Thus, while this mode may be a viable for some environments, it is not adequate for the vast majority of container environments — such as those that utilize Kubernetes heavily — and additional solutions for enabling RDMA in containers are still needed.

Multiple groups have recently developed solutions for enabling RDMA in containers, each with certain sacrifices in container and/or RDMA guarantees to overcome this incompatibility. The challenge is then to determine which solution (and subsequent sacrifices

in guarantees) works best for a given environment. This is the core problem this thesis seeks to address. It is currently unnecessarily difficult to pick a solution due to a lack of exhaustive performance data and thorough explanations on exactly which guarantees are sacrificed for these various solutions. This information is critical in HPC environments as applications may heavily rely on RDMA's low latency and/or high throughput guarantees; similarly, container security and isolation guarantees may be a requirement for those HPC environments that are multi-tenant or are running untrusted applications.

Therefore, this thesis will analyze multiple properties of the several solutions available today for enabling RDMA in containers, such as SoftRoCE [34], Shared HCA, SR-IOV [51], FreeFlow [22], and MasQ [15]; these properties being grouped in two main categories:

Container RDMA Network Properties:

- *Network Isolation*: each container has its own interface, routing tables, and network policies (also known as a network namespace); further, a container's IP should not inherently depend on the underlying NIC's network
- *Controllability*: enforcing routing policies and traffic shaping on a container with an RDMA-capable NIC
- *Resource Utilization*: container networks should not be CPU or memory intensive

RDMA Properties:

- *Throughput*: how closely can a solution match host RDMA throughput
- *Latency*: what latency overhead is incurred per message for a given solution

and analyzed with varying levels of:

- number of virtualized RDMA devices on the cluster and host (e.g. where each virtualized RDMA device is allocated to a single container)

- message size of network payloads

which should adequately test the scalability of these properties. Finally, comments will also be made on:

- *Scalability Limits*: are there any other hardware or software limits on the NIC for scalability
- *Proprietary*: can a solution be used across various RDMA-capable NIC vendors
- *Maturity*: what support exists for a solution, how well tested is a solution
- *Ease in Deployment*: are application changes necessary for a solution, how difficult is it to deploy the solution
- *Execution Privileges*: can the solution be used without elevated capabilities such as `CAP_SYS_ADMIN`
- *Network Pressure*: what additional pressure do these solutions put on the network

to assist in determining if a given environment can even support a given solution.

Additionally, these solutions can be generalized to three implementations: pure software, paravirtualization-like, and pure hardware; with each solution in an implementation likely having similar properties. Thus, this thesis should also provide general insight as to which implementations may be best suited for a given HPC environment.

2 Background

2.1 Networking Planes

The state and logic of networking operations can be generally abstracted into three planes: the control plane, the data plane, and the management plane. Understanding this abstraction is vital for understanding how these various RDMA in container solutions operate and why they exhibit their specific performance characteristics.

The *control plane* handles the rules of how and to where network packets are forwarded. Operations such as initiating connections, terminating connections, and connection resource management are considered as part of control plane operations and policies such as firewalls, routing policies and tables, traffic shaping, and interface properties (e.g. IP addresses) are considered as part of control plane policies. It is desirable to have the control plane be highly flexible and configurable to allow for a wide expression of network topologies.

The *data plane*, also known as the forwarding plane, handles the actual transmission of network data based on the configured logic of the control plane. Operations such as memory copies into network buffers and packet transmission over the wire are considered as part of data plane operations and policies such as the size of network buffers are considered as part of data plane policies. It is desirable to keep the data plane as low overhead as possible as overhead in the data plane has a significant impact on network performance due to every byte of a packet having to traverse the data plane. Notably, it is the data plane that handles the "execution" of control plane policies, thus the control plane is only as expressive as the data plane allows.

The *management plane* handles the configuration of the control plane and data plane

policies. Its operations consists primarily of using software tools such as the iproute2 collection of tools, `iptables` / `nftables`, `DHCP`, and `ovs-vsctl` that take input to reconfigure the specified plane's policies. The management plane's policies primarily consist of security; specifically, defining who has access to use its tools, with what capabilities, and in what contexts.

In traditional host networking, the control plane and the data plane[1] exist as software in the kernel. This provides an extremely high degree of flexibility and configurability for the control plane and data plane, but it results in the kernel being interposed in every networking operation. As a result, this incurs performance penalties from the overhead of context switches, memory copies from user-space to kernel-space (or from RAM to CPU to the network device), and from simply being software (i.e. not being hardware accelerated or run on FPGAs). On top of this, use of standard network protocols like TCP may exacerbate the performance penalties due to higher code complexity, built in retries and throttling, and protocol header overhead.

2.2 RDMA Protocol

Remote Direct Memory Access (RDMA) is an aptly named network protocol designed for the direct reading and writing of the memory of remote hosts with extremely low latency, high bandwidth, and minimal CPU overhead. Support for it has been added to a number of parallel applications and libraries, such as TensorFlow and OpenMPI, and has seen wide spread adoption in the high performance computing world where it is run on top of either Infiniband or Ethernet [22]. At its core, RDMA is a form of kernel-bypass networking in

[1]barring the actual transmission on the wire which the NIC handles

that it shifts almost all of the data plane and control plane from the kernel to a combination of user-space and the network card. While this eliminates most of the control plane and data plane configurability provided by the kernel, it provides substantial improvements in network latency and bandwidth. For example, current top-of-the-line RDMA enabled NICs are capable of achieving sub-1µs latency with >100Gb/s flows, performance not possible with even the most aggressive tuning of the Linux networking stack [1, 18, 27].

Understanding the data paths, architecture, and implementation of network planes for RDMA itself is critical for understanding how and why these various RDMA in containers solutions work and will therefore be explained.

2.2.1 Data Plane

While traditional host networking relies on kernel syscalls and interrupts as a communication channel for signalling packet sends and receives, RDMA instead utilizes in-memory queues that live in regions of memory explicitly shared between the user-space process and the NIC[2] as its data plane communication channel. A user-space process will write a *work request* (WR) to the designated TX Queue to initiate a message[3] send. The NIC will spin-loop on these shared-memory queues to detect when a work request has been posted. Inside of this work request will be a fat pointer[4] to another region of shared memory — this being the actual message payload. The NIC will consume the posted work request, read the payload directly from memory (a process known as Direct Memory Access or DMA), then transmit the payload over the wire, all without any kernel or CPU interposition. The user-space

[2]the NIC using IOMMU to directly access memory without CPU interposition

[3]in contrast to TCP and UDP where the user-space process handles chunking a message into packets, RDMA operations post full messages and the NIC handles chunking the message into packets

[4]fat pointers being a pointer with extent information

process is then notified of whether that message send succeeded or not by a *completion queue event* (CQE) that is posted by the NIC to the designated Completion Queue.

Following this, a user-space process can receive messages by posting a *receive request* (RR) to the designated RX Queue, with these requests containing fat pointers to writable buffers in shared memory. Only after a receive request with a memory buffer large enough has been posted will a NIC accept an incoming message. When an incoming message is accepted, the NIC will similarly write the incoming payload to the specified shared-memory buffer found in the receive request [32].

It is worth noting that what is described above is technically considered *two-sided* RDMA operations (these use the verbiage *send* and *receive*). RDMA is also capable of *one-sided* operations that consist of directly reading and writing the memory of the remote host (these use the verbiage *write* and *read*) without the remote host being involved in or aware of such operations, i.e. the remote user-space process does not need to post receive requests. These one-sided operations follow a nearly identical data path to two-sided operations, minus the need for the RX Queue.

2.2.2 Control Plane

With the data plane split between shared-memory and the NIC, it then follows that the majority of the RDMA control plane must also live between shared-memory and the NIC. This, in conjunction with RDMA having a different implementation for each network fabric, results in the RDMA control plane having a radically different paradigm than that of traditional host networking. Further, the RDMA control plane has the notion of both a *device* and *interface*, where the device is what is interacted with for RDMA operations and

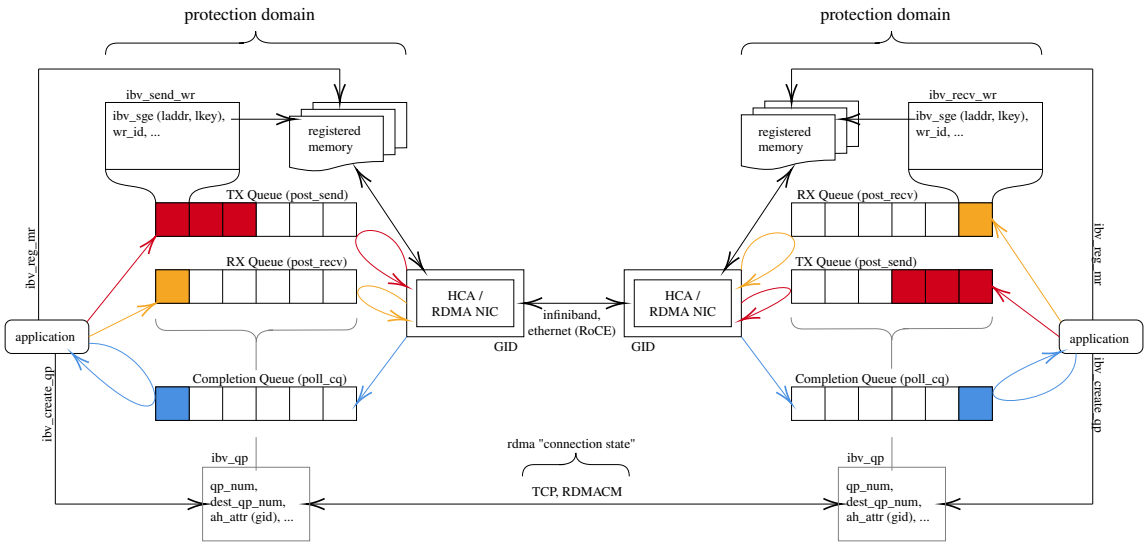

Figure 1: Diagram of RDMA's data paths

the interface is what the kernel provides on top of the NIC for standard network operations. There are four key pieces that make up the RDMA control plane: the RDMA API library and device drivers, connection state, routing information, and interface management (i.e. multiplexing).

First, user-space processes utilize RDMA through a library known as `libibverbs`, which provides a central API for RDMA use regardless of network fabric [41]. This library handles both data plane operations, such as creating work requests or polling the completion queue, and control plane operations such as device metadata queries, queue creation and instantiation, or registration of shared memory with the NIC. However, while data plane operations use the shared-memory regions as a communication channel, most control plane operations cannot do the same. For example, these operations may involve the actual creation or manipulation of the shared memory region, or may be invoked when a shared memory region does not exist. Thus, device drivers must be used as a communication channel for these operations. This is, critically, one of two times the kernel is involved in RDMA

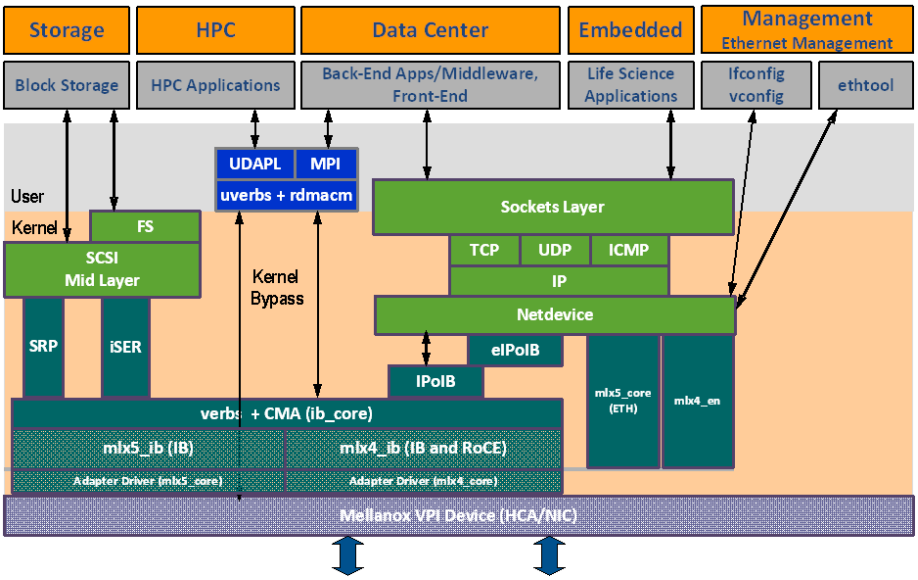

Figure 2: `libibverbs` stack
Source: [31]

operations. Alongside `libibverbs` are drivers for the NIC itself; these drivers expose character device files (usually in `/dev/infiniband` and/or `/sys/class/infiniband`) that user-space processes can interact with[5] as a control plane communication channel to the NIC [54]. Thus, all RDMA operations are initiated with `libibverbs`, which then feed into either shared memory for data plane operations or device drivers using character devices for control plane operations. Note, these control plane operations tend to be so infrequent (e.g. most applications will use long running connections and thus will initiate connections rarely), the performance penalty of kernel interposition is negligible here.

Second, RDMA maintains a notion of connection state, similar to that of TCP, stored in a structure known as `ibv_qp`. This structure contains fields such as `qp_num`, `ibv_device`, `dest_qp_num`, and `dlid` / `dgid`[6]. These are, in respective order, analogous to local port, local IP, remote port, and remote IP in a TCP connection, and tell the NIC to which machine

[5] usually with `read`, `write`, or `ioctl` syscalls, which is what results in the kernel / driver involvement

[6] there are Infiniband-specific layer-2 and layer-3 addresses respectively, however the `gid` can and is used to store MAC addresses, IPv4 addresses, and/or IPv6 addresses for Ethernet

and to which receive queue on that machine all work requests in a specific TX queue should go to. Further, this structure also contains fields such as `ibv_pd` which specify which regions of shared memory a remote host can access using a set of generated cryptographic keys.

There is the question of how some of the state in this structure — specifically `dest_qp_num`, `dlid / dgid`, pointers to remote memory, and the cryptographic keys for accessing those regions of remote memory — is populated. The RDMA protocol itself does not specify how to share this state and is thus deferred to standard networking facilities [48], this being the second time the kernel is involved in RDMA operations. The user-space process can either manually populate and share this state over a standard protocol like TCP, or use the `rdmacm` facilities (having its own library and character devices) which handles the full creation of `ibv_qp` using the most appropriate network protocol. Once this state is populated, this structure is then registered with the NIC [14, 32, 41].

Third, RDMA-capable NICs are, first and foremost, standard NICs and thus have at least minimal support for routing capabilities. On Ethernet fabrics, this means that at the hardware layer (ergo being exposed to the RDMA control plane) the RDMA-capable NIC has a MAC address and, in many cases, support for VLAN tagging. These properties are used by the now outdated RDMA protocol RoCE v1, which allowed for RDMA over a single Ethernet broadcast domain. Now, a MAC address alone does not suffice for standard networking, thus the host networking stack adds an IP layer on top of this for layer-3 routing. Normally, this IP layer exists solely in the host's software, but drivers for the RDMA-capable Ethernet NICs have added functionality to expose IP addresses and VLAN tags to the NIC, which are then stored in the NIC's GID table. These are then accessible to the RDMA

control plane and are used by the RDMA protocol RoCE v2 that encapsulates RDMA packets inside of UDP packets for IP routable RDMA. On Infiniband fabrics, the story is more straightforward. In contrast to Ethernet NICs, most of the Infiniband networking logic lives on the NIC hardware (these Infiniband NICs being formally called HCAs), thus much of the Infiniband interface information — such as its Local Identifier (LID), a layer-2 address, and Global Identifier (GID), a layer-3 address — is stored on the NIC and is accessible to the RDMA control plane [31]. However, for both Ethernet and Infiniband, next hop information (usually deduced with the subnet mask or Infiniband PathRecord) is not exposed by the kernel to the NIC. As the final piece, user-space libraries (or the `rdmacm` module) will, depending on the specific NIC implementation, populate this information using the `ibv_ah_attr` and `ibv_global_route` structures, which are nested inside of `ibv_qp` [14, 17, 32].

Support beyond these minimal routing capabilities for the RDMA control plane, such as firewalls at the host level and complex routing rules, is dependent on the features the NIC itself provides as these would have to be executed in the NIC's hardware. This results in widely varying capabilities of the RDMA control plane depending on the NIC used — a point that will be pertinent when discussing the various RDMA in container solutions. Regardless, even if the NIC does not support some RDMA control plane functionality, an option may be to enforce it at the network level if the network fabric supports it.

Finally, RDMA-capable NICs tend to support the creation of multiple interfaces on top of a single NIC using both software and hardware based multiplexing. The exact capabilities and functionality of this multiplexing is, again, dependent on what the NIC or NIC driver provides; but, in general, software based multiplexing is managed by the kernel

and integrates with the NIC using the aforementioned sharing and storage of IP addresses and VLAN tags to the NIC's GID table. When a new interface or IP address assignment is created in the kernel (using commands like `ip addr add`), this creates a new entry in the NIC's GID table, which can then be used for both RDMA connection state sharing (using the kernel's interface abstraction), and for RDMA sends and receives (by specifying this new GID in the `ibv_device` structure). The scalability of this is, however, limited as the GID table has a fixed size in the NIC and with protocols like RoCEv2 requiring two GID entries minimum per interface, GID table entry exhaustion is a real possibility.

Hardware-based multiplexing, in general, uses Single Root I/O Virtualization (SR-IOV) which allows a single NIC (called the Physical Function or PF) to expose itself as multiple PCIe devices (each called a Virtual Function or VF). These PCIe devices will appear as unique hardware network devices (as opposed to a software abstraction) to the kernel or hypervisor that can be handed out to individual virtual machines or isolated in namespaces. On Ethernet, these VFs will appear as unique interfaces to the network, each with its own MAC address and, on some NICs, can have VLAN restrictions set on them by the host VM that cannot be overridden by the guest VM. On Infiniband, the network is not made aware of each VF (formally called a vHCA), and the PF handles switching from the network fabric to each individual VF. SR-IOV provides another form of virtualization, often useful in the context of virtual machines, but does suffer from hardware constraints as it relies on available hardware resources. For example, the number of possible VFs is usually capped to under 150; use of VFs on Ethernet NICs can result in IP address exhaustion; and SR-IOV struggles with the same GID table entry exhaustion as software-based multiplexing as VFs share the same GID table as PFs, with GID table entry exhaustion resulting in the disabling

of RDMA on some VFs [31].

2.2.3 Management Plane

The management plane for RDMA, which is integrated in the management plane for the NICs themselves, is then relatively straightforward. Similar to control plane operations, character devices and device drivers are used as the communication channel for management plane operations, with libraries like `libibumad` being used to define the communication data types. User tools like `ibstat` or `ibdiag`, as well as network managers like Infiniband's Subnet Manager, will use these libraries to reconfigure NIC policies. These character devices can also be hidden from user-space processes as a method to prohibit NIC policy configuration.

Recently, RDMA-capable NIC device drivers have begun adding functionality to existing kernel systems to allow for minimal NIC and RDMA policy configuration with native OS tools like `devlink` and iproute2. This has allowed for some native compatibility with existing kernel systems, like using capabilities to control IP assignment on RDMA-capable NICs [31]. However, it still remains that the majority of the kernel's data plane and control plane systems, such as `iptables`/`nftables`/netfilter, are not exposed to the RDMA-capable NIC and are thus invalid in the context of RDMA operations.

2.3 Containers

Containers are, as mentioned previously, a Linux-based packaging and deployment framework for complex applications. Reminiscent of a unikernel, containers provide portable runtime environments, strong isolation, and fine grain access control to kernel resources for

user-space processes. These are accomplished with a combination of several Linux kernel systems.

First, for portable runtime environments. Containers are bundled as *container images*, which contains a file system[7] that holds all runtime dependencies and binaries for the application, much like a static binary. The kernel will then isolate the application process to this environment using `chroot`, or something similar like `pivot_root`, such that the application can view only the provided file system and will be restricted from viewing the host's file system [45].

Second, for strong isolation. The Linux kernel provides *namespace* functionality that allows for isolating a given process's view of kernel resources. For example, if a process is placed in its own PID namespace, the kernel provided `procfs` will return, to that process, information such that the process appears to be running as PID 1 and will see only it and its children in the process tree[8]; therefore preventing the PID-namespaced process from viewing any other processes that may be running on the host. Similarly, other namespaces like the network namespace will return to the process an isolated view of available network resources. In total, the Linux kernel provides namespaces for PIDs, IPCs, Cgroups, IPCs, mounts, users, time, and hostnames (through the UTS namespace) [11, 28].

