Iron: Isolating Network-based CPU in Container Environments

Junaid Khalid, *UW-Madison;* Eric Rozner, Wesley Felter, Cong Xu, and Karthick Rajamani, *IBM Research;* Alexandre Ferreira, *Arm Research;* Aditya Akella, *UW-Madison*

https://www.usenix.org/conference/nsdi18/presentation/khalid

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).
April 9–11, 2018 • Renton, WA, USA
ISBN 978-1-939133-01-4

Iron: Isolating Network-based CPU in Container Environments

Junaid Khalid[†] Eric Rozner[*] Wesley Felter[*] Cong Xu[*]
Karthick Rajamani[*] Alexandre Ferreira[‡] Aditya Akella[†]
[†]*UW-Madison* [*]*IBM Research* [‡]*Arm Research*

Abstract

Containers are quickly increasing in popularity as the mechanism to deploy computation in the cloud. In order to provide consistent and reliable performance, cloud providers must ensure containers cannot adversely interfere with one another. Because containers share the same underlying OS, it is more challenging to provide isolation in a container-based framework than a traditional VM-based framework. And while many schemes can isolate CPU, memory, disk, or network bandwidth in multi-tenant environments, less attention has been paid to how the time spent *processing* network traffic affects isolation on the host server. This paper shows computational overhead associated with the network stack can break isolation in container-based environments. Specifically, a container with heavy network traffic can decrease the computation available to other containers sharing the same server. We propose a scheme, called Iron, that accounts for the time spent in the networking stack on behalf of a container and ensures this processing cannot adversely impact colocated containers through novel enforcement mechanisms. Our results show Iron effectively provides isolation under realistic and adversarial conditions, limiting interference-based slowdowns as high as 6× to less than 5%.

1 Introduction

Today, containers are widely deployed in virtualized environments. Companies such as IBM, Google, Microsoft, and Amazon allow customers to deploy applications and services in containers via public clouds. In addition, serverless computing platforms [8, 31, 55] rely on containers to deploy user code [38]. Because containers share components of the underlying operating system (OS), it is critical the OS provides *resource isolation* to the container's assigned resources, such as CPU, disk, network bandwidth, and memory. Currently, control groups (or cgroups) in Linux [3] enable resource isolation by allocating, metering, and enforcing resource usage in the kernel.

Resource isolation is an important construct for both application developers and cloud providers. Recent studies indicate today's workloads are heterogeneous and do not easily fit into predetermined bucket allocations [62, 69]. Therefore, developers should be able to allocate container resources in a fine-grained manner. For this to be effective, however, a container's provisioned resources must be readily available. When resource availability is compromised due to overprovisioning or ineffective resource isolation, latency-sensitive applications can suffer from performance degradation, which can ultimately impact revenue [7, 14, 20, 49]. In serverless computing, billing is time-based [38] and insufficient resource isolation can cause users to be needlessly overcharged. Cloud providers also rely on resource isolation to employ efficient container orchestration schemes [1, 13, 69] that enable hyper-dense container deployments per server. However, without hardened bounds on container resource consumption, providers are faced with a trade-off: either underprovision dedicated container resources on each server (and thus waste potential revenue by selling spare compute to lower priority jobs) or allow loose isolation that may hurt customer performance on their cloud.

In this paper, we show containers can utilize more CPU than allocated by their respective cgroup when sending or receiving network traffic, effectively breaking isolation. Modern kernels process traffic via interrupts, and the time spent handling interrupts is often not charged to the container sending or receiving traffic. Without accurately charging containers for network processing, the kernel cannot provide hardened resource isolation. In fact, our measurements indicate the problem can be severe: containers with high traffic rates can cause colocated compute-driven containers to suffer an almost **6×** slowdown. The overhead is high because kernels perform a significant amount of network processing: from servicing interrupts, to protocol handling, to implementing network function virtualizations (e.g., switches, firewalls, rate limiters, etc). Modern datacenter line rates are fast (10-100

Gbps), and studies have shown network processing can incur significant computational overhead [35, 36, 41, 61].

Interference in datacenters is a known problem [49, 53, 64], and researchers have developed schemes to isolate CPU [11, 15, 67] and network bandwidth [42, 56, 58, 65]. In contrast, the recent study of isolating network-based processing has been limited. Prior schemes cannot be applied to modern containerized ecosystems [34] or alter the network subsystem in such a way that interrupt processing becomes less efficient [10, 25].

This paper presents Iron (Isolating Resource Overhead from Networking), a system that monitors, charges, and enforces CPU usage for processing network traffic. Iron implements a careful set of kernel instrumentations to obtain the cost of processing packets at a fine-grained level, while maintaining the efficiency and responsiveness of interrupt handling in the kernel. Iron integrates with the Linux scheduler to charge a container for its traffic. Charging alone cannot provide hardened isolation because processing traffic received by a container after it consumes its CPU allocation can break isolation. As a result, Iron implements a hardware-based scheme to drop packets destined to a container that has exhausted its allocation.

Providing isolation in containerized systems is challenging for many reasons. A container's traffic traverses the entire network stack on the server OS and thus accurate charging requires capturing variations in processing different packet types. A given solution must be computationally light-weight because line rate per-packet operations are prone to high overhead and keeping state across cores can lead to inefficient locking. Finally, limiting interference due to packet receptions is difficult because administrators may not have control over traffic sources. Iron addresses these challenges to effectively enforce isolation for network-based processing. In short, our contributions are as follows:

- A case study showing the computational burden of processing network traffic can be significant. Current cgroup mechanisms do not account for this burden, which can cause an 6× slowdown for some workloads.

- A system called Iron to provide hardened isolation. Iron's charging mechanism integrates with the Linux cgroup scheduler in order to ensure containers are properly charged or credited for network-based processing. Iron also provides a novel packet dropping mechanism to limit the effect, with minimal overhead, of a noisy neighbor that has exhausted its resource allocation.

- An evaluation showing MapReduce jobs can experience over 50% slowdown competing with trace-driven network loads and compute-driven jobs can experience a 6× slowdown in controlled settings. Iron effectively isolates and enforces network-based processing to reduce these slowdowns to less than 5%.

2 Background and Motivation

This section first describes the *interference problem*: that is, how the network traffic of one container can interfere with CPU allocated to another container. Afterwards, we place Iron in the context of past solutions and then empirically examine the impact of interference.

2.1 Network traffic breaks isolation

The interference problem occurs because the Linux scheduler does not properly account for time spent servicing interrupts for network traffic. A brief background on Linux container scheduling, Linux interrupt handling, and kernel packet processing follows.

Linux container scheduling Cgroups limit the CPU allocated to a container by defining how long a container can run (`quota`) over a time period. At a high-level, the scheduler keeps a `runtime` variable that accrues how long the container has run within the current period. When the total runtime of a container reaches its quota, the container is throttled. At the end of a period, the container's runtime is recharged to its quota. The scheduler is discussed in [67].