Third, for fine grain access control. In conjunction with namespaces that can act as a rudimentary form of access control to kernel resources, the kernel provides *cgroups* for resource usage limiting and monitoring; as well as *capabilities* and *seccomp* for restricting kernel operations and syscalls. Cgroups are configurable controllers in the kernel that can

[7] usually a union file system to allow for deduplication across multiple container images

[8] internally, the kernel maintains a map of PIDs in process namespaces to PIDs in the root namespace for resolving the "true" PID of a process for operations like permission checks

monitor and limit per process usage of CPU time, memory, and IO usage. Capabilities are a form of permission control on groups of kernel operations for processes. For example, removing the `CAP_NET_ADMIN` capability from a process will restrict it from all network configuration operations and `CAP_IPC_LOCK` will restrict all memory locking operations. Seccomp is like capabilities, but addresses individual syscalls as opposed to groups of operations, thus providing more fine grain control [4, 11].

As these are all systems within the kernel, this provides an extremely low overhead and fast to reconfigure form of isolation, as compared to say virtual machines. Hundreds of containers can feasibly run on a single system with no more overhead than that of running hundreds of processes, and these containers can be deployed at speeds similar to that of launching a new process on Linux [39].

2.3.1 Container Networking

Up to this point, the systems that have been described provide intra-machine isolation in that they can isolate multiple containers running on a single host. If a container utilizes networking, there is then the question of inter-machine isolation and access control, such as how to restrict to what hosts a container can initiate network connections with. These are accomplished with Linux's netfilter system (for which `iptables` or `nftables` are the front end) and general network stack (for which the iproute2 collection of tools is the front-end).

First, netfilter provides a framework for defining network rules based on packet metadata. When a packet arrives or leaves an interface, the kernel will iterate over the defined netfilter rules to determine how to manipulate or restrict that packet — for example, to act as a host-level firewall or as a NAT. The kernel also provides hooks between netfilter and cgroups

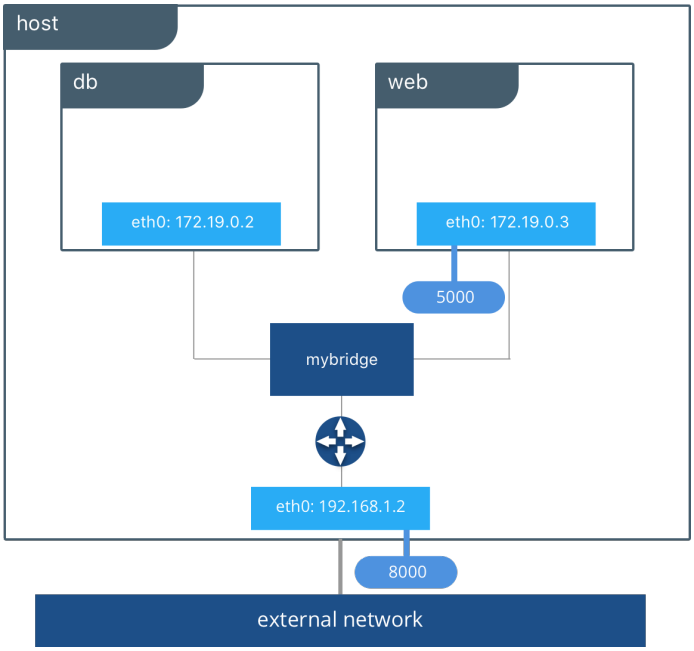

Figure 3: Typical Container Network
Source: [5]

to allow netfilter to tag all packets originating from a container, where these tags can then be used in netfilter rules. Second, the Linux network stack has several highly expressive systems for interface management, routing tables, and traffic control. These tools can be used to create virtual interfaces, virtual bridges, advanced routing rules (often in conjunction with netfilter rules), and traffic shaping policies on specific network interfaces [19, 29].

Together, netfilter and Linux's expressive network stack are used to create controllable, highly expressive, and isolated virtual networks for containers. Virtual interfaces will typically be created for individual containers that are then added to that container's network namespace. Outside of the container, that interface will often be attached to virtual bridge interfaces, as well as have netfilter, traffic control, and routing rules put in place (e.g. NAT on outgoing traffic), to control where to route and how to shape network traffic to or from the container. This allows for a wide expression of network topologies, has the benefit

of being fast to configure with these all being software systems, and allows containers to be relocatable in that they can be moved to other hosts without any disruptions in service availability. This also ensures that the container can be restricted from viewing the host's actual network, and vice versa where the container can be hidden from the host's network.

This should highlight why container networking is not runtime performant. Every packet to or from a container must traverse the Linux network stack and netfilter system at least once for every interface, virtual or not, along the container's network path. For example, if a container has a virtual interface, that is then connected to a virtual bridge, which is then connected to the interface for the NIC; that container's network traffic must traverse the Linux network stack a minimum of three times. With the network stack being primarily implemented in software, meaning both the control plane and data plane are in software, the performance costs explained in Section 2.1 can add up.

Putting it together, this, alongside the intra-machine isolation described in Section 2.3, provides an immense amount of power for container deployment. These systems are extremely quick to configure, highly expressive, and are relatively low overhead[9]. Further, container engines are typically used for the actual deployment of containers, such as Docker or podman, that handle the full configuration and monitoring of the aforementioned kernel systems, and use standards for both container images and schemas to describe container deployment. This makes container deployments simple, consistent across machines, and scalable, which explains why containers have become so widely adopted and why container orchestration tools like Kubernetes have excelled in industry adoption. As a final note, it

[9]for most users, other factors like processing time or multiple router hops overshadow the performance cost of container networks, thus container networking is usually considered low overhead

is critical to once again emphasize that these are software systems that run inside of the kernel, and thus can only influence operations whose data path cross the kernel.

2.3.2 RDMA Namespaces and Cgroups

With namespaces and cgroups being the standard systems for isolating Linux kernel resources in containers, there have been developments to add RDMA support for these to the Linux kernel itself. The `rdma-system` on Linux integrates with the existing network namespace system to allows for restricting RDMA devices (e.g. the `mlnx4_0` passed to `ib_dev` in most CLIs) from both an `libibverbs` and `librdmacm` view to an existing network namespace; this RDMA device being a physical function, virtual function, or virtualized RDMA NIC (e.g. using SoftRoCE). Further, when an interface (defined by the kernel) for an RDMA device is added to a network namespace, the GID table entries are also added to that namespace, in other words, if a process has a network namespace, it can only see the GID table entries for the interfaces in its network namespace. Next hop resolution is also performed inside of that network namespace [10].

The RDMA cgroup currently allows for tracking and restricting the number of RDMA control plane resource used (e.g. QPs and shared memory regions), and tracking counters exposed by the RDMA device. Neither of these systems currently integrate with existing kernel network isolation systems like netfilter and do not provide any form of data plane control (e.g. restricting the number of work requests), thus on their own, they cannot provide the network virtualization desired for containers. As such, they currently only provide intra-machine isolation but not inter-machine isolation for RDMA devices [35, 36, 37, 40, 42].

2.4 Related Work

2.4.1 Sharing RDMA NICs Among Virtual Machines

There's a significant overlap in the research of sharing RDMA NICs among containers and among virtual machines, with many of the to-be-explained solutions being compatible with both. However, there are some key differences when assessing RDMA NIC sharing for containers: (1) containers are often ephemeral, thus resource deployment and cleanup must be fast and low overhead, (2) container engines use kernel functionality for isolation, meaning sharing solutions must expose themselves to the kernel in some way, (3) a single machine could run tens to potentially hundreds of containers, as opposed to a handful of virtual machines, thus scalability is more of a concern, and (4) containers use a vastly different configuration scheme than virtual machines. Due to these differences, solutions like HyV, virtio-RDMA, and vRDMA are incompatible for use in containers and the remaining solutions must be assessed differently for containers than for virtual machines [12, 38, 53].

2.4.2 Programmable NICs

Independent of RDMA, there is a large field of research around programmable NICs (often FPGA based) as a method of speeding up networking. Network rules, like firewalls, would be programmed into the NIC as a form of hardware offload, as opposed to executing on the CPU through the kernel. This could be used to express RDMA network rules with minimal performance impact as these rules would run on the NIC itself, therefore allowing for highly-performant container RDMA network isolation. This too may be part of the future for RDMA device management in containers, but as of now, does not seem mature

enough for container use cases or for general use as: (1) some of these NICs are not available to the public (e.g. Azure's programmable NICs), (2) many of the available programmable NICs are slow or expensive to configure, (3) some NICs exhibit pathological performance characteristics, which is detrimental for RDMA use, and (4) these NICs do not integrate with existing kernel network isolation systems (e.g. netfilter), thus container engines would have to create complex shim layers to utilize these programmable NICs [13, 21, 23, 26, 47].

2.4.3 eBPF Container Networking

Many groups have identified netfilter as a massive performance bottleneck for container networking, primarily due to its sequential nature. Efforts have begun to replace netfilter with eBPF, a system for executing user-space provided code in kernel-space, which would simplify the code paths taken in the kernel for container networking. While this would alleviate many of the performance impacts of container networking, it would not be a panacea for high performance networking in containers [9, 56]. Even with eBPF, container networking would still have to traverse the Linux network stack several times, cross the NIC-CPU boundary for every packet, and cross the user-space/kernel-space boundary to receive and deliver packets; therefore retaining much of the overhead explained in Section 2.1. While this may be a solution for those needing faster container networking, RDMA in containers will still be needed for those who want highly performant networking.

2.4.4 DPDK in Containers

The Data Plane Development Kit (DPDK) is another, popular form of kernel bypass networking. Instead of relying on the kernel's networking stack, a user-space process will

communicate directly with the NIC and run its own, custom network stack. This can be used as a form of high performance networking in containers, but does come with its own unique challenges and research questions. For example, DPDK often requires heavy CPU utilization, also cannot utilize in-kernel isolation systems, creates more difficulty with network isolation as it has fewer boundaries, and requires privileged containers which may not be desired [16, 30, 33]. As such, it remains an orthogonal solution to high performance networking in containers.

3 Experimental Comparisons

3.1 Paravirtualized Solutions

Paravirtualized solutions are those that interpose in some part of the RDMA logic path while keep the rest of the logic path in tact. Typically, the interposition is done in a part of the logic path that is not hot or performance sensitive (e.g. control plane operations) to add additional controllability while maintaining the performance characteristics of RDMA. These solutions tend to require custom libraries, add more code complexity, may require heavier CPU utilization, and require frequent updates to stay in line with the underlying RDMA libraries. However, they tend to provide close to host-level RDMA performance, tend to scale better than hardware solutions, and can more closely integrate with existing kernel network isolation systems.

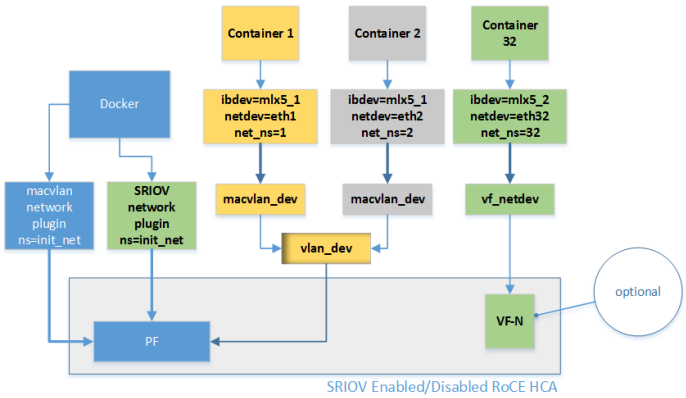

Figure 4: Shared HCA Architecture
Source: [50]

3.1.1 Shared HCAs

Though perhaps not strictly paravirtual given this solution is now built into the RDMA logic path in the kernel, shared HCA operates similarly to paravirtualized solutions in that it interposes namespace and cgroup restrictions in the RDMA control path. This is arguable the most "batteries not required" solution of using RDMA in containers as it uses the built in kernel isolation systems, but is still rudimentary is its expressivity of control. This solution creates new interfaces (with or without VLAN tags) on top of an RDMA device, then places this interface in the container's network namespace. As explained in Section 2.3.2, this also places that interface's corresponding GID table entries in the namespace. Paired with restricting the `CAP_NET_ADMIN` capability for the container, this can then be used to give a container an isolated RDMA interface with a set MAC address, IP address, possible VLAN tag, and restrictions on RDMA resource usage using cgroups. However, as the RDMA device is not made aware of the host's virtual networks, the interface's properties must be valid on the host's network (e.g. the interface must have a valid IP address on the host's actual network) [50, 55]. This solution is also typically called macvlan mode.

Container Network Properties

Network Isolation the only isolation provided is the setting of the IP address and VLAN tags on an isolated interface, the IP address is also dependent on the NIC

Controllability routing policies and traffic shaping capabilities are not possible unless implemented in the device

Resource Utilization aside from GID table entry consumption, resource utilization is identical to normal RDMA usage

Performance Tests

All of the following tests were performed on Cloudlab with two d6515 hosts, which have a 32 core AMD 7452 at 2.35GHz CPU, 128GB RAM, and a dual port ConnectX–5 100 Gbit NIC [8]. Both were running Ubuntu 20.04 with Linux kernel version 5.4.0, and were using the Mellanox OFED RDMA stack version 5.5-1.0.3.2.

Figures 5, 6, and 7 were all performed with either a single shared HCA device (labeled `shared`) or a single raw RDMA device (labeled `host`), with varying RDMA message sizes. The x-axis represents the size of each RDMA message in number of bytes. All three of these graphs indicate there is no perceivable difference between using the single raw RDMA device vs using a shared HCA device.

Figure 8 was performed with multiple shared HCA devices created on both the client and server, with a single `ib_(read|write|send)_bw` test running on each device with a byte size of 65536. Individual devices on each hosts were connected pair-wise for these tests. For example, with `pairs` $= 3$, both the client and server had 3 shared HCA interfaces