Linux interrupt handling Linux limits interrupt overhead by servicing interrupts in two parts: a top half (i.e., hardware interrupts) and bottom half (i.e., software interrupts). A hardware interrupt can occur at any time, regardless of which container or process is running. The top half is designed to be light-weight so it only performs the critical actions necessary to service an interrupt. For example, the top half will acknowledge the hardware's interrupt and may directly interface with the device. The top half then schedules the bottom half to execute (i.e., raises a software interrupt). The bottom half is responsible for actions that can be delayed without affecting the performance of the kernel or I/O device. Networking in Linux typically employs *softirqs* (a type of software interrupt) to implement the bottom half. Softirqs are used to transmit deferred transmissions, manage packet data structures, and navigate received packets through the network stack.

Linux's softirq handling directly leads to the interference problem. Software interrupts are checked at the end of hardware interrupt processing or whenever the kernel re-enables softirq processing. Software interrupts run in *process context*. That is, whichever unlucky process is running will have to use its scheduled time to service the softirq. Here, isolation breaks when a container has to use its own CPU time to process another container's traffic.

The kernel tries to minimize softirq handling in process context by limiting the softirq handler to run for a fixed time or budgeted amount of packets. When the budget is exceeded, softirq stops executing and schedules `ksoftirqd` to run. Ksoftirqd is a kernel thread (it does not run in process context) that services remaining softirqs. There is one ksoftirqd thread per processor. Because ksoftirqd is a kernel thread, the time it spends processing packets is not charged to any container. This breaks isolation by limiting available time to schedule other containers or allowing a container that exhausted its cgroup quota to obtain more processing resources.

Kernel packet processing Consider a “normal” packet transmission in Linux: a packet traverses the kernel from its socket to the NIC. Although this traversal is done in the kernel, it is performed in process context, and hence the time spent sending a packet is charged to the correct container. There are, however, two cases in which isolation can break on the sender. First, when the NIC finishes a packet transmission, it schedules an interrupt to free packet resources. This work is done in softirq context, and hence may be charged to a container that did not send the traffic. The second case arises when there is buffering along the stack, which can commonly occur with TCP (for congestion control and reliability) or with traffic shaping (in qdisc [2]). The packet is taken from the socket to the buffer in process context. Upon buffering the packet, however, the kernel system call exits. Software interrupts are then responsible for dequeuing the packet from its buffer and moving it down the network stack. As before, isolation breaks when softirqs are handled by ksoftirqd or charged to a container that didn’t send the traffic.

Receiving packets incurs higher softirq overhead than sending packets. Under reception, packets are moved from the driver’s ring buffer all the way to the application socket in softirq context. This traversal may require interfacing with multiple protocol handlers (e.g., IP, TCP), NFVs, NIC offloads (e.g., GRO [16]), or even sending new packets (e.g., TCP ACKs or ICMP messages). In summary, the whole receive chain is performed in softirq context and therefore a substantial amount of time may not be charged to the correct container.

2.2 Putting Iron in context

Previous works can mitigate the interference problem by designing new abstractions to account for container resource consumption or redesigning the OS. Below, Iron’s contributions are put in context.

System container abstraction In a seminal paper Banga proposed resource containers [10], an abstraction to capture and charge system resources in use by a particular activity. The work extends Lazy Receiver Processing (LRP) [25]. When a process is scheduled, a receive system call lazily invokes protocol processing in the kernel, and thus time spent processing packets is correctly charged to a process. This approach is inefficient for TCP because at most one window can be consumed between successive system calls [25], and therefore LRP employs a per-socket thread associated with each receiving process to perform asynchronous protocol processing so CPU consumption is charged appropriately.

Although LRP solves the accounting problem, the following issues must be considered. First, as the name implies, LRP only handles receiving traffic and cannot fully capture the overheads of sending traffic. Second, LRP requires a per-socket thread to perform asynchronous TCP processing[1]. Maintaining extra threads leads to additional context switching, which can incur significant overhead for processing large amounts of flows [41]. Third, the scheduler must be made aware of, and potentially prioritize, threads with outstanding protocol processing otherwise TCP can suffer from increased latencies and even drops while it waits for its socket’s thread to be scheduled. A similar notion of per-thread softirq processing was proposed in the Linux Real-Time kernel, but ultimately dropped because it increases configuration complexity and reduces performance [30].

Iron explicitly addresses the above concerns. First, Iron correctly accounts for transmissions. Second, Iron seamlessly integrates with Linux’s interrupt processing to maintain efficiency and responsiveness. In Linux, all of a core’s traffic is processed by that core’s softirq handler. Processing interrupts in a shared manner, rather than in a per-thread manner, maintains efficiency by minimizing context switching. Additionally, by servicing hardware interrupts in process context, protocol processing is performed responsively. Linux’s design, however, directly leads to the interference problem. Therefore, one contribution of our work is showing accurate accounting for network processing is possible even when interrupt handling is performed in a shared manner.

Redesigning the OS Library OSes [28, 48, 57, 60] redesign the OS by moving network protocol processing from the kernel to application libraries. In these schemes, packet demultiplexing is performed at the lowest level of the network stack: typically the NIC directly copies packets to buffers shared with applications. Since applications process packets from their buffers directly, network-based processing is correctly charged.

Library OSes have numerous practical concerns, however. First, these works face similar challenges as LRP with threaded protocol processing. Second, explicitly removing network processing from the kernel can make management difficult. In multi-tenant datacenters, servers host services such as rate limiting, virtual networking, billing, traffic engineering, health monitoring, and security. With a library OS, admin-defined network processing must be performed in the NIC or in user-level software.

[1] Banga’s design uses a per-process asynchronous thread

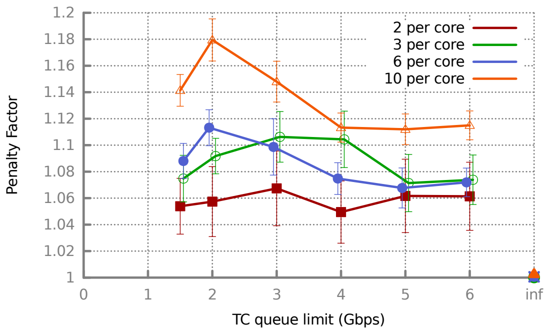


Figure 1: Penalty factor of UDP senders.

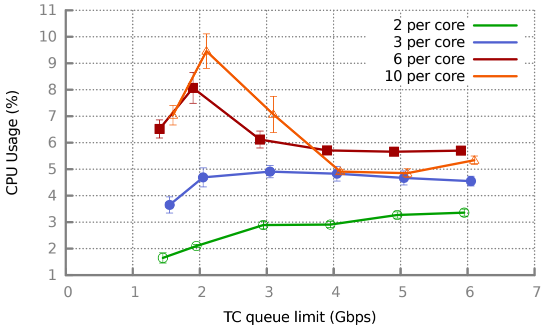


Figure 2: ksoftirqd overhead with UDP senders.

Neither approach is ideal. Application libraries linked by developers make it difficult for admins to insert policies and functionalities at the host. For example, an admin's ability to perform traffic shaping or simply configure a TCP stack may be limited. Furthermore, porting network services to user-level requires every NFV application to track, charge, and enforce network processing to mitigate interference. NIC-based techniques can cost more (to upgrade hosts), scale more poorly in the number of flows and services, and be less flexible and harder to configure than software. And while NICs are becoming more flexible [76], it is likely network management will be dictated by a combination of admin-controlled software and hardware in the future. As such, Iron can help track and enforce software-based network processing. Finally, adapting Library OSes to support multi-tenancy and replacing currently deployed ecosystems can have a high barrier to entry for providers and customers.

2.3 Impact of network traffic

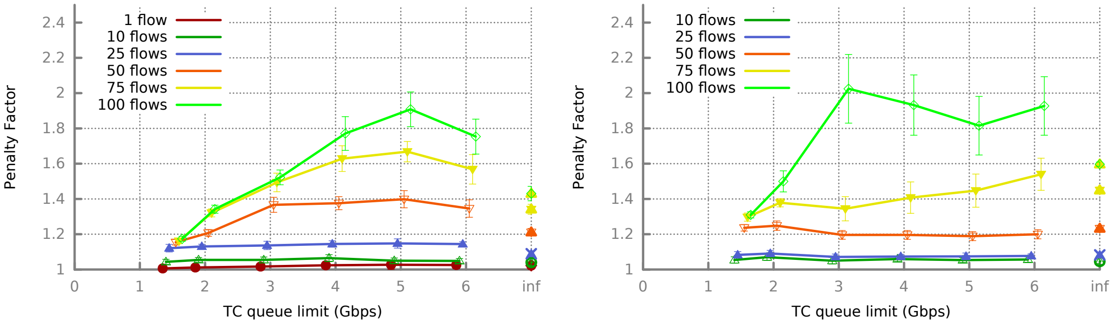
In this section, a set of controlled experiments quantifies the impact of both UDP and TCP network processing on isolation in containerized environments.

Methodology In each experiment, n containers are allocated per core, where n varies from 2, 3, 6, or 10. Each container is configured to obtain an equal share of the core (i.e., $quota = period/n$). This allocation is replicated over all cores. NICs are 25 Gbps, and Section 4 further details methodology. One container per core, denoted the *victim*, runs a CPU-intensive `sysbench` workload [46]. The time to complete each victim's workload is measured under two scenarios. In the first scenario all non-victim containers, henceforth denoted *interferers*, also run `sysbench`. This serves as a baseline case. In the second scenario, the interferers run a simple network flooding application that sends as many back-to-back packets as possible. The victim's completion time is measured under both scenarios, and a *penalty factor* indicates the fraction of time the victim's workload takes when competing with traffic versus competing with `sysbench`. Penalty factors greater than one indicate isolation is broken because traffic is impacting the victim in an adverse way.

For the reception tests, containers are allocated on a single core and all NIC interrupts are serviced on the same core to ensure cores without containers do not process traffic. As before, the victim container runs `sysbench`, but the interferers now run a simple receiver. A multi-threaded sender varies its rate to the core, using 1400 byte packets and dividing flows evenly amongst the receivers. All results are averaged over 10 runs.

UDP senders These results show the impact when the interfering containers flood 1400 byte UDP traffic. Studies have shown rate limiters can increase computational overhead [61], so the penalty factor is measured when no rate limiters are configured and also when hierarchical token bucket (HTB) [2] is deployed for traffic shaping.

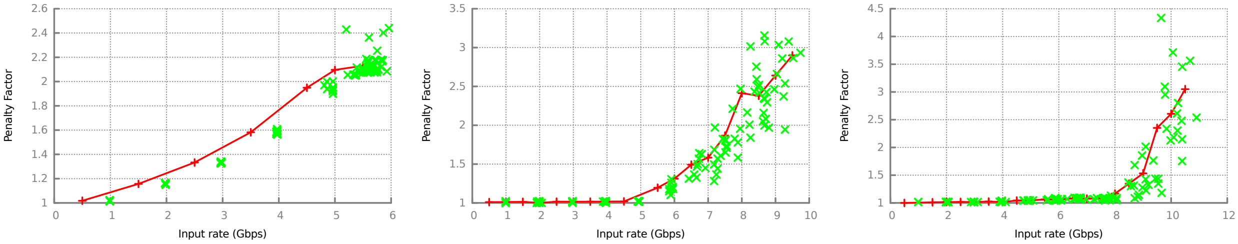
Figure 1 presents the results. Lines denote how many containers are allocated on a core, the x-axis denotes the rate limit imposed on a core, and the y-axis indicates the penalty factor. With n containers per core, each container receives $\frac{1}{n}^{th}$ of the bandwidth allocated to the core. The right-most point labeled "inf" is when no rate limiter is configured. We note the following trends. First, there is no penalty factor with no rate limiting because the application demands are lower than the link bandwidth, so there is no queuing at the NIC. Second, rate limiting causes penalty factors as high as 1.18. The summed application demands can be higher than the imposed rate limit on each core, which means packets are queued in the rate limiter. Softirq handling interferes with the processing time of the victims, leading to high penalty factors. Third, HTB experiences a relatively higher penalty for 1-3 Gbps. When rate limits are 4 Gbps and above, the rate limiter does not shape traffic because senders are CPU-bound and cannot generate more than 4 Gbps of traffic demand. Isolation still breaks because rate limiters maintain state and perform locking (this overhead was also witnessed



(a) 2 containers per core

(b) 10 containers per core

Figure 3: Penalty factor of victims with TCP senders.



(a) 1 receiver

(b) 5 receivers

(c) 9 receivers

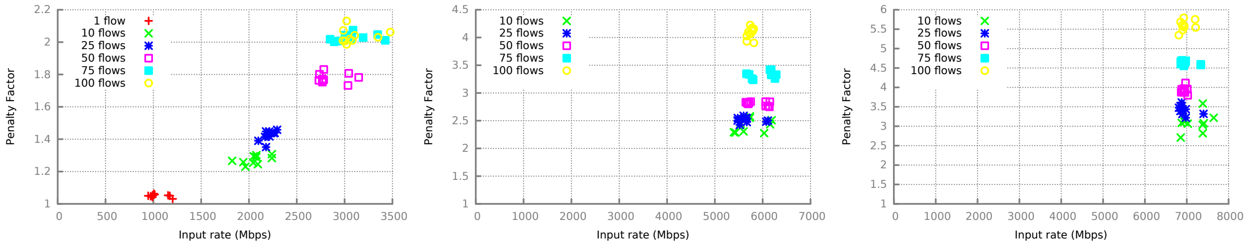
Figure 4: Penalty factor when there are 10 containers on 1 core. $i = 1, 5, 9$ of the containers are UDP receivers.

in [43]). For rates below 4 Gbps, senders generate more traffic than the enforced rate and higher overheads occur.