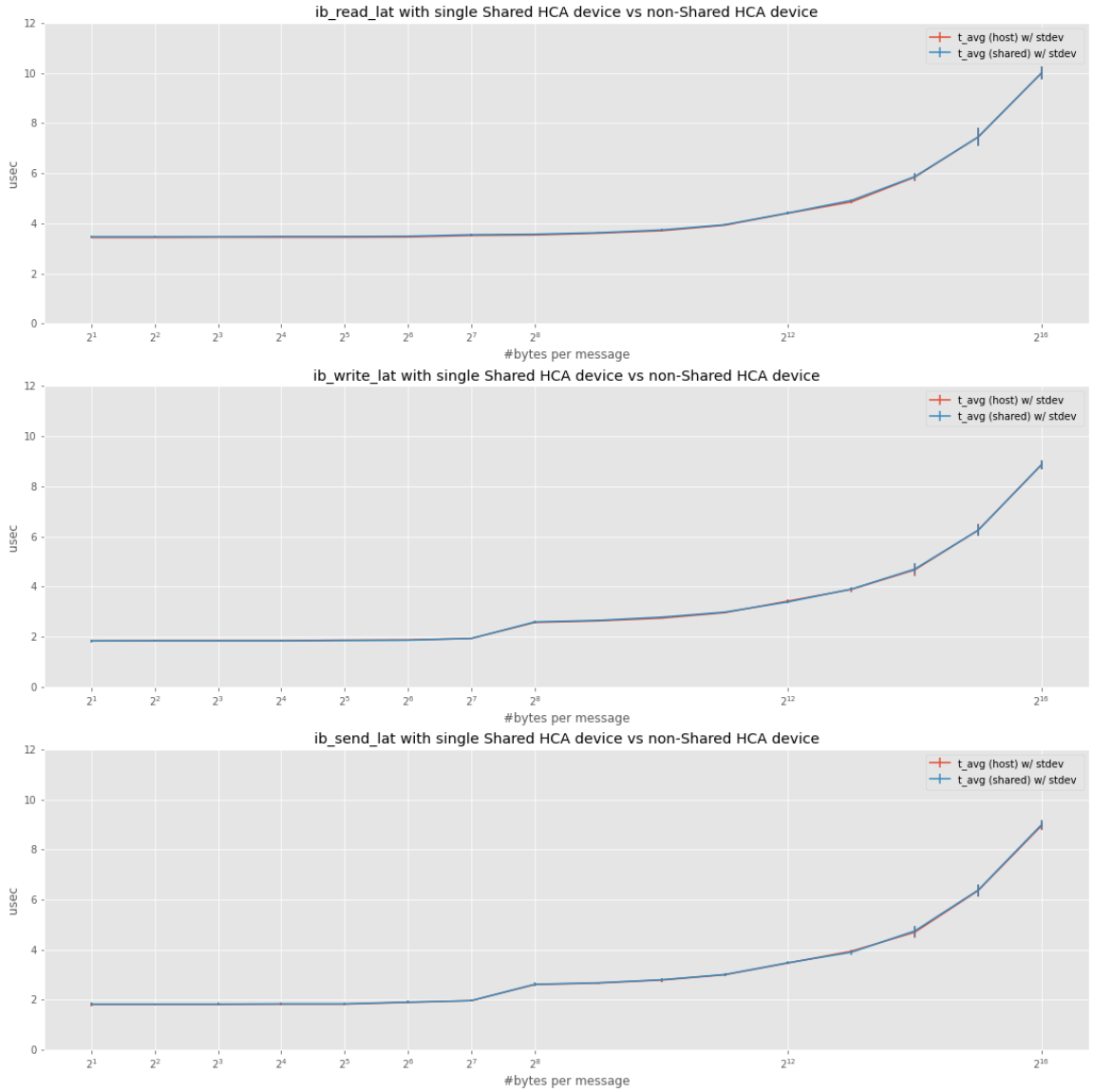

Figure 5: Latency Tests

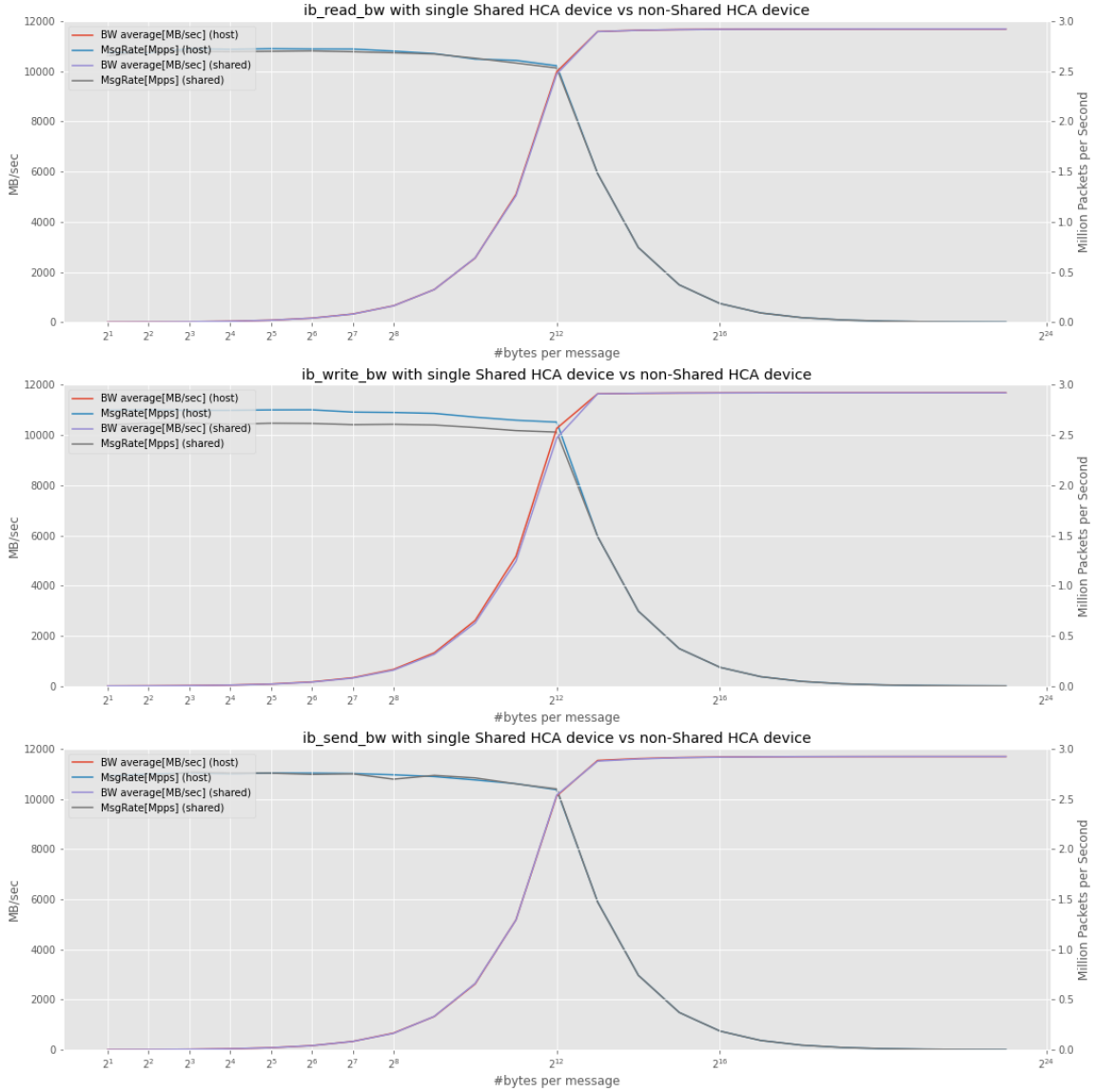

Figure 6: Bandwidth Tests

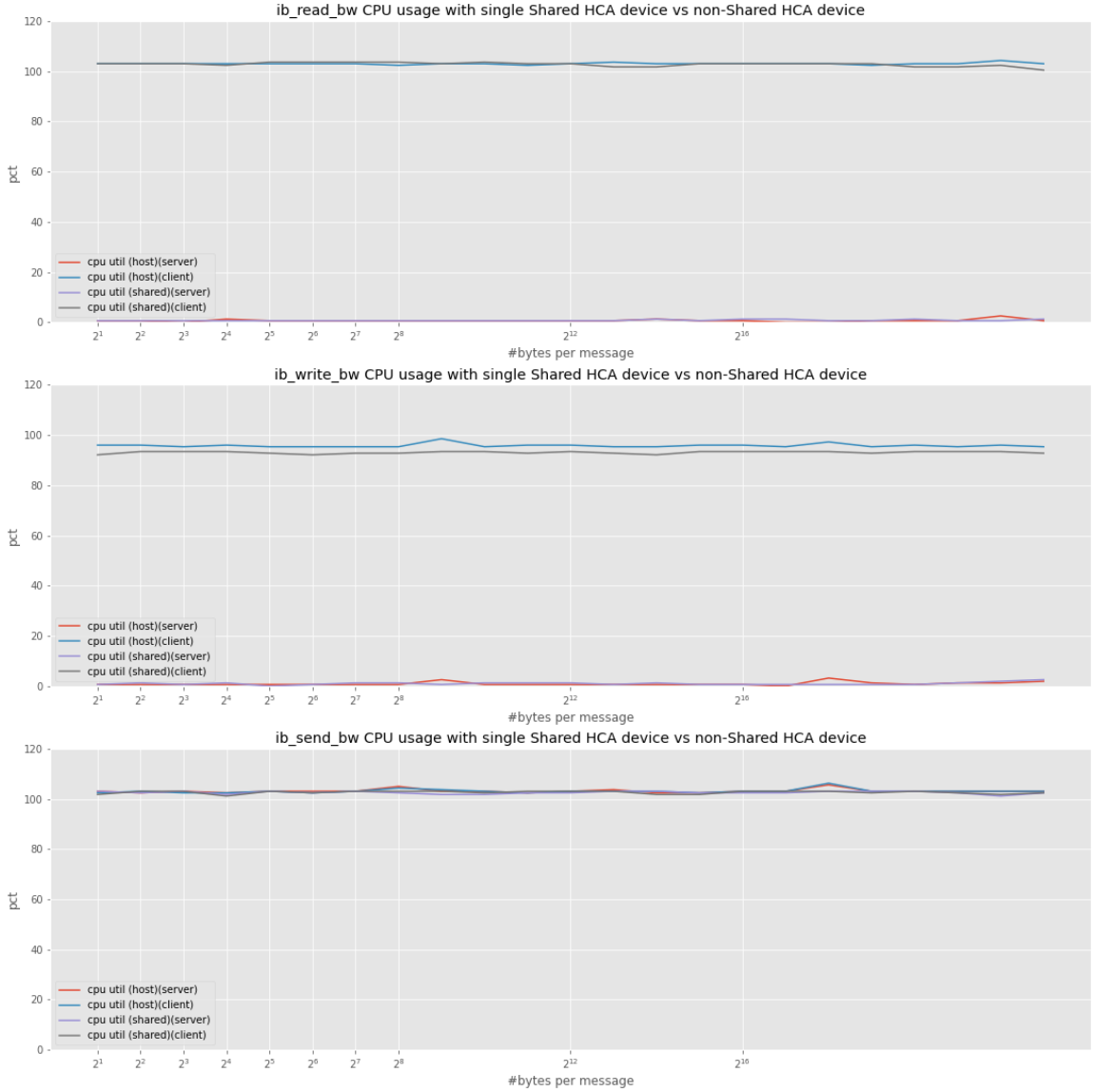

Figure 7: CPU Utilization

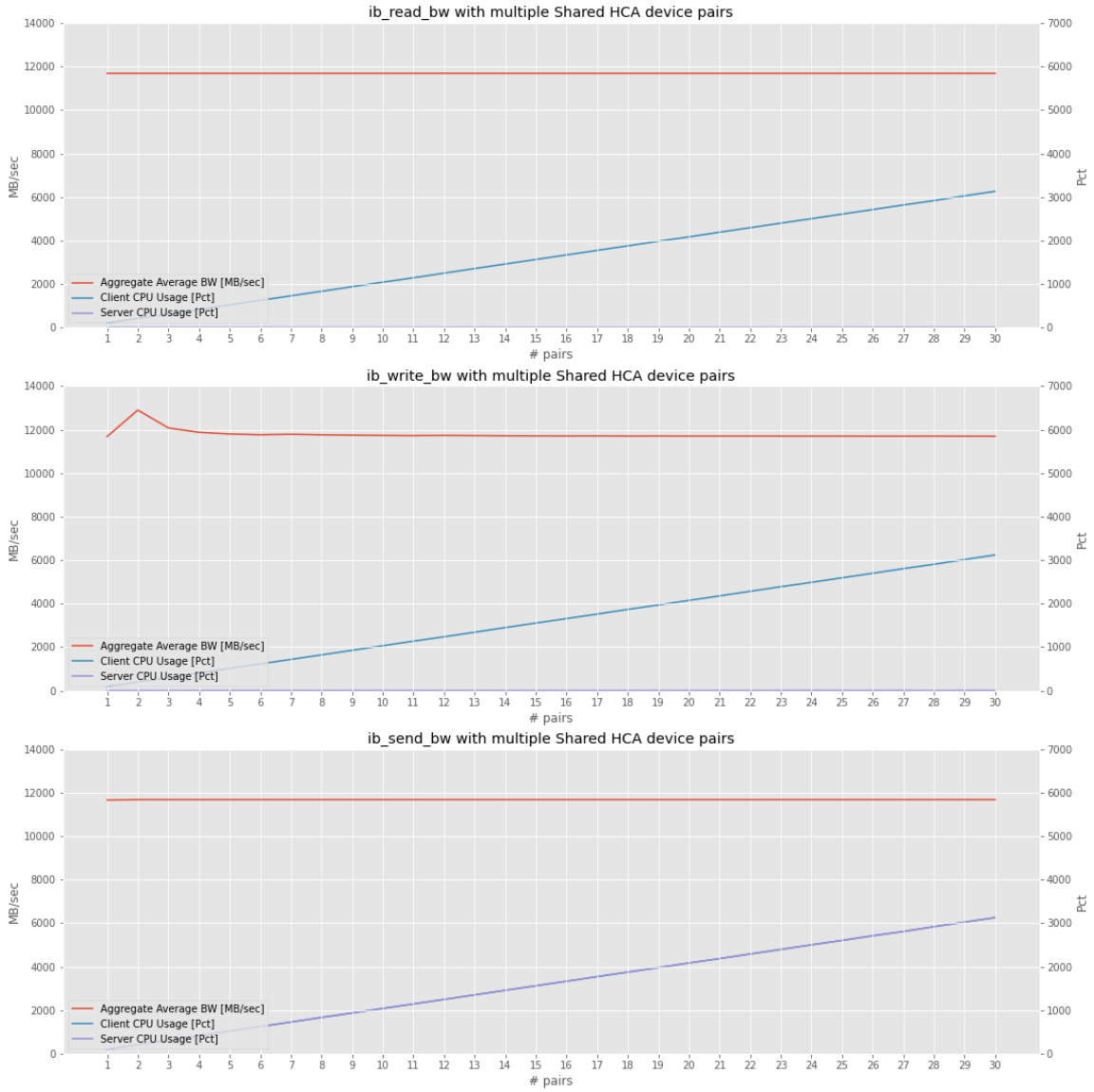

Figure 8: Bandwidth + CPU with Multiple Devices

/ devices provisioned. Shared HCA device 1 on the client connected to the `ib_read_bw` server running on shared HCA device 1 on the server. Similarly, shared HCA device 2 on the client connected to the `ib_read_bw` server running on shared HCA device 2 on the server, and so on.

Each of these devices were "provisioned" by Docker and the tests ran inside of the corresponding Docker container. Similarly to the individual device tests, the multi-shared HCA device tests show that shared HCA can fully saturate the bandwidth of the host RDMA device with no additional CPU utilization (the CPU utilization is expected with the `ib_(read|write|send)_bw` clients processes using 100% CPU). As expected, the use of shared HCA devices requires no performance or resource penalties.

Miscellaneous Comments

Scalability Limits can only scale to the size of the NIC's GID table, usually below 150 entries

Proprietary these features are built into the kernel's RDMA stack thus are available to everyone

Maturity have been in development since 2015, so is likely stable enough for use, but is still missing several controllability features; is built into the kernel so likely will be actively maintained

Ease in Deployment uses standard kernel isolation systems, thus is similar in ease of deployment as containers

Execution Privileges requires no additional privileges

Network Pressure requires a unique IP address per interface, thus can result in IP

address exhaustion or network switch table exhaustion

For those that don't need virtual network control for RDMA, do not anticipate running hundreds of RDMA containers, and do not require fine grain traffic shaping of RDMA packets (or are fine relying on the network or global NIC policies for it), this would be an excellent solution. However, unless unique IP addresses for containers are required (e.g. for network level routing policies), it is likely more beneficial to directly expose RDMA devices to containers.

3.1.2 FreeFlow

FreeFlow operates in a far more paravirtualized style with the use of a user-space library and controller interposing in the RDMA control paths. It claims to provide close to bare-metal RDMA performance, while adding significantly more controllability to RDMA operations, thereby allow for performant use inside of containers. It does this with two key features: (1) communications to RDMA character devices (i.e. `uverbs0`) and consumption of work requests is instead routed to and processed by a controller (FreeFlow Router), and (2) the data path for *payloads* are kept in-tact using shared memory; these being implemented in two pieces of software: the FreeFlow Library (FFL) and FreeFlow Router (FFR).

Breaking this down, containers that use RDMA devices must use a modified form of the `libibverbs` library called the FreeFlow Library. This library forwards all `libibverbs` operations through an IPC channel, instead of the normal RDMA character devices, to a running FreeFlow Router process. This results in all of the RDMA control plane and most of the data plane flowing through FreeFlow, with the most critical parts of the data plane being kept intact. This is implemented as the following:

1. The FFR maintains virtual RDMA device information per container, for which it returns information about to the container when the container uses `libibverbs` operations like `ibv_query_device`.
2. The FFR maintains the true RDMA queue pairs for those virtual RDMA devices. This allows it to rewrite addressing information (such as dgid) when the container creates new queue-pairs.
3. On shared memory registration, the container (through the FFL) will shared those memory regions with the FFR, which then shares it with the NIC. Recall this shared memory region is where the RDMA message payloads live, thus this allows for one-sided and two-sided RDMA operations and keeps the hottest part of the RDMA data path in tact.
4. A container will post work requests through the IPC channel, which the FFR then processes and can apply traffic shaping rules to. Alternatively, the container can create "fake" queue-pairs in a region of memory shared with the FFR, on which the FFR will spin-loop on. When it reads a new work request, the FFR will process it with traffic shaping rules and copy it to the true queue pairs. This alternative mode, called FastPath, allows for lower latency, but requires the use of an entire CPU core.

This clever interposition in the least performance critical parts of RDMA allow for significantly more isolation and controllability, and allows for integrations with existing network isolation systems [22].

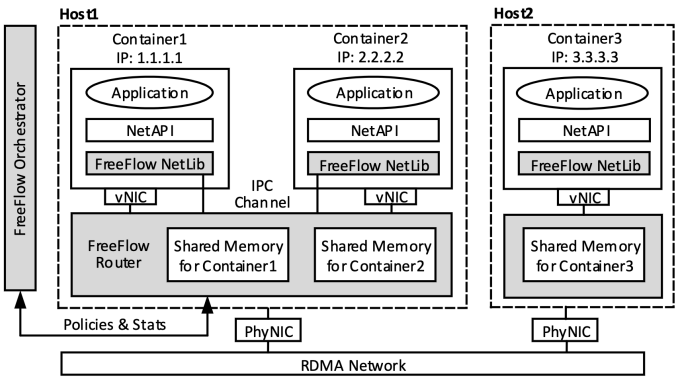

Figure 9: FreeFlow Architecture
Source: [22]

Container Network Properties

Network Isolation each container is given its own virtual RDMA device, "namespace" in the FreeFlow Router, and is not exposed to the NIC's actual network, thus allowing for the same level of isolation as standard container networks

Controllability allows for full control over the routing and traffic shaping of RDMA messages

Resource Utilization can require the use of an entire CPU core (or more) for low latency needs, otherwise uses no more overhead than one additional running process

Performance Tests

All of the following tests were performed on Cloudlab with two c6220 hosts, which have a 2, 8 core Intel Xeon E5-2650v2 at 2.6GHz CPUs, 64GB RAM, and a single port ConnectX–2 FDR (40 Gbit) Infiniband NIC [8]. Both were running Ubuntu 20.04 with Linux kernel version 5.4.0, and were using the Mellanox OFED RDMA stack version 4.9-4.1.7.0. The FreeFlow stack used is a fork (developed for this thesis) of the official Microsoft FreeFlow repository, having been updating from the MLNX OFED 4.0 stack to the 4.9 stack. This fork

also includes a handful of patches to fix FreeFlow's rkey mappings and the build process.

FreeFlow provides two modes. The first, fastpath, polls for completed work requests on the client side to provide improved latency results at the cost of a fully pinned CPU core. The second, no fastpath, waits for messages from the unix socket shared with the FreeFlow router, which reduces CPU utilization but increases latency. Both are tested and their results are reported here. Unfortunately, it appears no fastpath suffers from deadlocks with specific RDMA message sizes and with multiple FreeFlow clients, thus its results are more limited than the fastpath version.

Figures 10, 11, and 12 were all performed with either a single FreeFlow device (labeled `freeflow`) or a single raw RDMA device (labeled `host`), with varying RDMA message sizes. Figure 10 indicates that, even with the client and FreeFlow router spinning on the completion queue, there is still a significant latency overhead to using FreeFlow. Figure 11 shows that FreeFlow is capable of fully saturating the NIC's bandwidth, albeit requiring almost 4 times the RDMA message sizes than when using the raw host RDMA device. This appears to be from FreeFlow's inability to handle large message rates, which lags considerable behind the message rate achievable when using the raw host RDMA device. As this is primarily a CPU bound limitation, FreeFlow may see significantly better bandwidth performance on higher clock rate CPUs. Figure 12 shows FreeFlow using double the CPU on both the server and client when compared to using the raw host RDMA device. This is likely due to the use of fastpath, which pins the FreeFlow router at 100% CPU utilization on both the client and server. Aside from this, FreeFlow doesn't appear to require any additional CPU utilization.

Figures 13, 14, and 15 were all performed with either a single FreeFlow device in