Figure 2 shows the CPU usage of ksoftirqd (on core 0) for the experiment in Figure 1. The trends roughly correspond to the penalty factor overhead. Time spent in ksoftirqd is not attributed to any process, which means that time cannot be issued to other containers. This increases the time it takes for the victim workload to complete. To understand the remaining penalty, we instrumented a run with `perf` [5]. With 10 containers per core and 2 Gbps rate limit, the victim spent 6.99% of its scheduled time servicing softirqs even though it sent no traffic.

TCP senders Figure 3 shows TCP sender performance for 2 and 10 containers per core. Different from the UDP results, the number of flows per core is varied, and flows are divided equally amongst all sending containers on a core. We note the following trends. First, TCP overheads are higher than UDP overheads– in the worst case, the overhead can be as high as 1.95×. TCP overheads are higher because TCP senders receive packets, i.e., ACKs, and also buffer packets at the TCP layer. Both ACK processing and pushing buffered packets to the NIC are completed via softirqs. Therefore, no rate limiting has higher overhead in TCP than UDP. The second interesting trend is overheads increase as the number of flows increase. This occurs for two reasons. First, the number of TCP ACKs increase with flows, and in general, there exists more protocol processing with more flows. Second, a single TCP flow can adapt to the rate limiter, but multiple flows create burstier traffic patterns that increase queuing at the rate limiter.

UDP receivers Figure 4 shows the UDP receiver results. Ten containers are allocated on the core, and if i containers receive UDP traffic, then $10 - i$ containers run `sysbench`. The sender increases its sending rate from 1 Gbps to 12 Gbps at 1 Gbps increments. For each sending rate, 10 trials are run. Each green dot represents the result of a trial. The x-axis indicates the input rate to the core, which can differ from the sending rate due to drops. The red line, provided for reference, averages the penalty factor in 500 Mbps buckets. We varied the number of receivers from 1 to 9, but only show 1, 5, and 9 receivers in the interest of space. We note the following trends. First, the penalty factor for receiving UDP is higher than sending UDP. Packets traverse the whole network stack in softirq context and therefore overheads are larger. Next, as more of the core is allocated to receive (as i increases), the rate at which the server can process traffic increases. As the rate of incoming traffic increases, so does the penalty factor. Under high levels of traffic, the



(a) 1 receiver (b) 5 receivers (c) 9 receivers

Figure 5: Penalty factor when there are 10 containers on a 1 core. $i = 1,5,9$ of the containers are TCP receivers.

overheads from softirqs cause the victim to take almost 4.5× longer.

TCP receivers Figure 5 shows the results when interfering containers receive TCP traffic. Different from UDP experiments, TCP senders are configured to send as much as they can. TCP will naturally adapt its rate when drops occur (from congestion control) or when receive buffers fill (from flow control). As before, the penalty factor increases as the input rate increases and also when the number of flows increase. In the worst case, interference from TCP traffic causes the victim to take almost six times longer. To further understand this overhead, we instrument `sysbench` with `perf` for nine TCP receivers and 100 flows. Here, ksoftirqd used 54% of the core and `sysbench` spent 60% of its time servicing softirqs. This indicates that isolation techniques must capture softirq overhead in both ksoftirqd and process context.

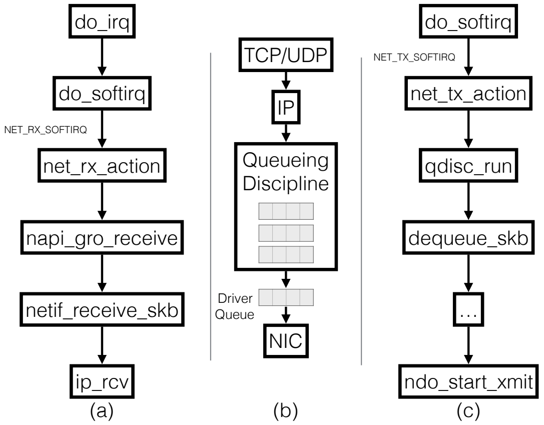


Figure 6: Networking in Linux: (a) subset of receive call stack, (b) send architecture, (c) subset of send call stack.

3 Design

This section details Iron's design. Iron first *accounts* for time spent processing packets in softirq context. After obtaining packet costs, Iron integrates with the Linux scheduler to charge or credit containers for softirq processing. When a container's runtime is exhausted, Iron *enforces* hardened isolation by throttling containers and dropping incoming packets via a hardware-based method.

3.1 Accounting

This section outlines how to obtain per-packet costs in order to ensure accounting is accurate. First, receiver-based accounting is detailed, followed by sender-based accounting. Afterwards, we describe how to assign packets to containers and the state used for accounting.

Receiver-based accounting In Linux, packets traverse the network stack through a series of nested function calls. For example, the IP handler of a packet will directly call the transport handler. Therefore, a function low in the call stack can obtain the time spent processing a packet by subtracting the function start time from the function end time. Figure 6a shows a subset of Linux's receive call stack. Iron instruments `netif_receive_skb` to obtain per-packet costs because it is the first function that handles individual packets outside the driver, regardless of transport protocol[2].

Obtaining the time difference is nontrivial because the kernel is preemptable and functions in the call tree can be interrupted at any time. To ensure only the time spent processing packets is captured, Iron relies on scheduler data. The scheduler keeps the cumulative execution time a thread has been running (*cumtime*), as well as the time a thread was last swapped in (*swaptime*). Coupled with the local clock (*now*), the start and end times can be calculated as: $time = cumtime + (now - swaptime)$.

Besides per-packet costs, there is also a fixed cost associated with processing traffic. That is, there are overheads for entering the function that processes hardware interrupts (`do_IRQ`), processing softirqs, and performing `skb` garbage collection. In Iron, these overheads are lumped together and assigned to packet costs in a weighted fashion. In Linux, six types of softirqs are processed by the

[2] TCP first traverses GRO, but we instrument here for uniformity

softirq handler (`do_softirq`): HI, TX, RX, TIMER, SCSI, and TASKLET. For each interrupt, we obtain the total `do_IRQ` cost, denoted H, and the cost for processing each specific softirq (denoted S_{HI}, S_{TX}, etc). Note software interrupts are processed at the end of a hardware interrupt, so $H > \sum_i S_i$. The overhead associated with processing an interrupt is defined as: $O = H - \sum_i S_i$ and the fair share of the receive overhead within that interrupt is: $O_{RX} = O\frac{S_{RX}}{\sum_i S_i}$. Last, O_{RX} is evenly split amongst packets processed in a given `do_softirq` call to obtain a fixed charge added to each packet.