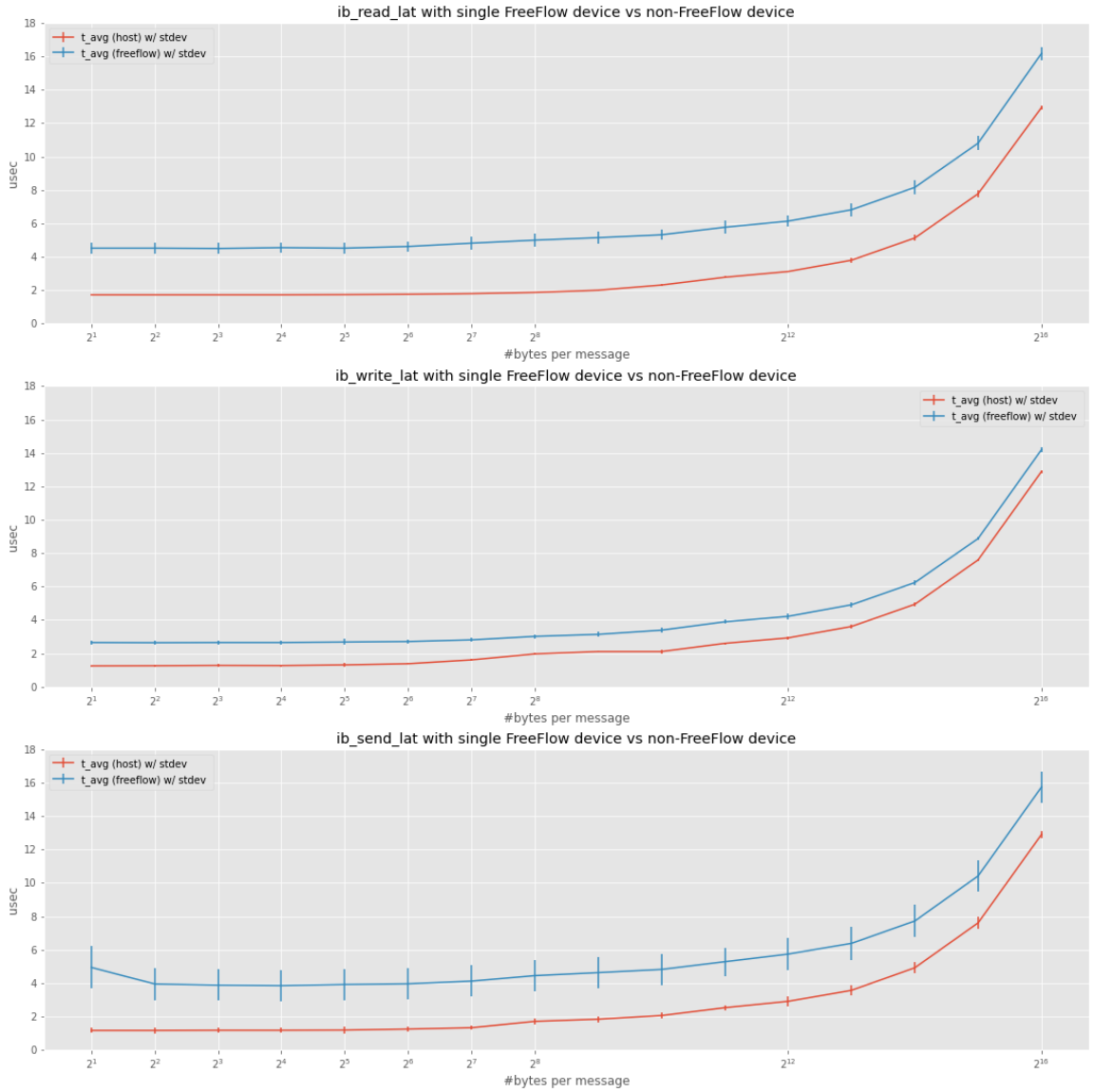

Figure 10: Fastpath Latency Tests

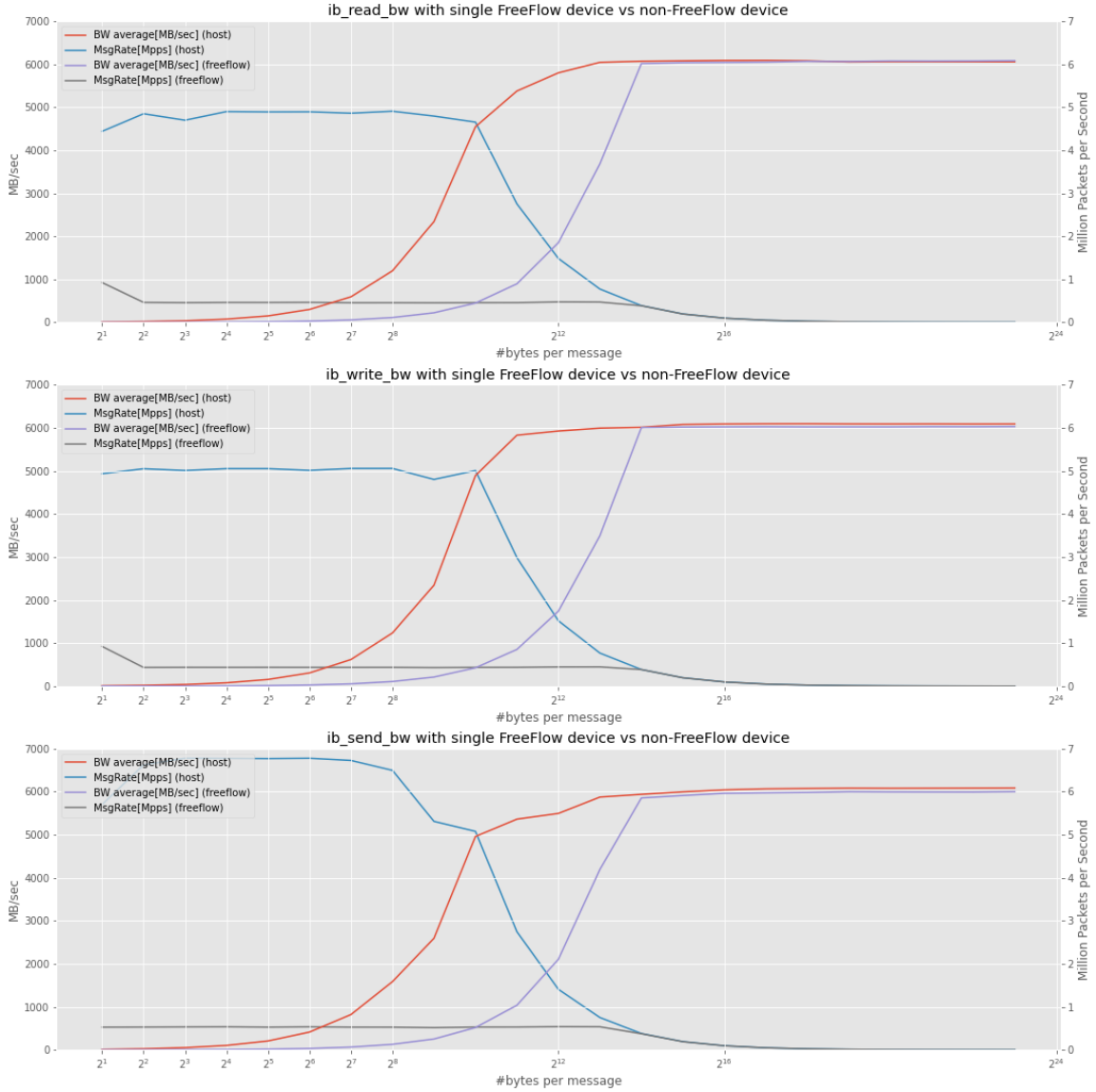

Figure 11: Fastpath Bandwidth Tests

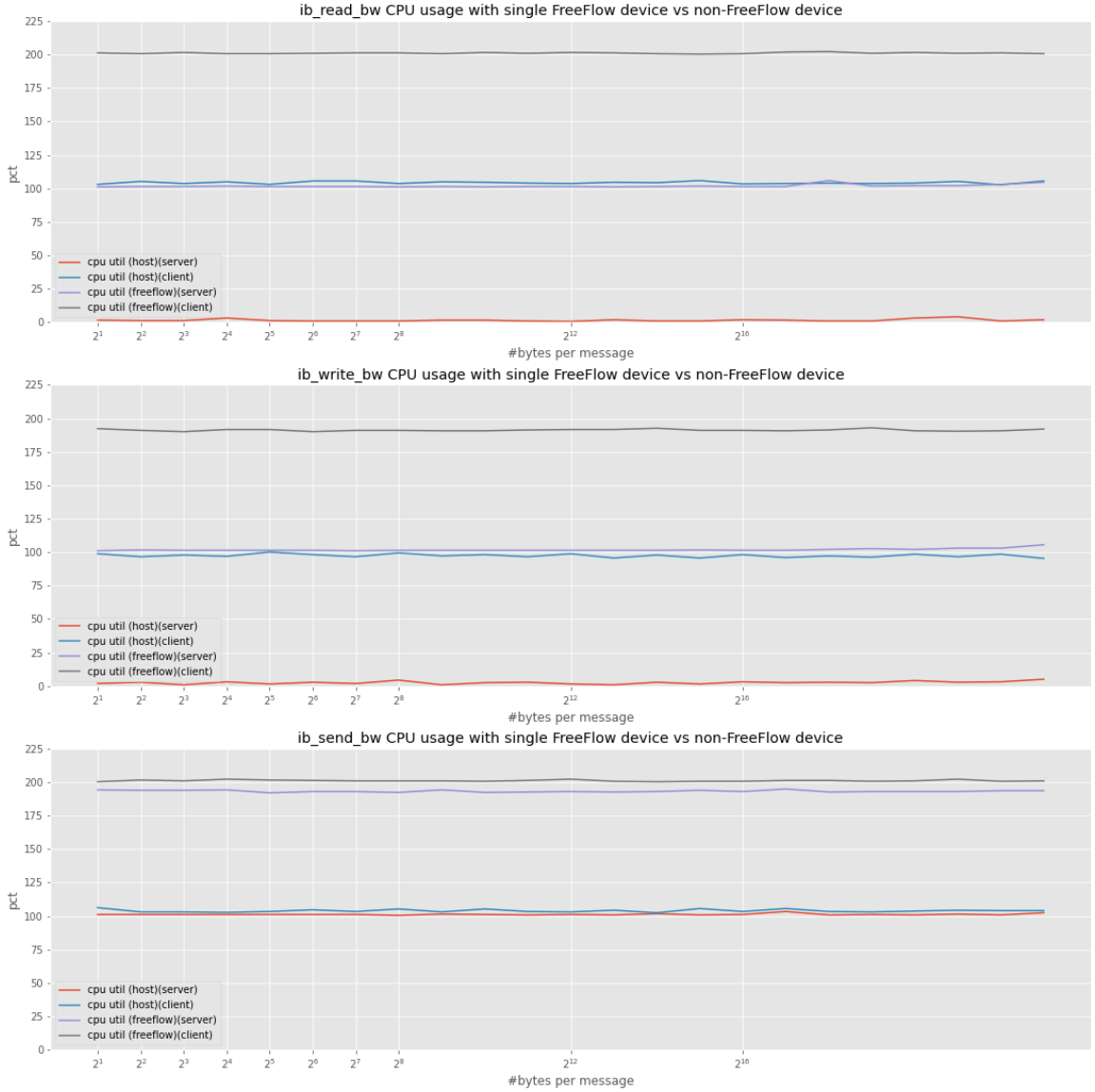

Figure 12: Fastpath CPU Utilization

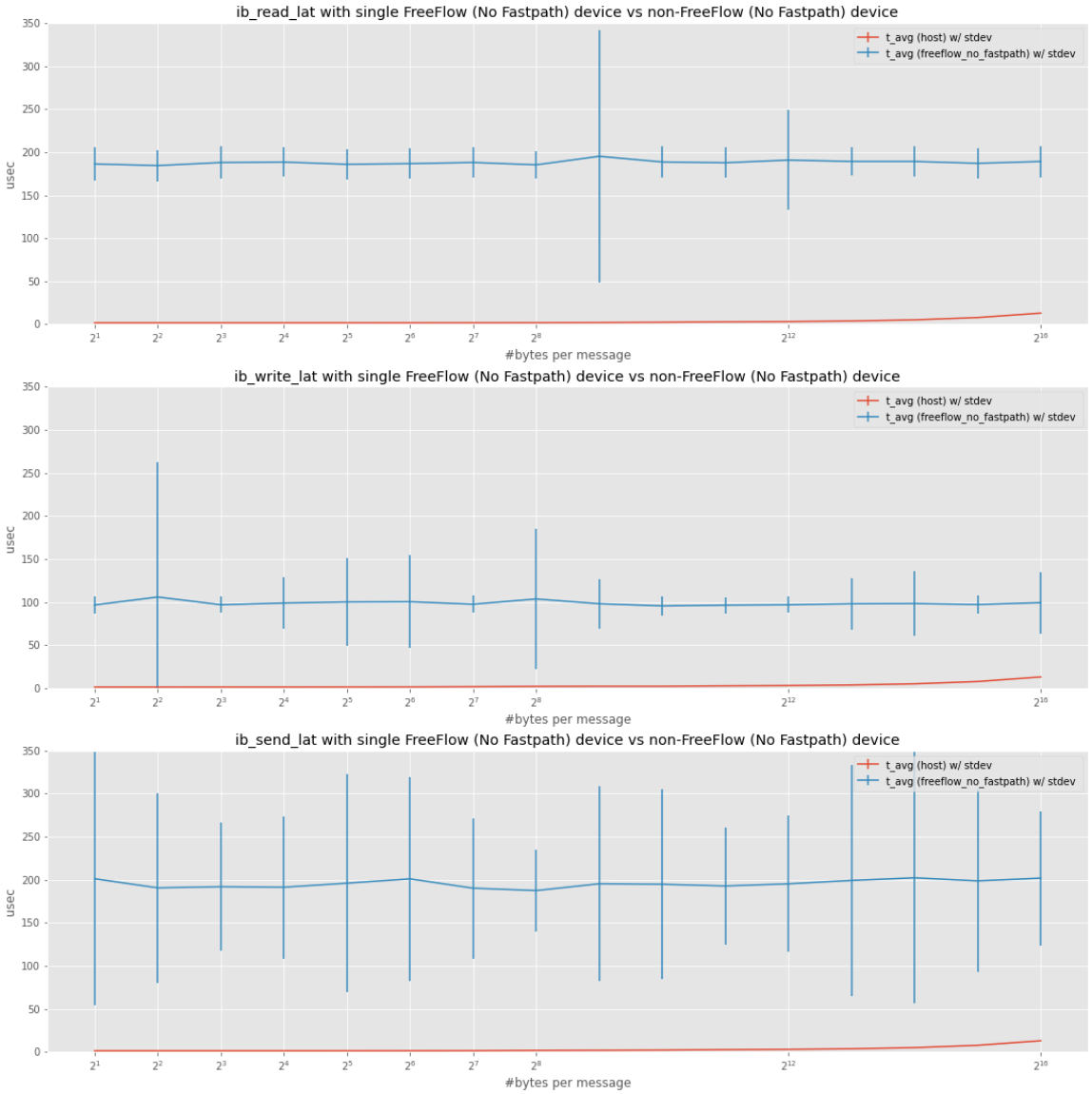

Figure 13: No Fastpath Latency Tests

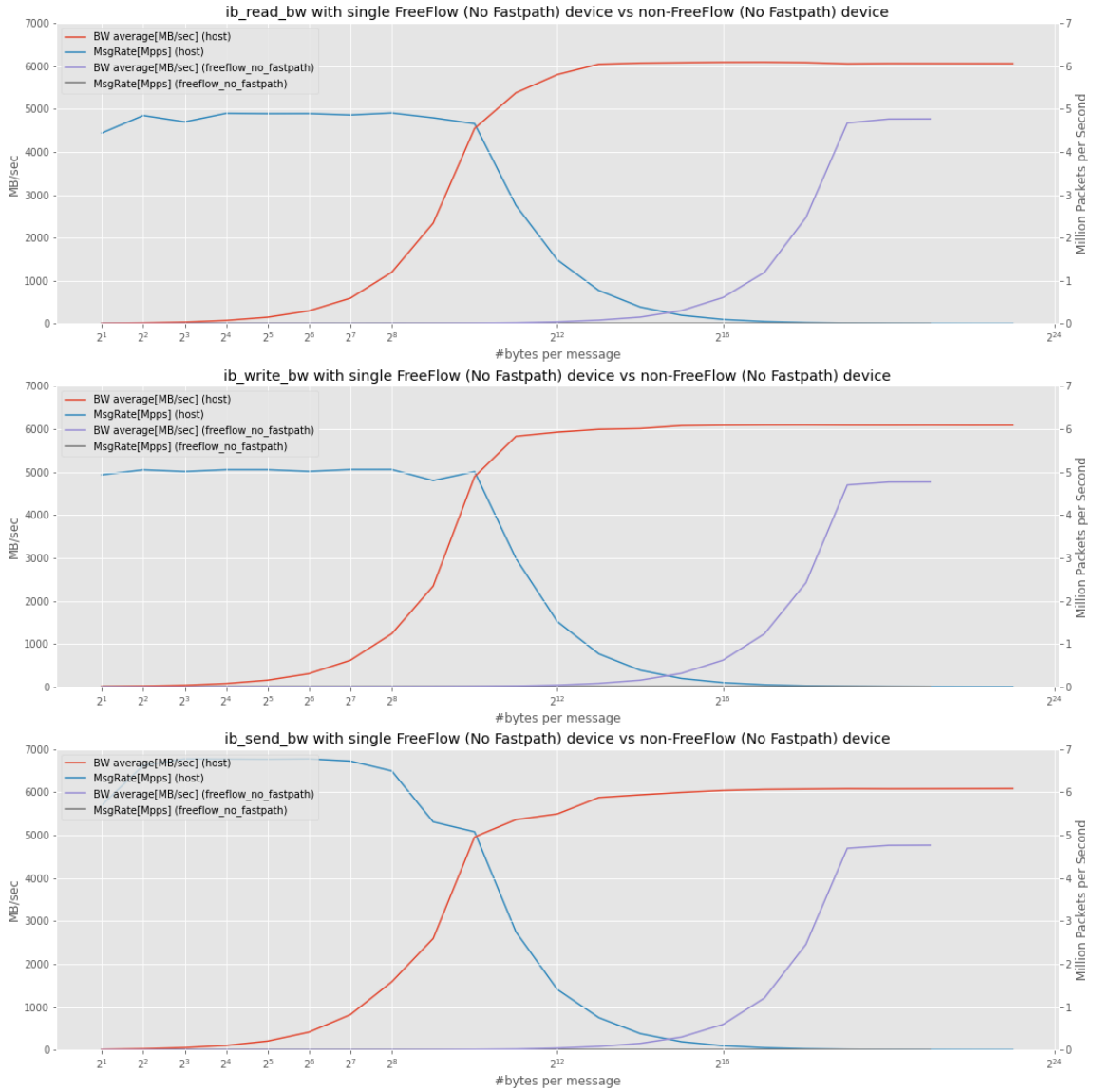

Figure 14: No Fastpath Bandwidth Tests

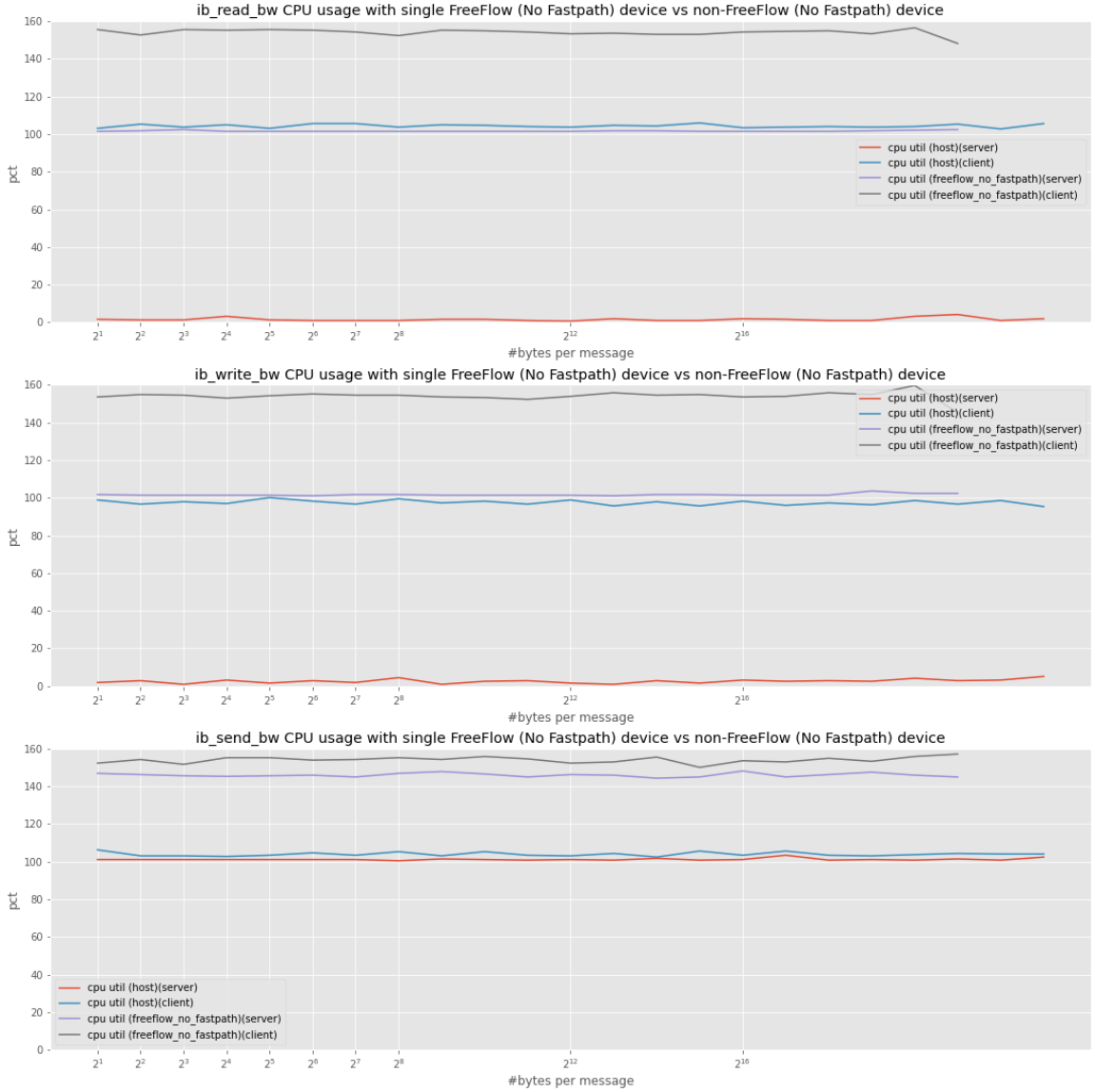

Figure 15: No Fastpath CPU Utilization

non-fastpath mode (labeled `freeflow`) or a single raw RDMA device (labeled `host`), with varying RDMA message sizes. Figure 13 shows a extreme increase in the latency overhead when compared to using FreeFlow with fastpath. This confirms, as mentioned in the paper, that non-fastpath mode is unacceptable for any latency sensitive purposes. Figure 14 indicates that non-fastpath mode also suffers substantially in bandwidth; it is no longer capable of fully saturing the NIC's bandwidth like FreeFlow with fastpath was able to. This is likely due to the even worse message rate handling with the non-fastpath mode, as the message rate is bottle necked by the notifications sent from the FreeFlow router to the FreeFlow client through the unix socket. Data past 2097152 bytes is not reported for FreeFlow without fastpath as the FreeFlow router appears to locks up after this message size in bandwidth tests. Figure 15 indicates that even with the FreeFlow router not spinning on completion queues, it still uses a noticeable amount of CPU. While it is not using 100% of a CPU core like the fastpath version does, it does appear to incur around a 60% CPU utilization overhead on the client.

Figure 16 was performed with multiple SoftRoCE devices created on both the client and server, with a single `ib_(read|write|send)_bw` test running on each device with a byte size of 65536. Interestingly, this is perhaps the most compelling performance picture of FreeFlow. At 65536 bytes, FreeFlow is able to adequately saturate the host NIC's bandwidth for a single FreeFlow device. This remains the case with each additional FreeFlow device added, suggesting the FreeFlow router is capable of juggling multiple client queues with no bandwidth performance impact. This test was not done for the non-fastpath mode as that mode appears to lock up with multiple connected FreeFlow clients.