Finally, we note this scheme is effective in capturing TCP overhead. That is, Iron gracefully handles TCP ACKs and TCP buffering. A TCP flow is handled within a single thread, so when data is received, the thread directly calls a function to send an ACK. When the ACK function returns, the received data continues to be processed as normal. Therefore, ACK overhead is captured by our start and end timestamps. Buffering is also handled correctly. Say packet $i-1$ is lost and thus received packet i is buffered. When retransmitted $i-1$ is received the gap in sequence numbers is filled and TCP will push up the packets to the socket. Correct charging occurs because the cost of moving packet i from the buffer to the socket is captured in the cost of retransmitted packet $i-1$.

Sender-based accounting When sending packets, the kernel has to obtain a lock on a NIC queue. Obtaining a lock on a per-packet basis has high overhead, so packets are often batched for transmission in Linux [18]. Therefore, Iron measures the cost of sending a batch and then charges each packet within the batch for an equal share of the batch cost. The `do_softirq` function calls `net_tx_action` to process transmit softirqs (refer to Figure 6b,c). Then `net_tx_action` calls into the qdisc layer to retrieve packets. Multiple qdisc queues can be dequeued and each queue may return multiple packets. As a result, a linked list of skbs is created and sent to the NIC. Similar to the receiver, `net_tx_action` obtains a start and end time for sending the batch, and O_{TX} is obtained to split the transmission's fixed overheads. Overheads are calculated per core because HTB is work conserving and may dequeue a packet on a different core than it was enqueued.

Container mapping and accounting data structures Iron must identify the container a packet belongs to. On the sender, an skb is associated with its cgroup when enqueued in the qdisc layer. On the receiver, Iron maintains a hash table on IP addresses that is filled when copying packets to a socket.

In Iron, each process maintains a local (per-core) list of packets it processed in softirq context and their individual costs. The per-process structures are eventually merged into a global per-cgroup structure. Iron does this in a way that does not increase locking by merging state when the scheduler obtains a global lock. The per-cgroup structure maintains a variable (`gained`) that indicates if a cgroup should be credited for network processing. Section 3.2 details data structure use.

Algorithm 1 Global runtime refill at period's end

```
1: if gained $> 0$ then
2:     runtime $\leftarrow$ runtime + gained
3:     gained $\leftarrow 0$
4: end if
5: if cgroup_idled() and runtime $> 0$ then
6:     runtime $\leftarrow 0$
7: end if
8: runtime $\leftarrow$ quota + runtime
9: set_timer(now + period)
```

3.2 Enforcement

This subsection shows how isolation is enforced. Isolation is achieved by integrating accounting data with CPU allocation in Linux's CFS scheduler [67] and dropping packets when a container becomes throttled.

Scheduler integration The CFS scheduler implements CPU allocation for cgroups via a hybrid scheme that keeps both local (i.e., per core) and global state. Containers are allowed to run for a given `quota` within a `period`. The scheduler minimizes locking overhead by updating local state on a fine-grained level and global state on a coarse-grained level. At the global level a `runtime` variable is set to `quota` at the beginning of a period. The scheduler subtracts a `slice` from `runtime` and allocates it to a local core. The `runtime` continues to be decremented until either it reaches zero or the period ends. Regardless, at the end of a period `runtime` is refilled to the quota.

On the local level, a `rt_remain` variable is assigned the `slice` intervals pulled from the global `runtime`. The scheduler decrements `rt_remain` as a task within the cgroup consumes CPU. When `rt_remain` hits zero, the scheduler tries to obtain a slice from the global pool. If successful, `rt_remain` is recharged with a slice and the task can continue to run. If the global pool is exhausted, the local cgroup gets *throttled* and its tasks are no longer scheduled until the period ends.

Iron's global scheduler is presented in Algorithm 1. A global variable `gained` tracks the time a container should get back because it processed another container's softirqs. Line 2 adds `gained` to `runtime`. Next, `runtime` is reset to 0 if the container didn't use its previous allocation because it was limited by its demand (lines 5-7), preserving a CFS policy that disallows unused cycles to be accumulated for use in subsequent periods.

Algorithm 2 Local runtime refill

```
1:  amount $\leftarrow$ 0
2:  min_amount $\leftarrow$ slice - rt_remain
3:  **if** cpuusage $> 0$ **then**
4:      **if** cpuusage $>$ gained **then**
5:          runtime $\leftarrow$ runtime - (cpuusage - gained)
6:          gained $\leftarrow$ 0
7:      **else**
8:          gained $\leftarrow$ gained - cpuusage
9:      **end if**
10: **else**
11:     gained $\leftarrow$ gained + abs(cpuusage)
12: **end if**
13: cpuusage $\leftarrow$ 0
14: **if** runtime $= 0$ and gained $> 0$ **then**
15:     refill $\leftarrow$ min(min_amount, gained)
16:     runtime $\leftarrow$ refill
17:     gained $\leftarrow$ gained - refill
18: **end if**
19: **if** runtime $> 0$ **then**
20:     amount $\leftarrow$ min(runtime, min_amount)
21:     runtime $\leftarrow$ runtime - amount
22: **end if**
23: rt_remain $\leftarrow$ rt_remain + amount
```

Last, line 8 refills `runtime`. Note, the runtime input can be negative when a container exceeds its allocated time by sending or receiving too much traffic.

Iron's local algorithm is listed in Algorithm 2. The scheduler invokes this function when `rt_remain` ≤ 0 and after obtaining appropriate locks. The `cpuusage` variable is added to maintain local accounting: positive values indicate the container needs to be charged for unaccounted networking cycles and negative values indicate the container needs a credit for work it did on another container's behalf. Lines 3-9 cover when a container is to be charged, trying to take from `gained` if possible. Lines 10-12 cover the case when a container is to be credited, so `gained` is increased. Lines 14-18 cover a corner case where the runtime may be exhausted, but some credit was accrued and can be used. Lines 19-22 are unchanged: they ensure the container has global runtime left to use. If not, then `amount` remains 0. Line 23 updates the new `rt_remain` by `amount`.

Dropping excess packets While scheduler-based enforcement improves isolation, packets still need to be dropped so a throttled container cannot accrue more network-based processing. Iron does not explicitly drop packets at the sender because throttled containers already cannot generate more outgoing traffic. There exists a corner case when a container has some runtime left and sends a large burst of packets. Currently, the scheduler charges this overage on the next quota refill. We did implement a *proactive* charging scheme that estimates the cost of packet transmission, charges it up-front, and drops packets if necessary. This scheme didn't substantially affect performance, however.

Dropping the receiver's excess packets is more important because a throttled receiver may continue to receive traffic, hence breaking isolation. Iron implements a hardware-based dropping mechanism that integrates with current architectures. Today, NICs insert incoming packets into multiple queues. Each queue has its own interrupt that can be assigned to specified cores. To improve isolation, packets are steered to the core in which their container runs via advanced receive flow steering [39] (FlexNIC [45] also works). Upon reception, the NIC DMAs a packet to a ring buffer in shared memory. Then, the NIC generates an IRQ for the queue, which triggers the interrupt handler in the driver. Modern systems manage network interrupts with NAPI [4]. Upon receiving a new packet, NAPI disables hardware interrupts and notifies the OS to schedule a polling method to retrieve packets. Meanwhile, additionally received packets are simply put in the ring buffer by the NIC. When the kernel polls the NIC, it removes as many packets from the ring buffer as possible, bounded by a budget. NAPI polling exits and interrupt-driven reception is resumed when the number of packets removed is less than the budget.

Our hardware-based dropping mechanism works as follows. First, assume the NIC has one queue per container. Iron augments the NAPI queue structure with a map from a queue to its container (i.e., task_group). When the scheduler throttles a container, it modifies a boolean in task_group. Different from default NAPI, Iron does not poll packets from queues whose containers are throttled. From the kernel's point of view, the queue is stripped from the polling list so that it isn't constantly repolled. From the NIC's point of view, the kernel is not polling packets from the queue, so it stays in polling mode and keeps hardware interrupts disabled. If new packets are received, they are simply inserted into the ring buffer. This technique effectively mitigates receiving overhead because the kernel is not being interrupted or required to do any work on behalf of the throttled container. When the scheduler unthrottles a container, it resets its boolean and schedules a softirq to process packets that may be enqueued.

As a slight optimization, Iron can also drop packets before a container is throttled. That is, if a container is receiving high amounts of traffic and the container is within $T\%$ of its quota, packets can be dropped. This allows the container to use some of its remaining runtime to stop a flood of incoming packets.

Hardware-based dropping is effective when there are a large number of queues per NIC. Even though NICs are

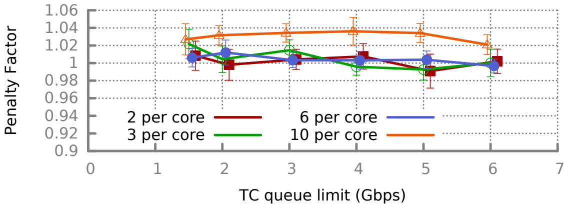


Figure 7: Performance penalty of victim with UDP senders. Compare to Figure 1.

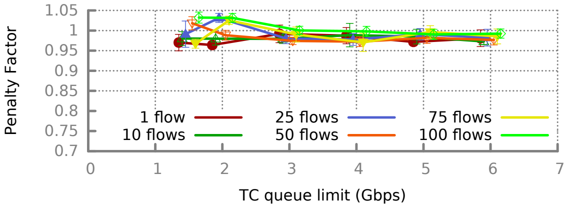


Figure 8: Performance penalty of victim with TCP senders. Compare to Figure 3a.

increasingly outfitted with extra queues (e.g., Solarflare SFN8500-series NICs have 2048 receive queues), in practice the number of queues may not equal the number of containers. Iron can allocate a fixed number of queues per core and then dynamically map problematic containers onto their own queue. Containers without heavy traffic can incur a software-based drop by augmenting the __netif_receive_skb function early in the softirq call stack. This dynamic allocation scheme draws inspiration from SENIC [61], which uses a similar approach to scale NIC-based rate limiters. Alternatively, containers can be mapped to queues based on prepurchased bandwidth allocations.

4 Evaluation

This section evaluates the effectiveness of Iron. First, a set of macrobenchmarks show Iron isolates controlled and realistic workloads. Then, a set of microbenchmarks investigates Iron's overhead and design choices.

Methodology The tests are run on Super Micro 5039MS-H8TRF servers with Intel Xeon E3-1271 CPUs. The machines have four cores, with hyper-threading disabled and CPU frequency fixed to 3.2 Ghz. The servers are equipped with Broadcom BCM57304 NetXtreme-C 25 Gbps NICs (driver 1.2.3 and firmware 20.2.25/1.2.2). The servers run Ubuntu 16.04 LTS with Linux kernel 4.4.17. The NICs are set to 25 Gbps for UDP and 10 Gbps for TCP (we noticed instability with TCP at 25 Gbps).

We use lxc to create containers and Open Virtual Switch as the virtual switch. Simple UDP and TCP sender and receiver programs create network traffic. The sysbench's CPU benchmark is used to measure the computational overhead from competing network traffic. Rate limiters are configured with default burst settings.

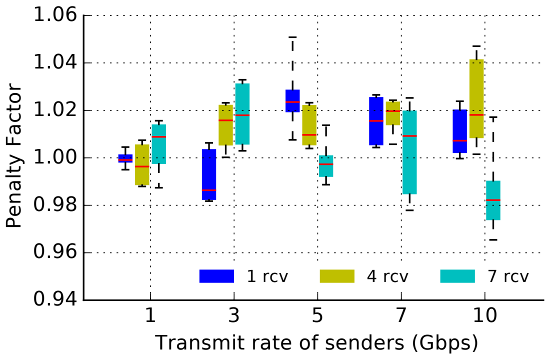


Figure 9: Performance penalty of victim when there are 8 containers on a core. i of the containers are UDP receivers.

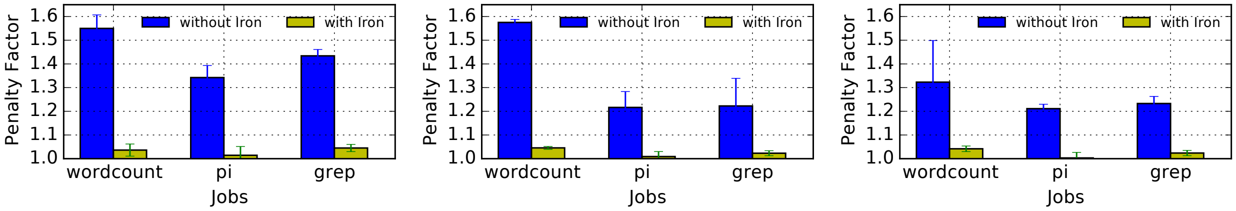
4.1 Macrobenchmarks

Sender-side experiments We run the same experiments in Section 2 to evaluate how well Iron isolates sender interference. Figure 7 shows the impact of UDP senders on sysbench. Note this experimental setup is the same as Figure 1. Iron obtains average penalty factors less than 1.01 for 2, 3, and 6 containers, as compared penalty factors as high as 1.11 without Iron. With 10 containers, Iron's penalty factor remains below 1.04, a significant decrease from the maximum of 1.18 without Iron.