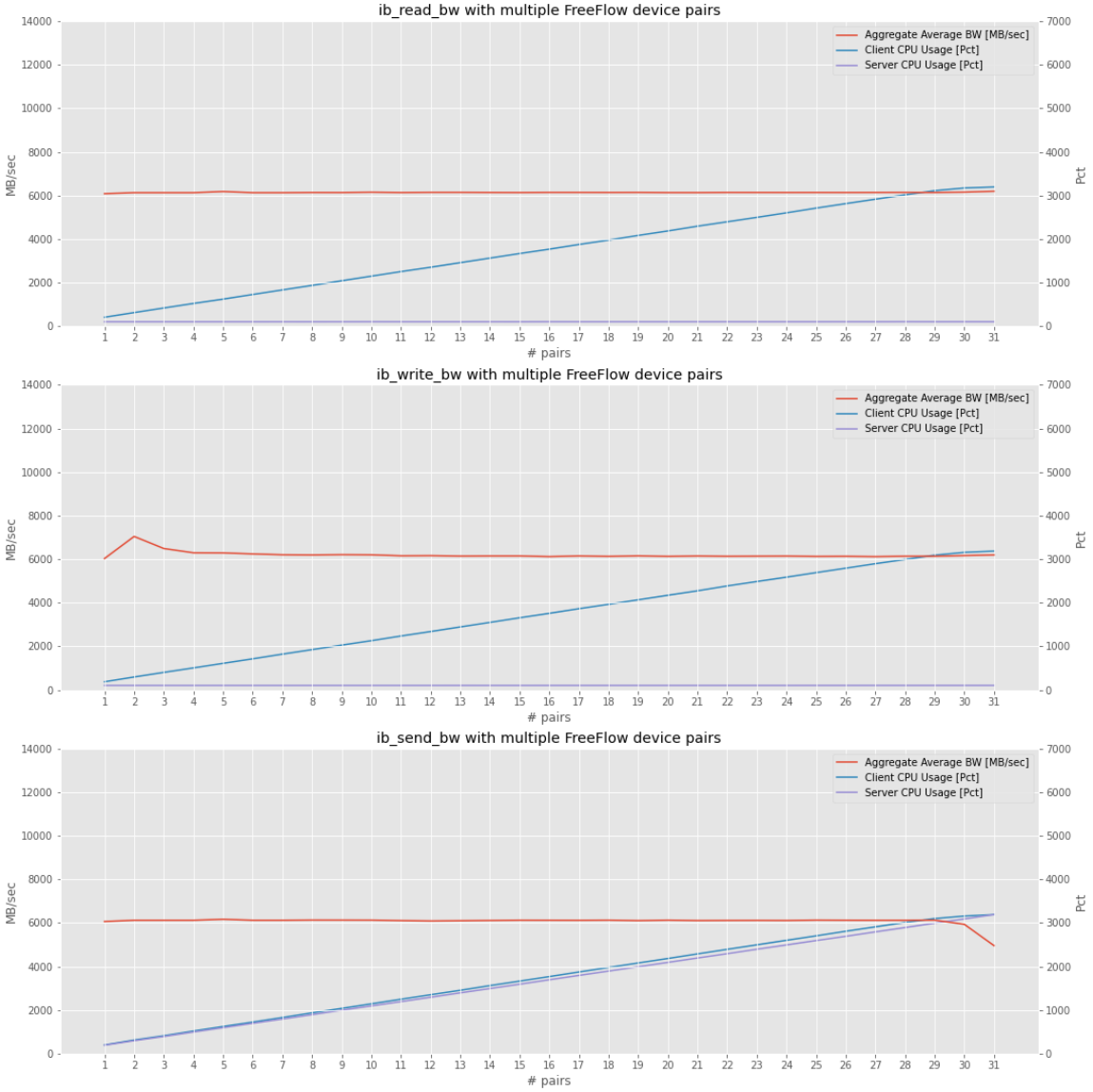

Figure 16: Bandwidth + CPU with Multiple Devices

Miscellaneous Comments

Scalability Limits can feasibly scale to hundreds of containers as it is constrained only by the available hardware on the host, not the NIC

Proprietary FreeFlow is available to use on GitHub and uses the standard `libibverbs` library

Maturity has not been maintained since publication and relies on an older `libibverbs` version, thus is not mature enough for general use

Ease in Deployment it does not currently plug into existing container orchestration tools and requires patching headers for us, thus is not easy to deploy

Execution Privileges requires the FreeFlow Router to be a privileged container (if it runs in one), but client containers do not have to be privileged

Network Pressure virtual FreeFlow RDMA devices are not exposed to the network, thus creates no additional network pressure

While FreeFlow seems like a promising solution and does not rely on the RDMA hardware capabilities, it suffers from being an unmaintained research project and is, unfortunately, not fit for general use. Further, FreeFlow, even with FastPath enabled, exhibits latency penalties on small message sizes, thus is likely not viable for latency sensitive applications.

3.1.3 MasQ

MasQ operates almost identically to FreeFlow, but instead leaves the entire RDMA data plane intact. This meaning that queue pairs live in the container's memory space and are

directly exposed to the RDMA device, thereby allowing for low latency RDMA operations without the need for a pinned CPU core like FreeFlow. This comes at the cost of being unable to enforce traffic shaping policies, as the controller is no longer on the RDMA data plane. However, as the controller is still interposed in the RDMA control plane, address translation, routing policies, and the notion of virtual RDMA devices is still applicable.

MasQ is not publicly available and is similar enough to FreeFlow that it will not be fully elaborated on. The key differences between MasQ and FreeFlow are: (1) reduced controllability, (2) reduced resource utilization, and (3) improved latency performance; as well as being proprietary and inaccessible for public use [15].

3.2 Hardware Solutions

Hardware solutions rely on functionality built into the RDMA-capable NIC itself. While these solutions tend to be the fastest as they require no software interposition from the kernel, they tend to be highly proprietary and costly, require the newest versions of RDMA-capable NICs, and are often not compatible with older RDMA-capable NICs. Further, they to have strict scalability limits as hardware resources on NICs are limited.

3.2.1 SR-IOV

As explained in Section 2.2.2, SR-IOV is a method of multiplexing RDMA-capable NICs in hardware and is traditionally used in the context of virtual machines. It has seen adoption as an RDMA in container solution as each virtual function of an RDMA-capable NIC is exposed as a unique, isolated RDMA device that can then be used by individual containers. These virtual functions can then be plugged into RDMA namespaces to isolate

each container's RDMA view to just that singular RDMA device. Further, with SR-IOV being a hardware feature, some RDMA-capable NICs allow for hardware enforcement of VLAN tagging and flow control per virtual function, which could then be applied to SR-IOV usage in containers.

Recall that SR-IOV cannot scale past a certain number of virtual functions due to hardware resource limitations. Further, because these are still hardware devices, virtual functions cannot integrate with kernel network isolation systems.

Container Network Properties

Network Isolation each NIC is given its own RDMA device, which can have its own IP address and VLAN tags; however the IP address must be exposed to the host network

Controllability routing policies and traffic shaping capabilities are not possible unless implemented in the device

Resource Utilization consumes SR-IOV slots in the NIC and can exhaust the GID table of the NIC, however does not require any other resources

Performance Tests

All of the following tests were performed on Cloudlab with two d6515 hosts, which have a 32 core AMD 7452 at 2.35GHz CPU, 128GB RAM, and a dual port ConnectX–5 100 Gbit NIC [8]. Both were running Ubuntu 20.04 with Linux kernel version 5.4.0, and were using the Mellanox OFED RDMA stack version 5.5-1.0.3.2.

Figures 17, 18, and 19 were all performed with either a single SR-IOV device (labeled `sriov`) or a single raw RDMA device (labeled `host`), with varying RDMA message sizes.

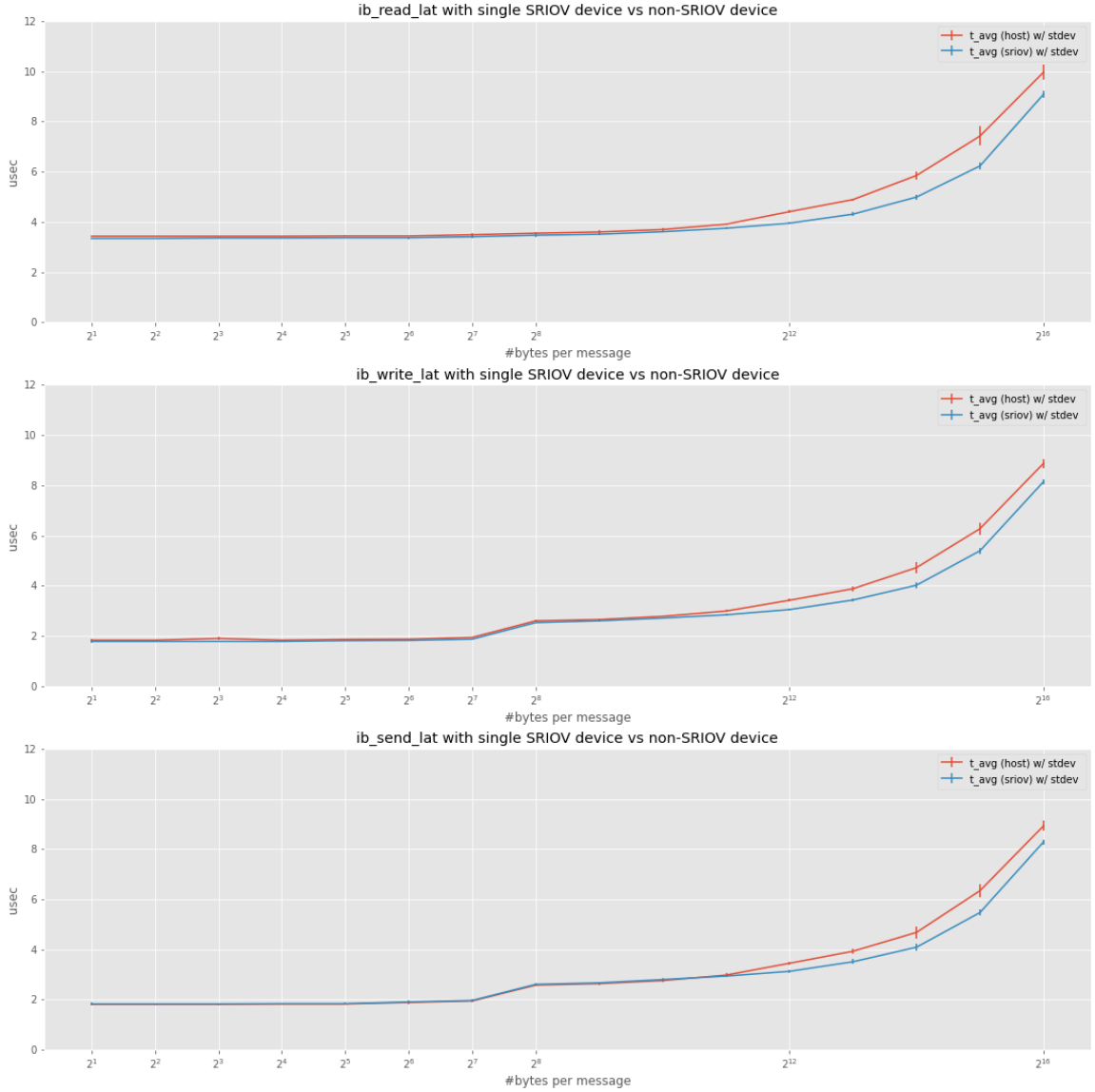

Figure 17: Latency Tests

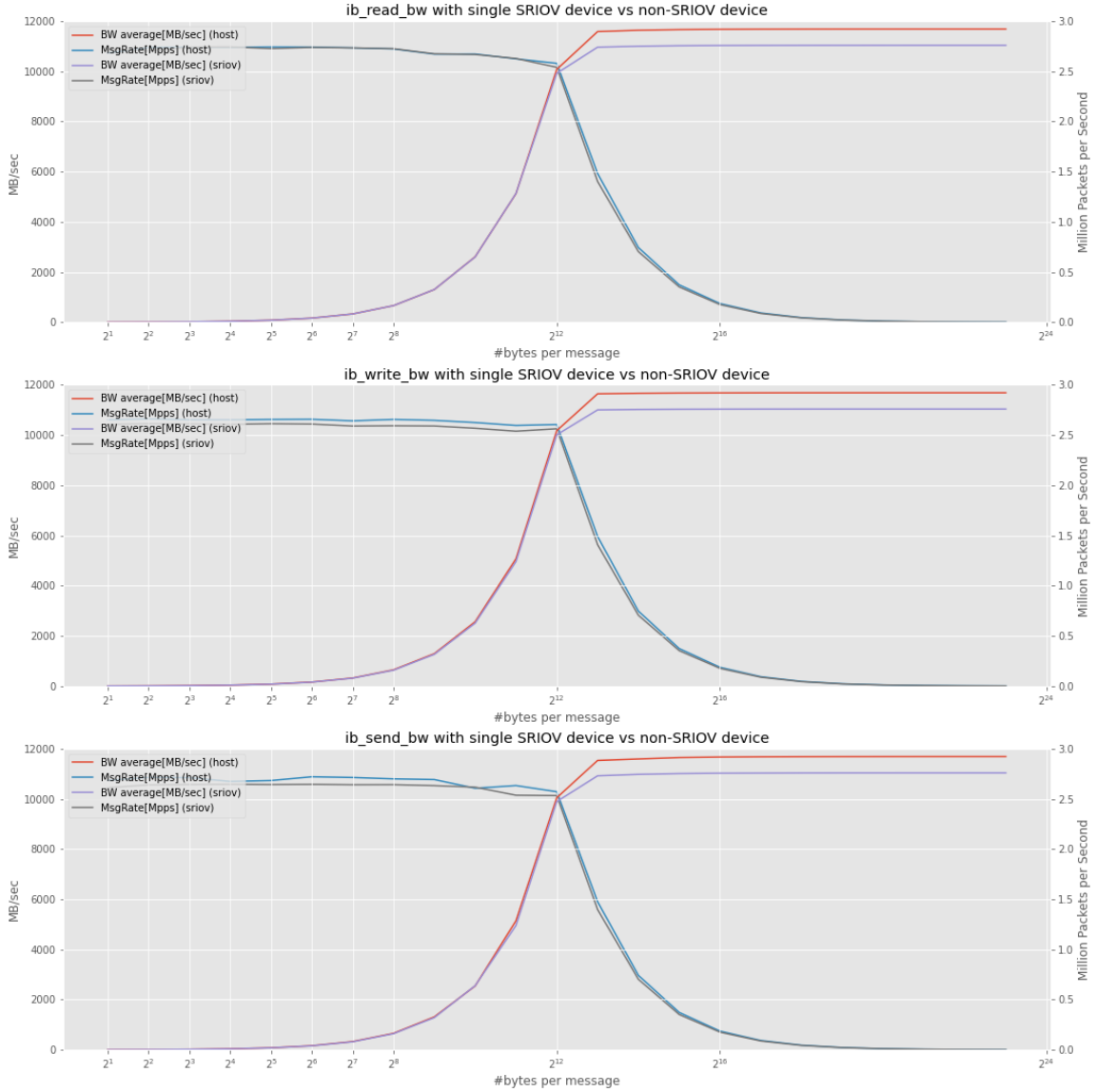

Figure 18: Bandwidth Tests

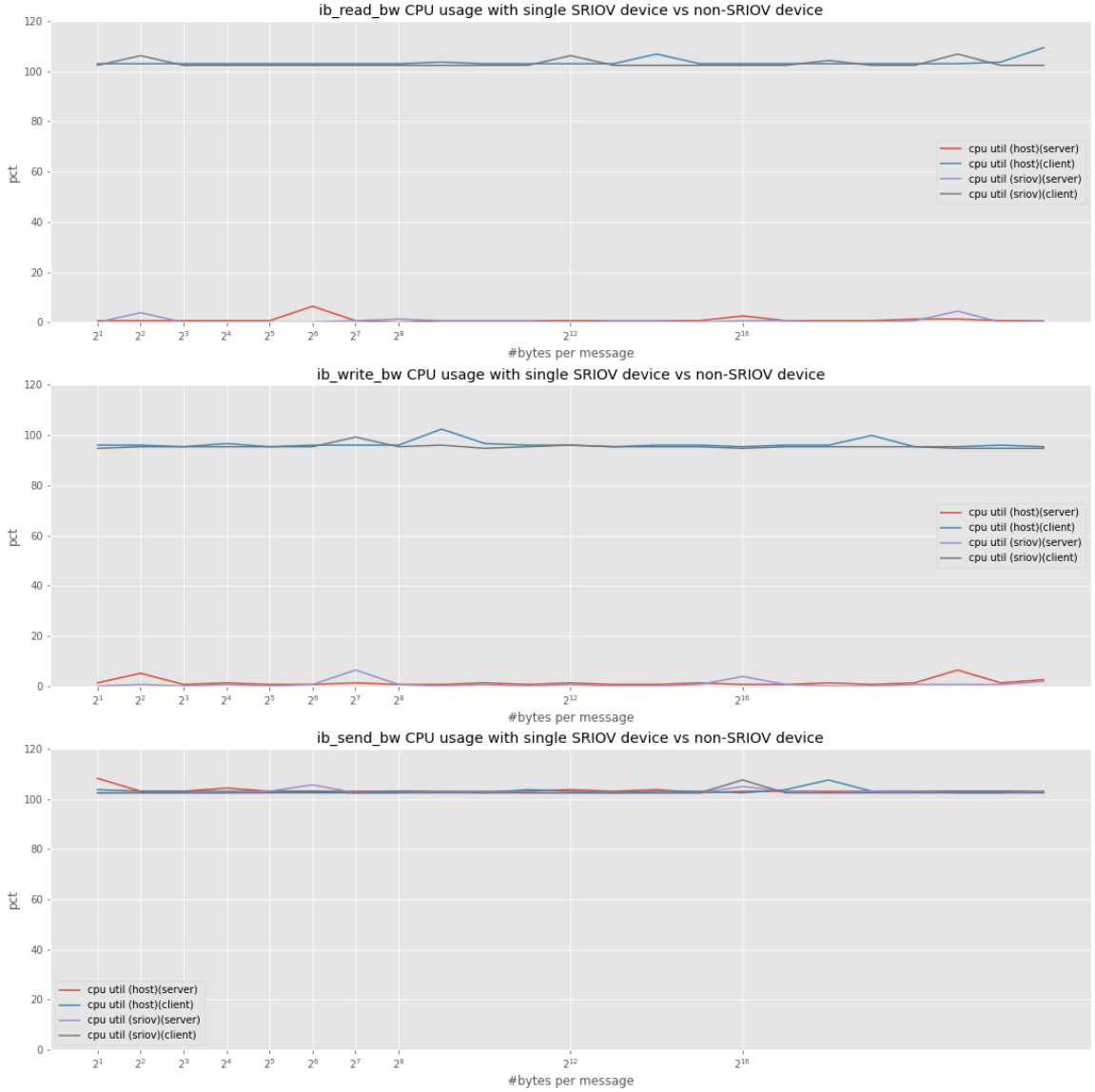

Figure 19: CPU Utilization

The use of SR-IOV does appear to have an impact on bandwidth, but surprisingly, appears to perform better than the raw host device with latency. Figure 17 shows that latency appears to be about equal with host performance for small message sizes and becomes up to 1 microsecond better with larger payloads. Figure 18 shows that, for bandwidth, SR-IOV appears to be capped at about 500 to 1000 MB/sec less than if using the raw host RDMA device. As expected, Figure 19 shows CPU performance is not worse when using an SR-IOV device as all of the device multiplexing is done on the NIC itself. Unfortunately, with SR-IOV implementations being a black box, it's difficult to reason about why the use of SR-IOV incurs bandwidth penalties and why it appears to perform better in terms of latency.

Figure 20 was performed with multiple SR-IOV devices created on both the client and server, with a single `ib_(read|write|send)_bw` test running on each device with a byte size of 65536. SR-IOV appears to not suffer in bandwidth with the use of additional SR-IOV devices, indicating its performance scales nicely. However, similar to the single device tests, it seems that even the use of multiple SR-IOV devices has a bandwidth cap less than what is achievable with the raw host RDMA device. SR-IOV did have one issue scaling however; the configuration of virtual functions appeared to get buggier and buggier with the more virtual functions that were provisioned. As a result, data is only reported for up to 22 configured virtual functions as any attempts to provision more virtual functions failed.