Figure 8 shows the performance of Iron with TCP senders, and can be compared to Figure 3a. The maximum penalty factor experienced by Iron is 1.04, whereas the maximum penalty factor without Iron is 1.85. These results show Iron can effectively curtail interference from network-based processing associated with sending traffic.

Receiver-side experiments We rerun the experiments in Section 2 to evaluate how well Iron isolates receiver interference. Even though our NICs support more than eight receive queues, we were unable to modify the driver to expose more queues than cores. Therefore, different from Section 2, a single core is allocated with 8 containers, instead of 10. In these experiments, the number of receiver containers varies from 1, 4, or 7. Containers that are not receivers run an interfering sysbench workload. For the UDP experiments, the hardware-based enforcing mechanism was employed, while the TCP experiments utilize our software-based enforcing mechanism.

Figure 9 shows the impact of UDP receivers. The x-axis shows aggregated traffic rate at the sender. This is different from the graphs in Section 2 because Iron drops



(a) with TeraSort (b) with data-mining workload (c) with web-search workload

Figure 10: Penalty factor when MapReduce jobs share resources with other workloads.

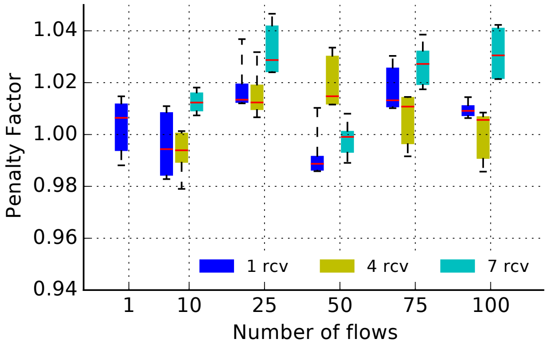


Figure 11: Performance penalty of victim when there are 8 containers on a core. i of the containers are TCP receivers.

packets when container quotas are exceeded, causing received rates to converge. Each number of receivers is indicated by a different bar color. The error bars represent the 5% and 95%. The height of the bars indicate the 25% to 75% and the red horizontal line within each bar is the median. In the previous results without Iron, penalty factors ranged from maximums of 2.45 to 4.45. With Iron, the median penalty factor ranges between 0.98 and 1.02 and never exceeds 1.05. The penalty factor can be lower than 1 when Iron overestimates hard interrupt overheads: overheads, including those occurring after softirq processing, are approximated by using measured values from previous cycles.

Figure 11 shows when interfering containers receive TCP traffic. Unlike UDP, TCP adapts its rate when packet drops occur. Therefore, the software-based rate limiter is effective in reducing interference. In Section 2, the maximum penalty factor ranged from 2.2 to 6. However, with Iron, penalty factors do not exceed 1.05.

Realistic applications Here we evaluate the impact of interference on real applications. We run the experiment on a cluster of 48 containers spread over 6 machines. Each machine has 8 containers (2 per core). The cluster is divided into two equal subclusters such that a container in a subcluster does not share the core with a container from the same subcluster. HTB evenly divides bandwidth between all containers on a machine.

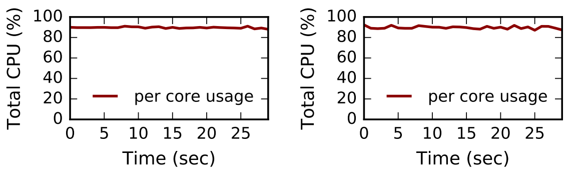
Three MapReduce applications serve as the victims: *pi* computes the value of pi, *wordcount* counts word frequencies in a large file, and *grep* searches for a given word in a file. Three different trace-based interferers run on the other subcluster: the shuffle phase of a *TeraSort* job with a 115GB input file, a *web-search* workload [7] and a *data-mining* workload [6]. For the latter two workloads, applications maintain long-lived TCP connections to every other container in the subcluster, sequentially sending messages to a random destination with sizes distributed from each trace. Figure 10 shows the impact of interference on real applications. Iron obtains an average penalty factor less than 1.04 over all workloads, whereas the average penalty factor ranges from 1.21-1.57 without Iron. These results show Iron can effectively eliminate interference that arises in realistic conditions.

4.2 Microbenchmarks

This subsection evaluates Iron's overhead, the usefulness of runtime packet cost calculation, and the benefits of hardware-based packet dropping.

Performance overhead To measure how accurately Iron limits CPU usage, we allocated 3 containers on a core with each container having 30% of the core. One of the containers ran `sysbench`, while the other two were UDP senders. Figure 12a shows the total CPU used by all containers over a 30 second window. On average, the consumed CPU was around 90.02%. In an ideal case, no more than 90% of the CPU should be utilized. This indicates Iron does not have high overhead in limiting cgroup CPU allocation to its respective limits. We also ran the experiment with a UDP receiver, as shown in Figure 12b. On average Iron ensures an idle CPU of 10.07%, which again shows the effectiveness of our scheme.

Next we analyzed if Iron hurts a network-intensive workload. We instrumented a container to receive traffic and allowed it 100% of the core. Then, at the sender, we generated UDP traffic at 2 Gbps. Using `mpstat`, we measured the CPU consumed by the receiver. The



(a) CPU usage with senders (b) CPU usage with receivers

Figure 12: CPU overhead benchmarks.

Packet type	Average packet cost (usec)
UDP	0.706
TCP	1.670
GRE Tunnel	1.184

Table 1: Average packet processing cost at the receiver.

receiver consumed 35% of the core and received traffic at 1.93 Gbps. Next, we ran the same experiment with the receiver, but this time limited the container to 35% of the core. With Iron limiting the CPU usage, the receiver received traffic at 1.90 Gbps (and used no more than 35% of the CPU). This indicates the overhead of Iron on network traffic is minimal. We ran a similar experiment with the UDP sender and observed no degradation in traffic rate. Unlike the receiver, if a sender is out of CPU cycles it will be throttled, thus generating no extra traffic.

Packet cost variation A simple accounting scheme may charge a fixed packet cost and is likely ineffective because packet processing costs vary significantly at the receiver. Table 1 shows the average packet cost for three classes of traffic. TCP requires bookkeeping for flow control and reliability and results in higher costs than UDP. UDP packets encapsulated in GRE experience extra cost because those packets traverse the stack twice.

Dropping mechanism We compared the impact of software-based versus hardware-based dropping. The UDP sending rate is varied to a receiver with 8 containers on a core (7 are receivers). As shown in Figure 13, both approaches mitigate interference when traffic rates are low. However, when rates are high, the median penalty factor of the software-based rate limiter increases to 1.19, with the 95% approaching a penalty factor of 1.6. The hardware rate-limiter maintains a near-constant penalty factor, regardless of rate.