Miscellaneous Comments

Scalability Limits limited by both the number of possible SR-IOV virtual functions on the NIC, as well as the number of GID table entries available (ex. for RoCEv2,

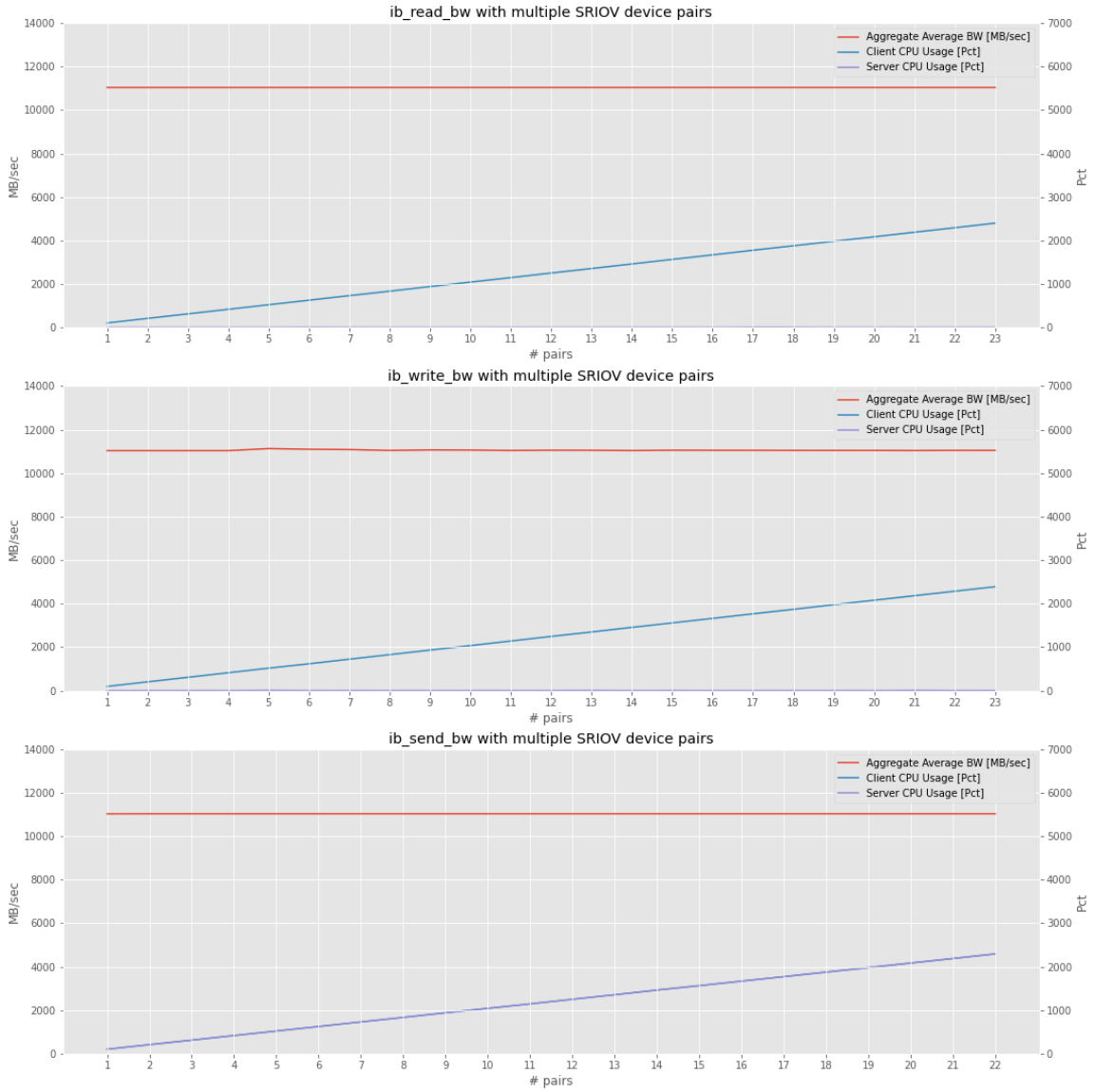

Figure 20: Bandwidth + CPU with Multiple Device

SR-IOV VFs need at least 2 entries)

Proprietary requires the NIC hardware to support it, some controllability features like flow control are dependent on the NIC features as well

Maturity has been commonly used for virtual machines and is mature enough for general use

Ease in Deployment requires custom handlers for container engines to provision, but these are commonly provided by NIC vendors thus is relatively easy to deploy

Execution Privileges requires privileges to provision (and possibly control) new SR-IOV NICs

Network Pressure requires new IP address and MAC addresses for each virtual function on Ethernet; adds no additional network pressure on Infiniband

SR-IOV provides many of the same benefits as shared HCA, thus in most situations, it may be advisable to use shared HCAs instead to avoid the limit on the number of possible virtual functions and the slight performance penalty for using SR-IOV. However, shared HCAs do not fully isolate RDMA devices and do not integrate as nicely with the control features of the NIC hardware, thus if this these are required, SR-IOV may be more desirable [31, 49].

3.2.2 Hardware Offload: Mellanox's ASAP^2 Direct

Hardware offload allows for executing certain routing and traffic shaping policies on the NIC's hardware — potentially with specialized hardware like FPGAs — instead of in the kernel. This provides significant performance gains in the execution of networking policies, and can potentially result in elimination of CPU interposition in network policy

enforcement all together. However, most forms of hardware offload have been limited in their expressiveness, for example being limited to very specific TCP offloads, thus were not capable of executing the complex policies required for container networks. This has changed with modern NICs, particularly high performance NICs like those supporting RDMA operations, which have added hardware offload support for software defined network policies (e.g. Open-vSwitch policies). If these policies are expressive enough they can be used for container isolation and, subsequently, inter-machine RDMA isolation with minimal overhead and with negligible performance impact.

One notable and more accessible hardware offload system that supports RDMA is Mellanox's ASAP^2 Direct, which allows for executing Open-vSwitch policies and operations on the NIC hardware [7]. Together with SR-IOV, ASAP^2 Direct can be used for full hardware enforced inter and intra-machine RDMA device isolation; allowing for extremely performant, isolated use of RDMA inside of containers. ASAP^2 Direct is not the only form of hardware offload for RDMA, but may be the most accessible given the prevalence of ConnectX series NICs in high performance computing environments and with it not being a cloud proprietary solution (e.g. Azure accelerated NICs). Thus, while these characteristics are not necessarily generalizable to all hardware offloads systems, its characteristics will be described below to give a general overview of what one form of hardware offload can provide.

Container Network Properties

Network Isolation when used with SR-IOV, full network isolation can be achieved, otherwise hardware offload cannot provide individual interfaces to containers

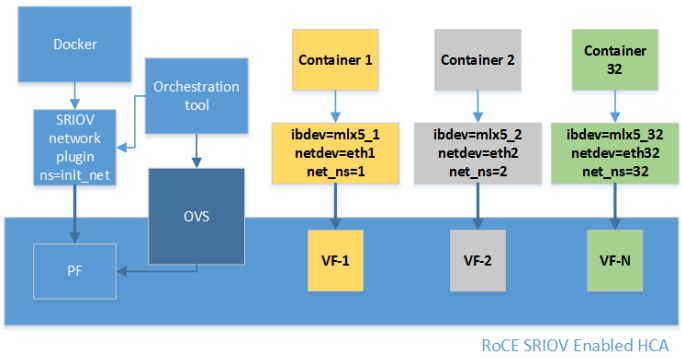

Figure 21: SR-IOV + OVS (w/ ASAP^2 Direct Hardware Offload) Architecture
Source: [7]

Controllability routing policies and traffic shaping are possible as part of Open vSwitch

Resource Utilization requires no host CPU or memory utilization

Performance Tests

All of the following tests were performed on Cloudlab with two c6525-100g hosts, which have a 24 core AMD 7402P at 2.8GHz CPU, 128GB RAM, and a dual port ConnectX–5 Ex 100 Gbit NIC [8]. Both were running Ubuntu 20.04 with Linux kernel version 5.4.0, and were using the Mellanox OFED RDMA stack version 5.5-1.0.3.2. As ASAP^2 Direct does not provide virtual interfaces itself, the tests configured ASAP^2 Direct hardware offload rules on top of SR-IOV virtual functions, as detailed in [7]. The tests also had configured one basic `tc-filter flower` rule to direct traffic from the VFs to the physical function, which was hardware offloaded. These results could differ widely depending on the number of Open-VSwitch rules configured, thus these results are only ballpark figures.

Figures 22, 23, and 24 were all performed with either a single SR-IOV device (labeled `sriov_ovs`) or a single raw RDMA device (labeled `host`), with varying RDMA message sizes. Figure 22 indicates that using ASAP^2 Direct has a noticeable latency overhead,

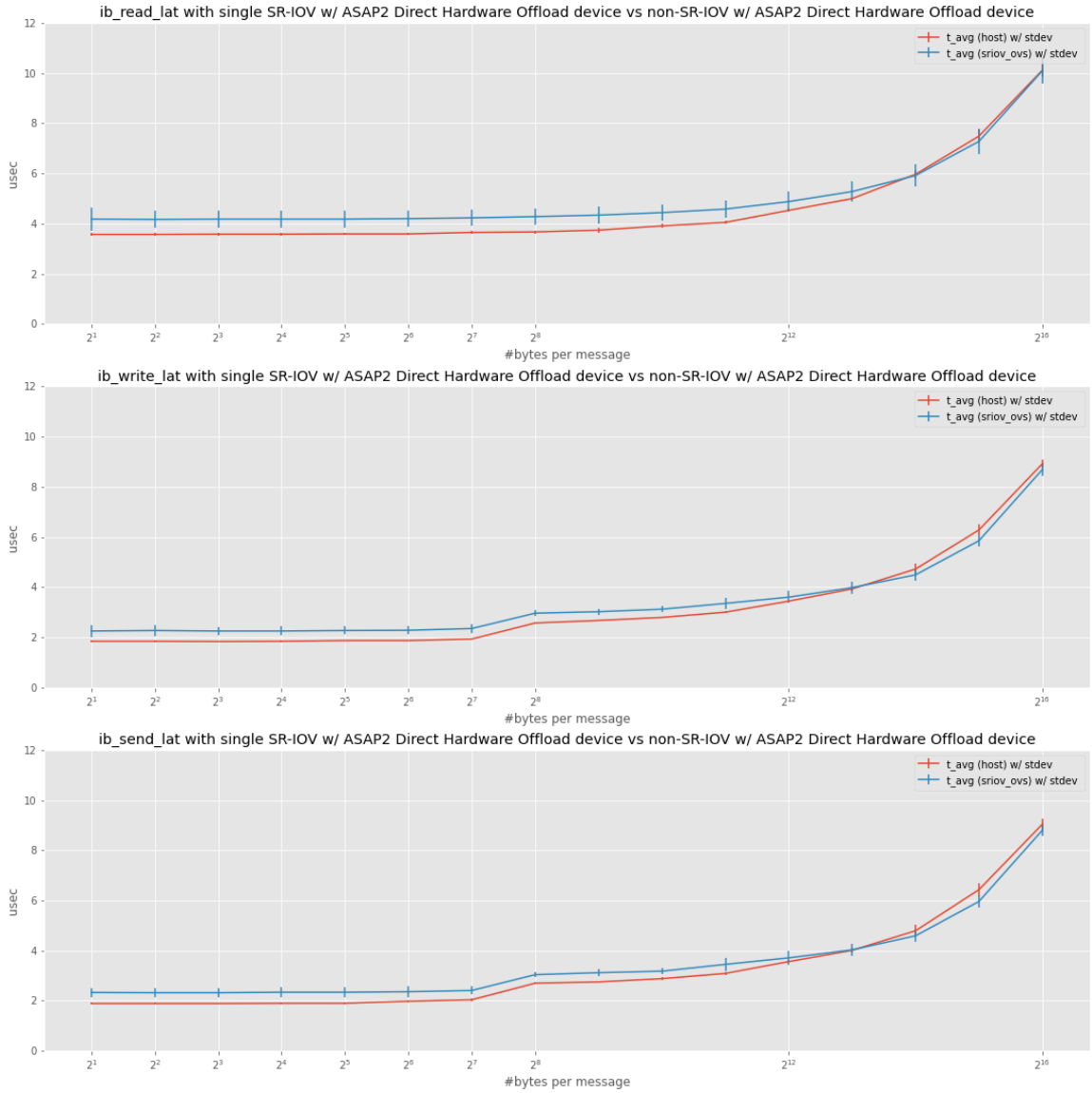

Figure 22: Latency Tests

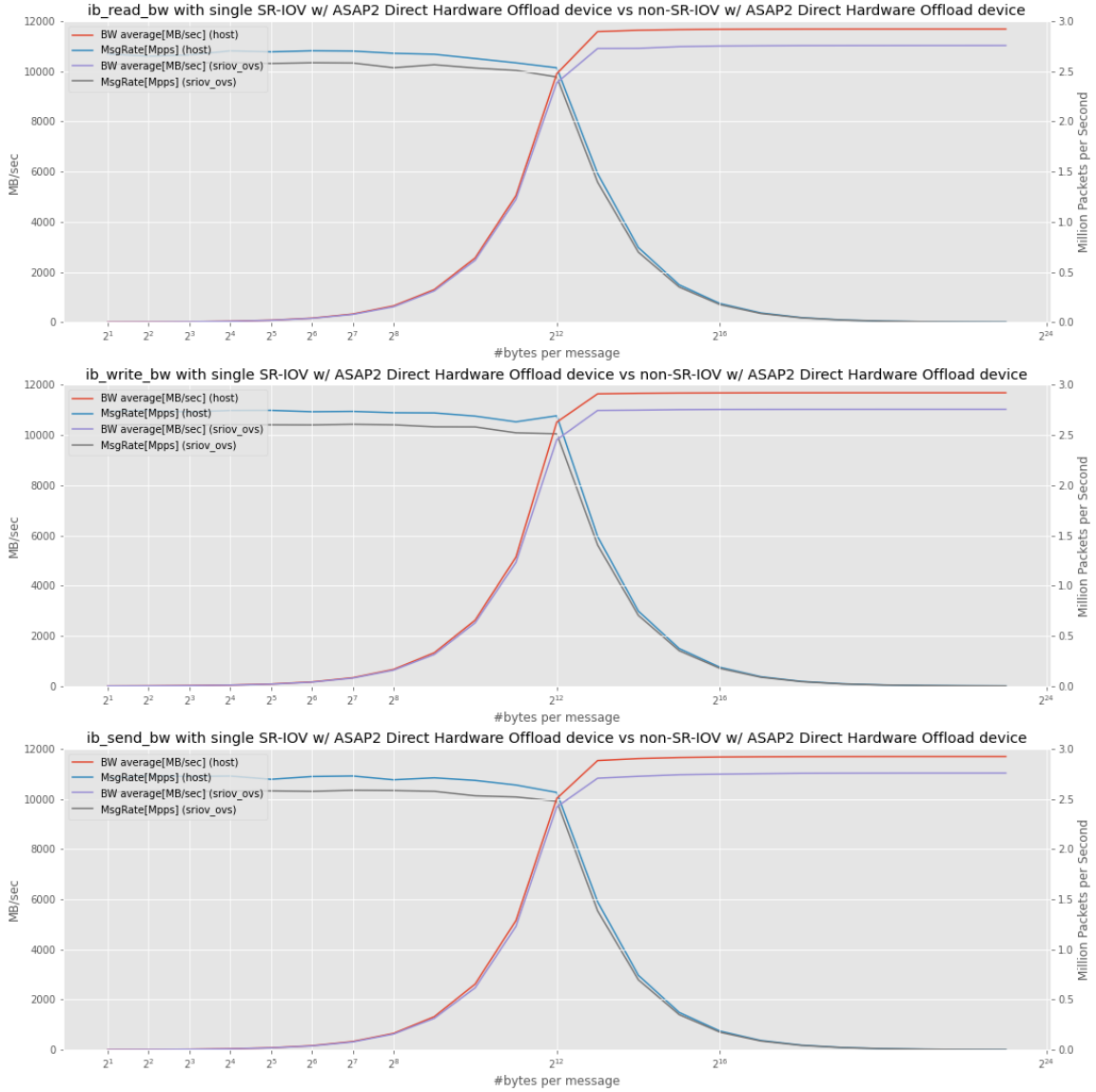

Figure 23: Bandwidth Tests

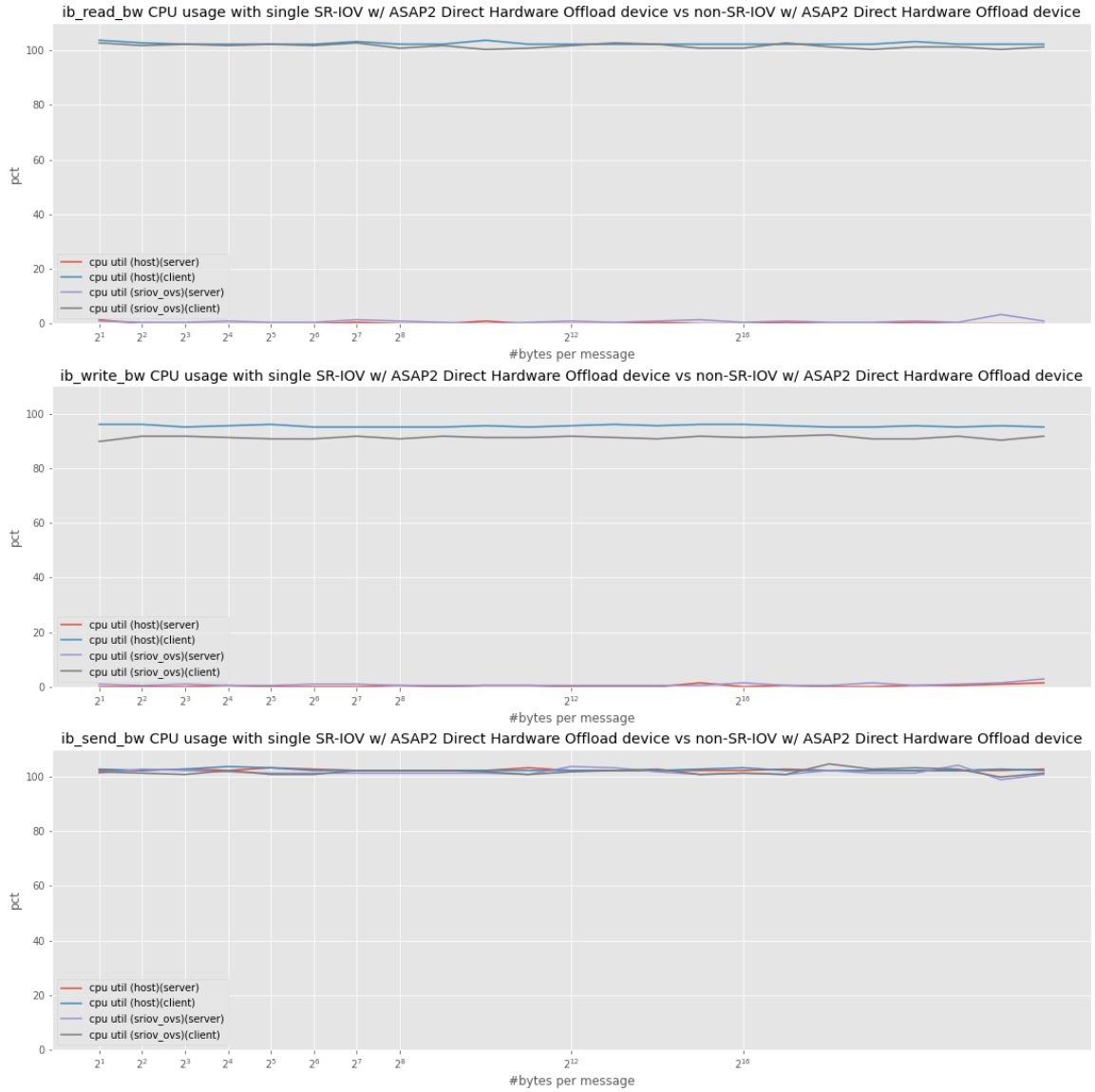

Figure 24: CPU Utilization

more so than when just using SR-IOV alone. Interestingly, this latency overhead appears to disappear at large payloads, where it tended to increase with SR-IOV alone. Figure 23 shows bandwidth results very similar to that of using SR-IOV alone, indicating perhaps that ASAP^2 Direct does not have much of an impact on bandwidth and the majority of that overhead comes from the use of SR-IOV. Again, as expected, Figure 24 shows CPU performance is identical as using the raw host RDMA device as all of the Open-VSwitch processing and NIC multiplexing is happening on the NIC and not on the CPU. Similarly to SR-IOV, ASAP^2 Direct is a black box so it's difficult to reason about why it has these latency and bandwidth overheads, and why they may differ from using SR-IOV alone.

Figure 25 was performed with multiple SR-IOV devices created on both the client and server, with a single `ib_(read|write|send)_bw` test running on each device with a byte size of 65536. Each device pair had a unique `tc filter flower` rule configured such that traffic from one device in the pair could only be sent to the other device in the pair, which was enforced with MAC address matching in the rule. Similar to the single device tests, these rules were hardware offloaded with ASAP^2 Direct.

It appears ASAP^2 Direct also scales quite nicely in bandwidth up to 16 devices, indicating that NIC was able to cleanly handle these additional hardware offloaded rules. However, similar to the SR-IOV tests, the scalability was only tested to up to 16 devices and virtual function provisioning became buggy after 16 devices.

It may be possible to use ASAP^2 Direct with shared HCA devices which, if possible, is likely to provide better performance than when used with SR-IOV. However, this was not tested as Mellanox provides no documentation for how to use ASAP^2 Direct with shared HCAs.

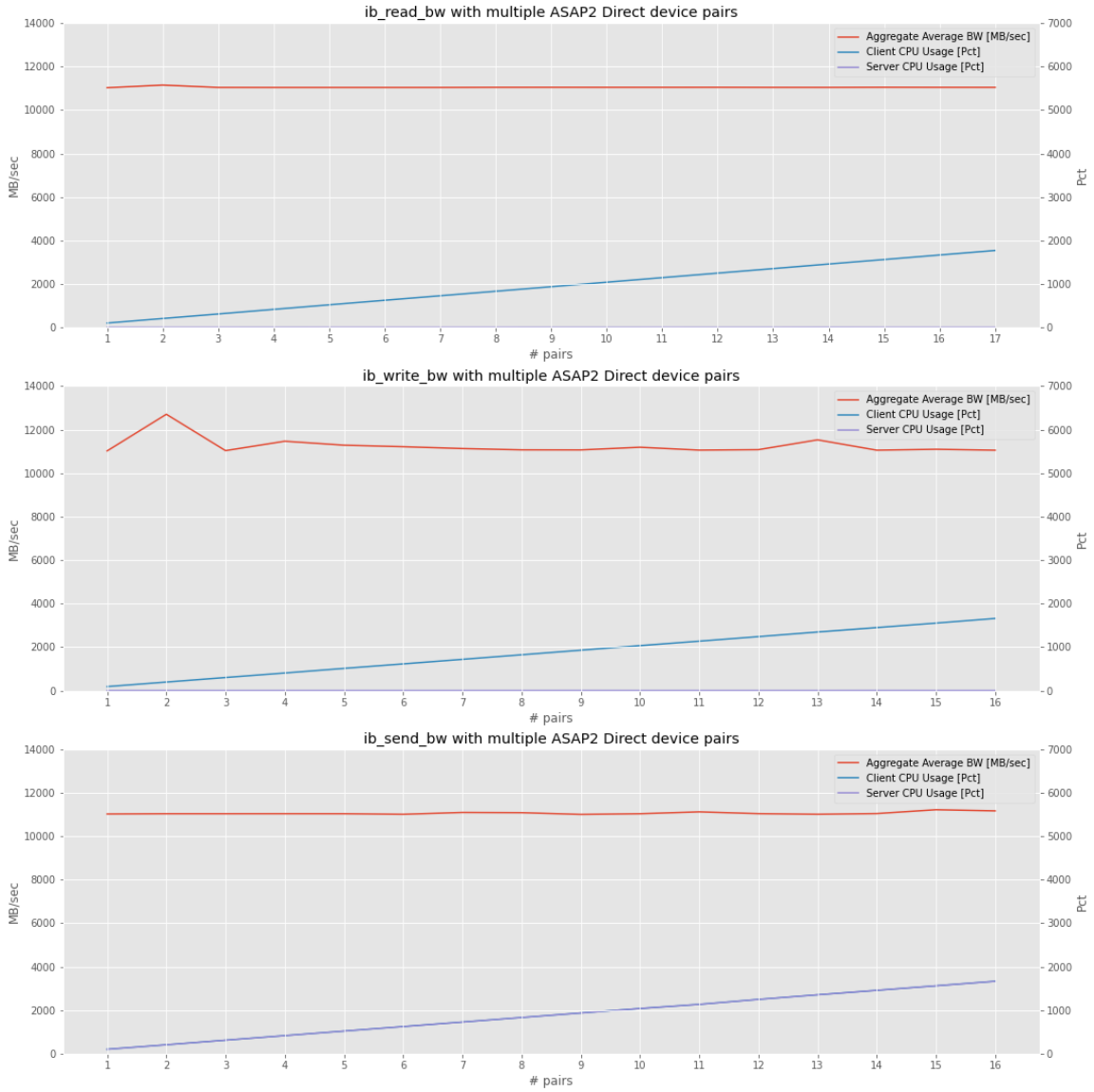

Figure 25: Bandwidth + CPU with Multiple Devices

Miscellaneous Comments

Scalability Limits unknown how many policies can be defined on a NIC at once

Proprietary policy definitions are not proprietary, but policy execution is entirely proprietary and requires specialized Mellanox hardware

Maturity unknown, but with heavy vendor support appears to be mature enough for production use

Ease in Deployment requires integration with SR-IOV and specialized Open vSwitch policy definitions, not as easy to plug and play

Execution Privileges requires no additional privileges

Network Pressure no additional network pressure when used with SR-IOV

3.3 Software Solutions: SoftRoCE

SoftRoCE is a full software implementation of a RoCE-capable RDMA stack that can be used to create virtualized RDMA devices that run on top of existing Ethernet NICs [46]. The "fully in-software" bit allows for total configuration of the virtualized RDMA device, which can be used to create fully isolated, virtualized RDMA devices that preserve and integrate with all container isolation systems. However, it also come at the cost of massive performance penalties from: (1) being unable to provide any hardware performance benefits from standard hardware RDMA devices, (2) requiring full software execution of the RDMA stack, and (3) having to be run on top of the existing kernel TCP/UDP stack. While some of these are optimized for with the SoftRoCE now being implemented as a kernel driver, SoftRoCE still remains a low performance RDMA solution [25]. Thus, SoftRoCE is likely

viable only for specific niche use cases, such as for those who don't need the performance of RDMA and are simply seeking to enable RDMA compatibility (e.g. an RDMA client inside of a container).

Container Network Properties

Network Isolation provides full container network isolation

Controllability routing policies and traffic shaping are possible and fully configurable

Resource Utilization requires significant CPU and memory to run the RDMA stack

Performance Tests

All of the following tests were performed on Cloudlab with two d6515 hosts, which have a 32 core AMD 7452 at 2.35GHz CPU, 128GB RAM, and a dual port ConnectX–5 100 Gbit NIC [8]. Both were running Ubuntu 21.10 with Linux kernel version 5.13.0-35[10], and were using the open source `rdma-core` and `perftest` packages.

Figures 26, 27, and 28 were all performed with either a single SoftRoCE device (labeled `rxe`) or a single raw RDMA device (labeled `host`), with varying RDMA message sizes. Figure 26 shows a significant increase in both latency and deviation in latency when compared to using a non-SoftRoCE device (the non-SoftRoCE device also has error bands, but they are so small they are imperceivable in the graph). Figure 27 highlights how a single SoftRoCE device has both a bandwidth and message rate potential that is significantly less than that of a non-SoftRoCE device. Figure 28 shows the total CPU usage during an `ib_read_bw` test on both the client and server. Unsurprisingly, SoftRoCE usage resulted

[10]Linux kernel version 5.8 on Ubuntu 20.04 appears to have a SoftRoCE bug that caused multiple kernel panics and was thus unusable.

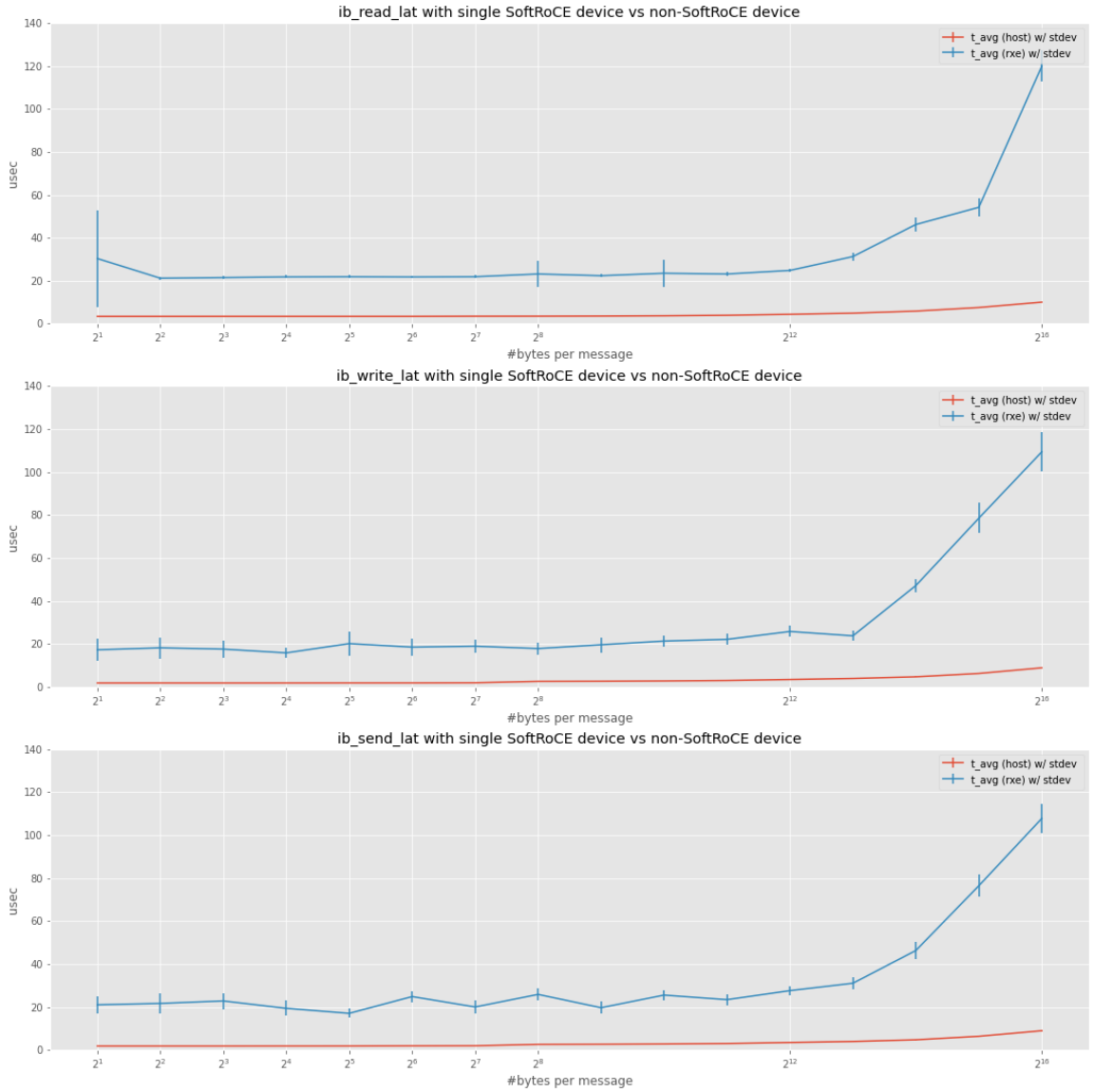

Figure 26: Latency Tests

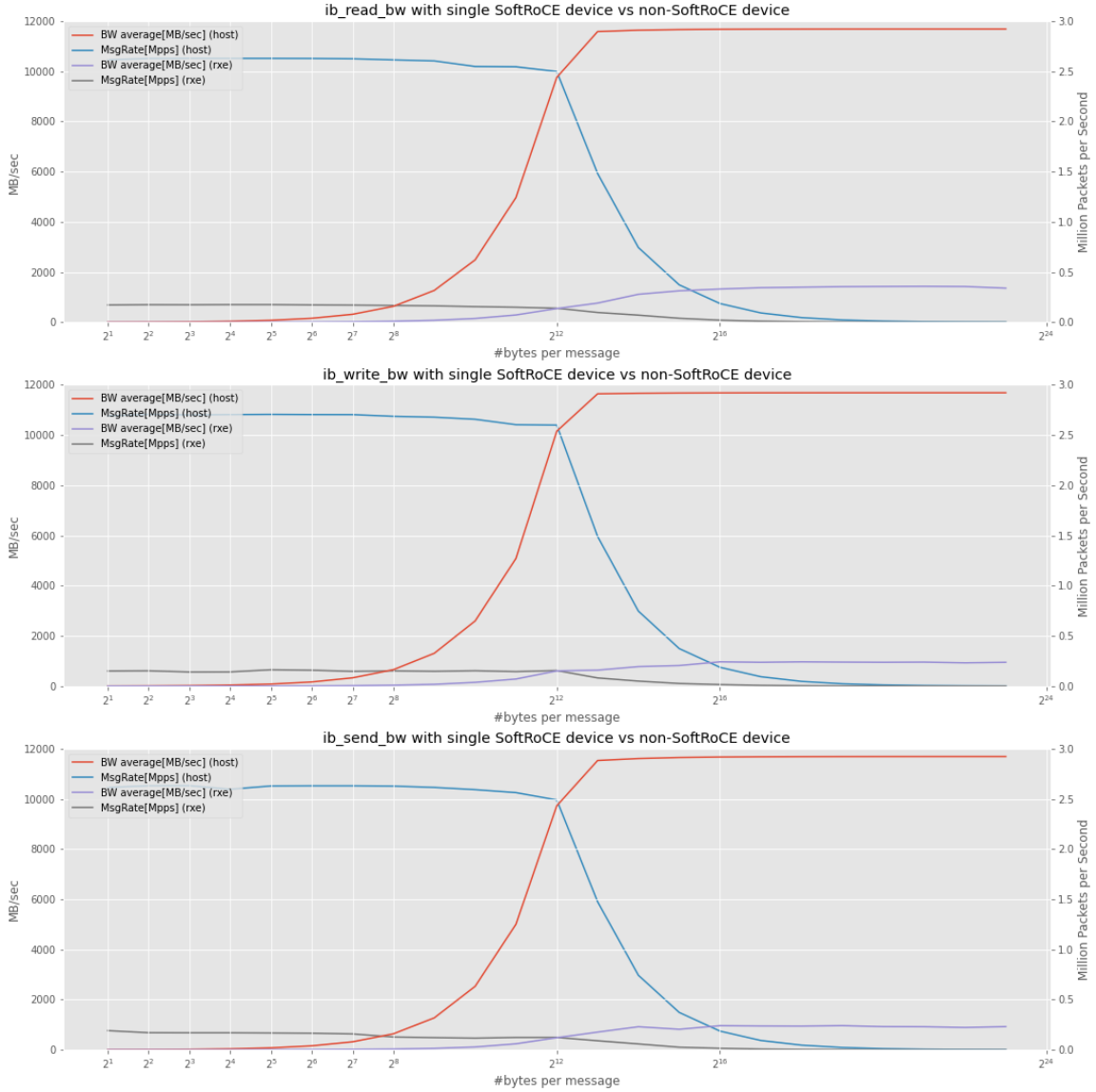

Figure 27: Bandwidth Tests

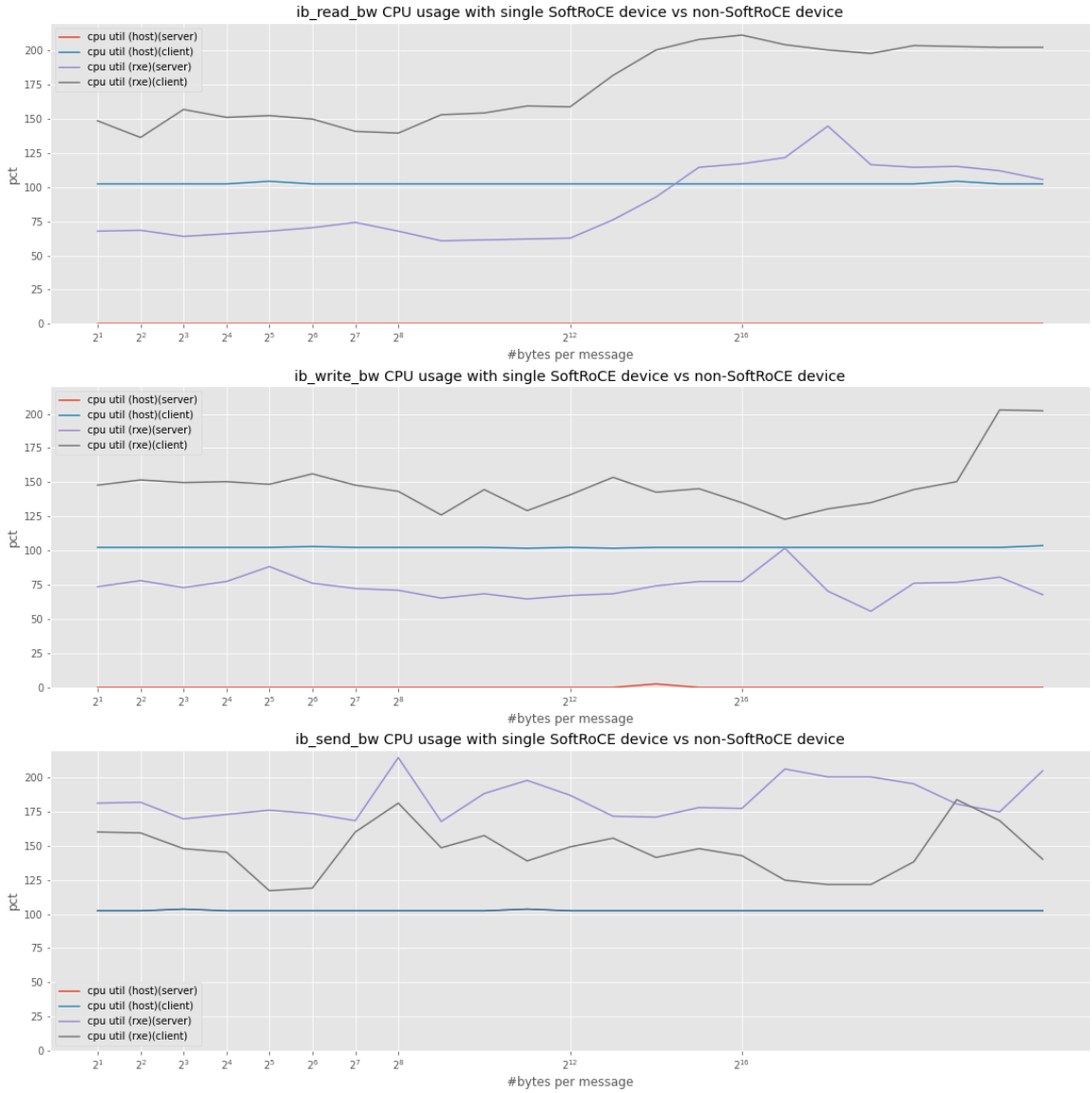

Figure 28: CPU Utilization

in more CPU utilization than a non-SoftRoCE device on the client. Furthermore, the total CPU usage across both the server and client is greater with a SoftRoCE device than on a non-SoftRoCE device, which is expected. In normal RDMA, the server requires no CPU utilization (for one-sided operations) as the RDMA device itself handles that. With SoftRoCE, the server has to spend CPU cycles processing incoming packets, resulting in higher CPU utilization. One noticable feature about the CPU graphs is how SoftRoCE usage can result in a CPU utilization of 200%. This is likely due to one CPU core being entirely used for SoftRoCE processing and one CPU core being entirely used for the actual `ib_read_bw` program.

Figure 29 was performed with multiple SoftRoCE devices created on both the client and server, with a single `ib_(read|write|send)_bw` test running on each device with a byte size of 65536. For each SoftRoCE device, a `macvlan` interface was created on the RDMA device's interface to serve as the SoftRoCE device's parent, which is necessary as an interface can only parent one SoftRoCE device.

This data highlights several things. First, use of SoftRoCE alone appears to be capable of fully saturating the underlying NIC's bandwidth. In this case, for `ib_read_bw`, it took approximately 18 pairs of SoftRoCE devices, approximately 30 CPU cores worth of CPU cycles on the server, and approximately 19 CPU cores worth of CPU cycles on the client to saturate the underlying NIC's bandwidth (i.e. match the bandwidth performance of the non-SoftRoCE device). Second, even after saturating the underlying NIC's bandwidth, client CPU utilization continued to climb linearly with each additional SoftRoCE device.

All in all, these graphs highlight how SoftRoCE fails to match the bandwidth, latency, and CPU performance of a raw RDMA device, and is not desirable for use in performance

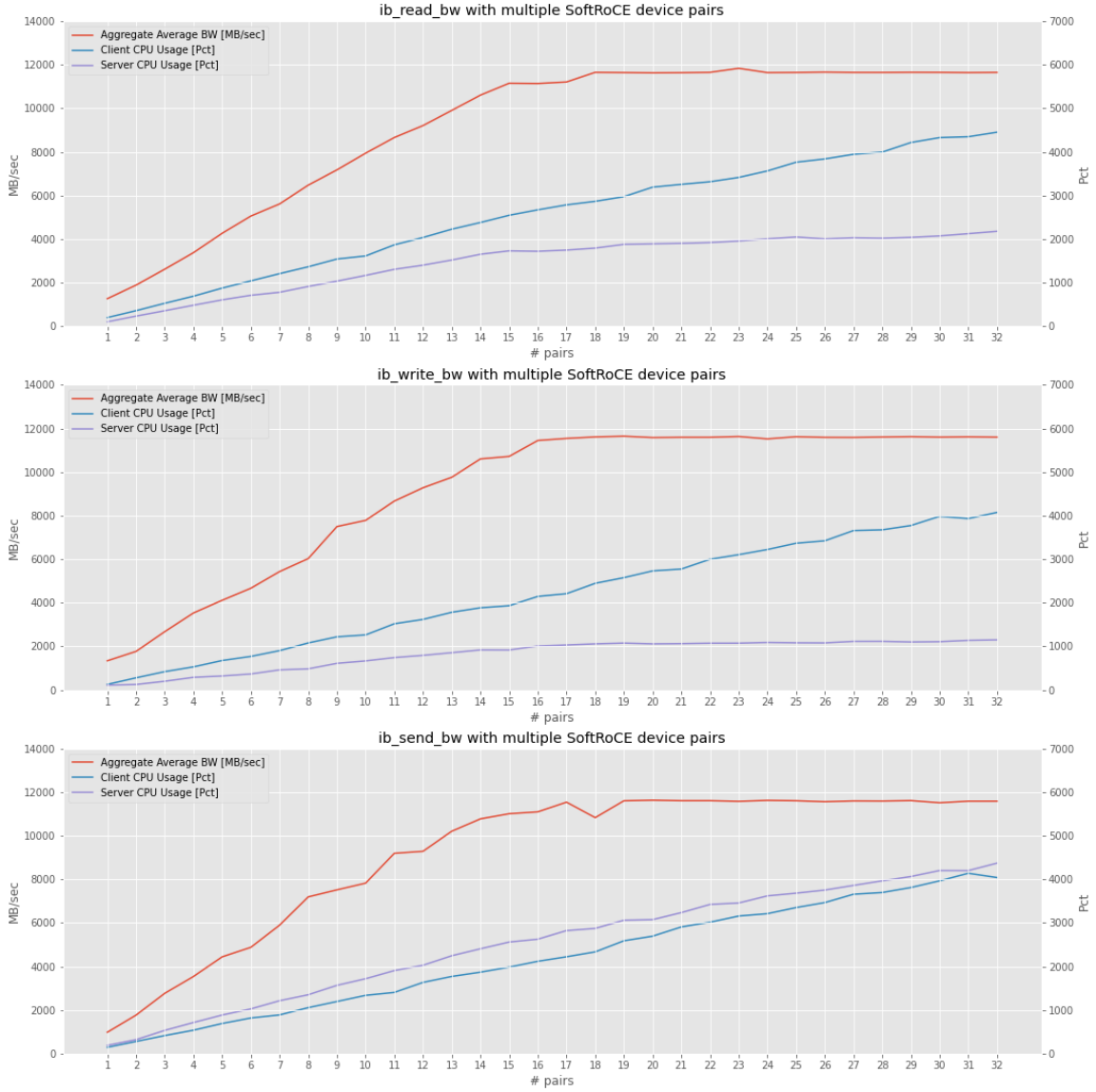

Figure 29: Bandwidth + CPU with Multiple Devices

sensitive environments. SoftRoCE can scale to saturate the underlying NIC's bandwidth, but requires substantial CPU utilization which significantly hurts its scalability.

Miscellaneous Comments

Scalability Limits cannot scale due to CPU contention

Proprietary open source kernel driver that has been upstreamed, so it is available to all Linux users

Maturity relatively mature, but has little development outside of the research community thus suffers from a number of bugs [25]

Ease in Deployment quick to deploy using standard `iproute2` tools

Execution Privileges requires no additional privileges

Network Pressure creates no additional network pressure than using standard networking

4 Discussion

4.1 Security

Security is often a concern among those adopting containers, especially when using containers to facilitate a multi-tenant environment. Focusing on just RDMA for a minute, RDMA provides on-host security with the use of the RDMA management plane and cryptographic keys, as explained in Section 2.2. The management plane ensures RDMA devices cannot be reconfigured without the proper permissions on the host and cryptographic keys provide a mechanism to ensure remote hosts cannot read arbitrary memory regions. However, RDMA

suffers from minimal network security as RDMA packets have no form of encryption, thus making RDMA vulnerable to eavesdropping, man-in-the-middle, and tampering attacks. This is particularly a problem as RDMA packets contain the `pkey` — the cryptographic key used to authorize access to remote memory regions — in plaintext in the unencrypted RDMA packet. If eavesdropped on, obtaining this `pkey` would allow outside actors to bypass all RDMA memory region restrictions [44].

Solving this is non-trivial; encryption would require reading and rewriting the entirety of the RDMA packet. As RDMA bypasses the kernel and CPU, encryption would then have to be done on the RDMA NIC itself and, until recently, no NIC has provided this feature. Given this, the use of RDMA in containers would inherit this security vulnerability and no RDMA in container solution would be able to solve this as it must be done at the NIC level.

For the security of the management plane for RDMA in container solutions, almost every solution inherits the security provided by the core RDMA protocol. Some solutions, such as FreeFlow, would have to implement their own security model for the management of their virtual RDMA devices as only FreeFlow is aware of these devices. For other solutions where the kernel is aware of the virtual devices, these solutions would piggyback on existing RDMA management plane security and their use would not open any potential security concerns.

Beyond the security vulnerabilities introduced by RDMA itself, use of these RDMA-in-container solutions do open a security hole if they do not provide true inter-host / network isolation. Nearly all RDMA devices have been shown to have deterministic, or at the very least highly guessable, memory key generation algorithms [43]. Thus, even if network operators limited the snooping ability of RDMA devices inside of containers, its

possible for individual containers to guess the memory keys of remote hosts. This would render multi-tenant environments with RDMA-in-container solutions extremely insecure. Environments looking to use an RDMA-in-container solution in a multi-tenant environment are recommended to create full, network level isolation between each RDMA-in-container device to mitigate these issues.

4.2 Network Reconfiguration

One other potential solution for enabling RDMA-in-containers would be to use network reconfiguration, where the container network policies are enforced by the underlying network instead of by the host or NIC. Alongside the use of SR-IOV, shared HCAs, or VLAN tagging, this could provide another fast, configurable, and low resource utilizing solution for RDMA-in-containers. This would operate similar to existing container networking solutions like Project Calico using BGP [6]. With the variability in network configuration, types (e.g. Infiniband or Ethernet), privilege escalation risks (e.g. would the container orchestration software itself be able to reconfigure the network?), and available network resources, it is difficult to generalize properties of this solution thus is left for each environment to assess for themselves.

4.3 Looking Forward

It seems the future of RDMA in containers is converging toward a paravirtualization-like approach for intra-machine RDMA device isolation and a hardware-based approach for inter-machine RDMA device isolation. In other words, paravirtualization — likely in the

form of kernel managed virtual RDMA devices with the NIC having some visibility into those — would be used to create individual RDMA devices for each container. Hardware offload would be used to enforce network policies for each of those virtual devices. This suggests that a true RDMA in container solution would require new, specialized NICs that have both hardware offload and paravirtualization support, meaning RDMA in containers would incur additional costs for new hardware and will not be possible on older, existing RDMA hardware.

This comes, in part, from the direction and substantial investment Mellanox is making in this space. Unfortunately, while projects like FreeFlow and MasQ are desirable from an accessibility perspective, they suffer from a number of problems. First, much of the RDMA space is industry driven to begin with, especially given how complex and hardware specific the RDMA protocol and code base is. It's difficult for non-industry driven solutions to remain relevant as they would have to continually pull and patch upstream changes in the RDMA protocol/code base, which is difficult to maintain. Second, as shown by the data, purely in software approaches require substantial CPU sacrifices and suffer in scalability. This indicates that at least part of the RDMA in container solution must involve a form of hardware offload. Third, for high performance computing environments, the security and support provided by vendor support solutions is unmatched when compared to small research projects like FreeFlow. It would seem obvious these environments should rely on vendor supported solutions for this reason alone.

However, while the short term accessibility for RDMA-in-containers looks bleak (likely locked in to using Mellanox hardware), the long term accessibility appears bright. RDMA namespaces, RDMA cgroups, and hardware offload APIs are being developed openly in

the Linux kernel and in collaboration with Open-VSwitch, thus making it possible for other vendors to support it as well. This also means there would be a single, unified interface to configure RDMA-in-containers (through tools like the `rdma` cli and `iproute2` package), which would further improve accessibility.

5 Conclusion

Containers present a compelling story for adoption in high performance computing environments in almost every way but in their support for RDMA. Despite the multiple solutions and advancements made to improve RDMA support inside containers, it remains that, currently, substantial assessment is required by each environment to determine which RDMA-in-containers solution is viable given each has its own unique downsides.

For example, multi-tenant environments may need complete inter-host / network isolation and traffic shaping, which can only be provided by two solutions (FreeFlow and ASAP^2 Direct) listed in this thesis. Both solutions have an unavoidable network performance penalty associated with them and require either a hardware cost (due to being vendor proprietary) or require significant CPU utilization. Non-multi-tenant environments may only need VLAN isolation between devices, which can be provided at zero-cost by using shared HCA device. In addition, every environment has separate support, scalability, and configurable needs, further adding to assessment required by each environment.

This thesis provides comprehensive data, analysis, and explanation on how they work for 5 different RDMA-in-containers solutions: shared HCAs, FreeFlow, SR-IOV, ASAP^2 Direct, and SoftRoCE. These range from being a pure software solution, to being a

paravirtualized solutions, to a few hardware specific solutions, providing a wide range in possible RDMA-in-containers solutions. This should aid high performance computing environments in deciding which solution is most viable for their environment. Further, this thesis provides reproducibility for the FreeFlow paper and makes available the code used to generate data for this thesis, which can be used to profile other types of hosts and NICs.

6 Acknowledgments

I would like to sincerely thank my thesis advisor, Dr. Ryan Stutsman, for all of his support in writing this thesis. I would also like to thank Joe Breen at the University of Utah Center for High Performance Computing for all of his guidance in the space of containers and networking.

7 Availability & Reproducibility

This thesis, all of the code used to run RDMA-in-container experiments, and the raw data collected from said experiments is open sourced and available at https://github.com/emersonford/thesis. All of the data produced for this thesis can be reproduced following the instructions listed in the Github repo's `README.md`.

References

[1] *200Gb/s ConnectX-6 Ethernet Single/Dual-Port Adapter IC*. NVIDIA. URL: https://www.mellanox.com/products/ethernet-adapter-ic/connectx-6-en-ic.

[2] Ubaid Abbasi et al. "A performance comparison of container networking alternatives". In: *IEEE Network* 33.4 (2019), pp. 178–185.

[3] Joe Breen et al. "Building the SLATE Platform". In: *Proceedings of the Practice and Experience on Advanced Research Computing*. Association for Computing Machinery, 2018. DOI: https://doi.org/10.1145/3219104.3219144.

[4] *cgroups(7)*. 5.12. Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html.

[5] Mark Church. "Understanding Docker Networking Drivers and their use cases". In: (Dec. 19, 2016). URL: https://www.docker.com/blog/understanding-docker-networking-drivers-use-cases/.

[6] *Configure Networking*. Project Calico, 2021. URL: https://projectcalico.docs.tigera.io/networking/configuring.

[7] *Docker RoCE SRIOV Networking using OVS with ConnectX4/ConnectX5*. Mellanox Technologies. Dec. 3, 2018. URL: https://community.mellanox.com/s/article/docker-roce-sriov-networking-using-ovs-with-connectx4-connectx5.

[8] Dmitry Duplyakin et al. “The Design and Operation of CloudLab”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[9] “eBPF - The Future of Networking & Security”. In: (Nov. 10, 2020). URL: https://cilium.io/blog/2020/11/10/ebpf-future-of-networking/.

[10] Haggai Eran. “Add network namespace support in the RDMA-CM”. In: (May 17, 2015). URL: https://lwn.net/Articles/644940/.

[11] Julia Evans. *What even is a container: namespaces and cgroups*. Oct. 10, 2016. URL: https://jvns.ca/categories/containers/.

[12] Shiqing Fan et al. “Towards a Lightweight RDMA Para-Virtualization for HPC”. In: *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*. 2017.

[13] Daniel Firestone et al. “Azure accelerated networking: Smartnics in the public cloud”. In: *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 2018, pp. 51–66.

[14] *GitHub Repository - RDMA core userspace libraries and daemons*. Version v36. linux-rdma. Aug. 1, 2021. URL: https://github.com/linux-rdma/rdma-core.

[15] Zhiqiang He et al. “MasQ: RDMA for Virtual Private Cloud”. In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 1–14.

[16] Jiayu Hu. “Accelerate Container Networking with Data Plane Development Kit (DPDK)”. In: (Oct. 22, 2018). URL: https://software.intel.com/content/www/us/en/develop/articles/accelerate-container-networking-with-dpdk.html.

[17] *IB Router Architecture and Functionality*. Mellanox Technologies. Dec. 3, 2018. URL: https://community.mellanox.com/s/article/ib-router-architecture-and-functionality.

[18] *Intel® Ethernet 800 Network Adapters (up to 100GbE)*. Intel. 2021. URL: https://www.intel.com/content/www/us/en/products/details/ethernet/800-network-adapters.html.

[19] *iproute2*. The Linux Foundation. Aug. 8, 2020. URL: https://wiki.linuxfoundation.org/networking/iproute2.

[20] Narūnas Kapočius. “Performance Studies of Kubernetes Network Solutions”. In: *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*. IEEE. 2020, pp. 1–6.

[21] Georgios P Katsikas et al. “What You Need to Know About (Smart) Network Interface Cards”. In: *PAM*. 2021, pp. 319–336.

[22] Daehyeok Kim et al. “FreeFlow: Software-based Virtual {RDMA} Networking for Containerized Clouds”. In: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, pp. 113–126.

[23] Duckwoo Kim, SeungEon Lee, and KyoungSoo Park. “A Case for SmartNIC-accelerated Private Communication”. In: *4th Asia-Pacific Workshop on Networking*. 2020, pp. 30–35.

[24] Kyungwoon Lee, Youngpil Kim, and Chuck Yoo. “The impact of container virtualization on network performance of IoT devices”. In: *Mobile Information Systems* 2018 (2018).

[25] Liran Liss. “The Linux SoftRoCE Driver”. In: *OpenFabrics Alliance 13th Annual Workshop 2017*. Mellanox Technologies. 2017. URL: https://www.openfabrics.org/images/eventpresos/2017presentations/205_SoftRoCE_LLiss.pdf.

[26] Jianshen Liu et al. “Performance Characteristics of the BlueField-2 SmartNIC”. In: *arXiv preprint arXiv:2105.06619* (2021).

[27] Marek Majkowski. *How to achieve low latency with 10Gbps Ethernet*. Cloudflare. 2015. URL: https://blog.cloudflare.com/how-to-achieve-low-latency/.

[28] *namespaces(7)*. 5.12. Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[29] *netfilter: firewalling, NAT, and packet mangling for Linux*. netfilter. July 28, 2021. URL: https://www.netfilter.org/.

[30] Quang-Huy Nguyen and Younghan Kim. “Performance of OVS-DPDK in Container Networks”. In: *2020 International Conference on Information and Communication Technology Convergence (ICTC)*. 2020, pp. 437–440. DOI: https://doi.org/10.1109/ICTC49870.2020.9289483.

[31] NVIDIA. *NVIDIA MLNX_OFED Documentation*. Version 5.1-0.6.6.0. Aug. 5, 2020. URL: https://docs.mellanox.com/x/5pXuAQ.

[32] NVIDIA. *RDMA Aware Networks Programming User Manual*. Version v1.7. URL: https://docs.mellanox.com/pages/viewpage.action?pageId=34256560.

[33] Yasufumi Ogawa. "Implementing DPDK based Application Container Framework with SPP". In: (Dec. 2018). URL: https://www.dpdk.org/wp-content/uploads/sites/35/2018/12/YasufumiOgwa_Implementing-DPDK-based-Application-Container-Framework-with-SPP.pdf.

[34] Shailesh Mani Pandey and Rajath Shashidhara. *SROCE: Software RDMA over Commodity Ethernet*.

[35] Parav Pandit and Dror Goldenberg. *Mellanox Container Journey*. Mellanox Technologies. June 2019. URL: http://qnib.org/data/hpcw19/7_END_2_MellanoxJourney.pdf.

[36] Parav Pandit and Dror Goldenberg. "RDMA Containers Update". In: *Proceedings of the 2018 ISC High Performance Container Workshop Conference*. Mellanox. 2018. URL: http://qnib.org/data/isc2018/roce-containers.pdf.

[37] Parav Pandit and Dror Goldenberg. "RDMA Device Isolation". In: *Proceedings of the 2019 ISC Container Workshop, Frankfurt*. Mellanox Technologies. June 2019.

[38] Jonas Pfefferle et al. "A Hybrid I/O Virtualization Framework for RDMA-Capable Network Interfaces". In: *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '15. New York, NY,

USA: Association for Computing Machinery, 2015, pp. 17–30. DOI: https://doi.org/10.1145/2731186.2731200.

[39] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. "An introduction to docker and analysis of its performance". In: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017), p. 228.

[40] *RDMA Controller*. May 2021. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/rdma.txt.

[41] *rdma-core Documentation*. Version v31.1. Nov. 3, 2020. URL: https://github.com/linux-rdma/rdma-core/tree/stable-v31/Documentation.

[42] *rdma-system(8)*. 5.12. Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man8/rdma-system.8.html.

[43] Benjamin Rothenberger et al. "ReDMArk: Bypassing RDMA Security Mechanisms". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 4277–4292. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger.

[44] Anna Kornfeld Simpson et al. "Securing RDMA for High-Performance Datacenter Storage Systems". In: *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/hotcloud20/presentation/kornfeld-simpson.

[45] John Skilbeck. *Docker: What's Under the Hood?* June 18, 2021. URL: https://www.codementor.io/blog/docker-technology-5x1kilcbow.

[46] "SOFT-RoCE: RDMA Transport in a Software Implementation". In: (2016). URL: https://www.roceinitiative.org/wp-content/uploads/2016/11/SoftRoCE_Paper_FINAL.pdf.

[47] Brent Stephens, Aditya Akella, and Michael Swift. "Loom: Flexible and efficient {NIC} packet scheduling". In: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, pp. 33–46.

[48] *Supplement to InfiniBand Architecture Specification - Annex A17: RoCEv2*. 1.2.1. InfiniBand Trade Association. Sept. 2, 2014. URL: https://cw.infinibandta.org/document/dl/7781.

[49] Mellanox Technologies. *Docker RDMA SRIOV Networking with ConnectX4 / ConnectX5 / ConnectX6*. July 30, 2020. URL: https://community.mellanox.com/s/article/Docker-RDMA-SRIOV-Networking-with-ConnectX4-ConnectX5-ConnectX6.

[50] Mellanox Technologies. *Docker RoCE MACVLAN Networking with ConnectX4 / ConnectX5*. June 17, 2020. URL: https://community.mellanox.com/s/article/docker-roce-macvlan-networking-with-connectx4-connectx5.

[51] Mellanox Technologies. *HowTo Create Docker Container Enabled with RoCE*. Mar. 1, 2020. URL: https://community.mellanox.com/s/article/howto-create-docker-container-enabled-with-roce.

[52] Mellanox Technologies. *RDMA and RoCE for Ethernet Network Efficiency Performance*. URL: https://www.mellanox.com/products/adapter-ethernet-SW/RDMA-RoCE-Ethernet-Network-Efficiency.

[53] "The Basics of Remote Direct Memory Access (RDMA) in vSphere". In: (2021). URL: https://core.vmware.com/resource/basics-remote-direct-memory-access-rdma-vsphere.

[54] *The Linux Kernel: Userspace verbs access*. Version 5.14.0-rc1. URL: https://www.kernel.org/doc/html/latest/infiniband/user_verbs.html.

[55] *Use macvlan networks*. Docker. Aug. 5, 2021. URL: https://docs.docker.com/network/macvlan/.

[56] "Why is the kernel community replacing iptables with BPF?" In: (Apr. 17, 2018). URL: https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables.

[57] Miguel G Xavier et al. "Performance evaluation of container-based virtualization for high performance computing environments". In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE. 2013, pp. 233–240.

[58] A. J. Younge et al. "A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds". In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2017, pp. 74–81. DOI: https://doi.org/10.1109/CloudCom.2017.40.

[59] Andrew J. Younge. “Containers in HPC”. In: *Workshop on NSF and DOE High Performance Computing Tools*. Sandia National Laboratories. July 11, 2019. URL: https://oaciss.uoregon.edu/NSFDOE19/talks/nsfdoe_workshop_ajy.pdf.

[60] Yang Zhao et al. “Performance of container networking technologies”. In: *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. 2017, pp. 1–6.