5 Related Work

Here we augment Section 2.2 to further detail prior art.

Isolation of kernel-level resources Many studies have examined how colocated computation suffers from interference [23, 49, 53, 64, 74]. As such, providing resource isolation to underlying server hardware has been a rich area of research. For example, researchers have investigated how to isolate CPU time [11, 15, 67], processor caches [27, 44], memory bandwidth [40, 71], energy [26], and storage [50, 54, 66, 70, 72]. These schemes address problems orthogonal to our work, and none can be generalized to solve the problem Iron solves. While Iron focuses on network-based interrupt processing, Iron's high-level principles of annotating and measuring container-based interrupt overhead can be applied to other interrupts (e.g., timers, storage, etc). For example, tags designed for scheduling I/O storage requests [32, 72] can help account for per-container storage processing overheads. Like Iron, these overheads can be integrated with the Linux CFS scheduler. While developing specific, low-overhead enforcement schemes for other interrupts remains future work, modifying the I/O block scheduler or utilizing software-defined storage mechanisms [66] are promising starts for storage.

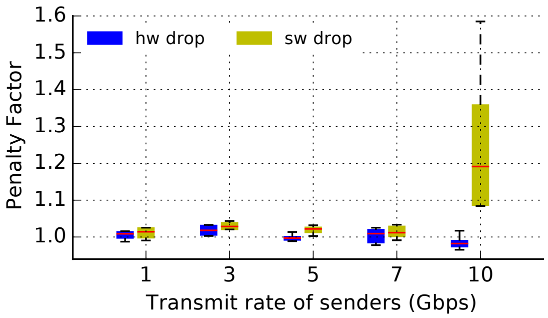


Figure 13: Impact of software and hardware-based packet dropping mechanisms on penalty factor for 7 receivers.

A large class of research allocates network bandwidth [9, 33, 42, 56, 58, 59, 63, 65] or isolates congestion control [19, 37] in shared datacenters. In short, these schemes affect network performance but do nothing to control network-based processing time, and thus are complimentary to Iron. Last, some schemes isolate dataplane and application processing on core granularity [12, 41], but do not generalize to support many containers per core nor explicitly study the interference problem.

Resource management and isolation in cloud Determining how to place computation within the cloud has received significant attention. For example, Paragon [21] schedules jobs in an interference-aware fashion to ensure service-level objectives can be achieved. Several other schemes, such as Borg [69], Quasar [22], Heracles [49], and Retro [51] can provision, monitor, and control resource consumption in colocated environments to ensure performance is not degraded due to interference. Iron is largely complementary to these schemes. By provid-

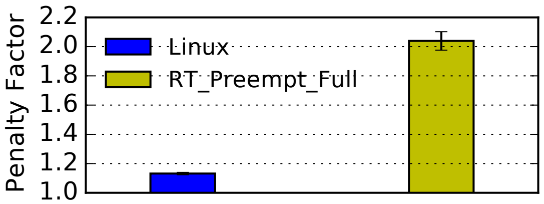


Figure 14: Performance penalty with RT Linux.

ing hardened isolation, Iron allows resource managers to make more informed decisions, so network-heavy jobs cannot impact colocated processor-heavy jobs.

VM network-based accounting Gupta's scheme accounts for processing performed in device drivers for an individual VM [34]. The scheme measures VM-based resource consumption in the hypervisor, integrates with the scheduler to charge for usage, and limits traffic when necessary. Iron differs in many regards. Iron provides performance isolation in container-based environments, instead of VM-based environments. The difference is significant because packets consume more processing time with containers because the network stack lies within the server's kernel, and not the VM's. Gupta relies on a fixed cost to charge for packet reception and transmission, but our results show packet costs vary significantly. Furthermore, because container-based environments incur more processing overhead for traffic, it is important that received traffic is discarded efficiently when necessary. Hence, Iron contains a novel hardware-based enforcement scheme, whereas Gupta's work relies on software.

Shared NFV infrastructure Many works study how to allocate multiple NFVs and their resources on a server [24, 29, 47, 52, 68]. Similar to library OSes, NFV servers require kernel bypass for latency and control. As discussed, kernel bypass approaches cannot easily generalize to solve the interference problem in multi-tenant containerized clouds.

Real-time kernel Real-time (RT) kernels typically aren't used for multi-tenancy, but some RT OSes redesign interrupt processing in a way that could mitigate the interference problem. For example, RT Linux patches the OS so the only type of softirq served in a process's context are those which originated within that process [17]. While this patch doesn't help with receptions, it prevents a container with *no* outgoing traffic from processing interrupts from another container's outgoing traffic. To understand this solution, we ran an experiment with 2 Gbps rate limit and 6 equally-prioritized containers per core: one `sysbench` victim and 5 interferers that flood outgoing UDP traffic. Figure 14 shows the penalty factor for normal Linux and RT Linux (RT_Preempt_Full). The penalty factor of RT Linux is significantly higher than Linux because in RT Linux the victim doesn't process interrupts in its context. Instead, interrupt processing is moved to kernel threads. The processing time used by the kernel threads reduces the time available to the victim. Additionally, RT Linux tries to minimize softirq processing latency and `perf` shows the victim experiences 270× more involuntary context switches as compared to Linux.

Finally, Zhang [75] proposes a RT OS that increases predictability by scheduling interrupts based on process priority and accounting for time spent in interrupts. Zhang's accounting scheme has up to 20% error [75], likely because it is coarse-grained in time and does not use actual, per-packet costs. Overheads in Iron are less than 5% because its accounting mechanism is immediately responsive to actual, per-packet costs. In addition, Iron comprehensively studies the interference problem and introduces enforcement schemes.

Microsoft Windows The scheduler in Windows does not count time spent processing interrupts towards a thread's execution time [73]. This is not sufficient to totally mitigate the interference problem because time spent in interrupt and deferred interrupt processing is not charged to an appropriate thread. Therefore, the large number of cycles consumed by kernel packet processing leave less cycles available to colocated, CPU-heavy threads.

6 Conclusion

This paper presents Iron, a system providing hardened isolation for network-based processing in containerized environments. Network-based processing can have significant overhead, and our case study shows a container running a CPU-intensive task may suffer up to a 6× slowdown when colocated with containers running network-intensive tasks. Iron enforces isolation by accurately measuring the time spent processing network traffic in softirq context within the kernel. Then, Iron relies on an enforcement algorithm that integrates with the Linux scheduler to throttle containers when necessary. Throttling alone is insufficient to provide isolation because a throttled container may receive network traffic. Therefore, Iron contains a hardware-based mechanism to drop packets with minimal overhead. Our scheme seamlessly integrates with modern Linux architectures. Finally, the evaluation shows Iron reduces overheads from network-based processing to less than 5% for realistic and adversarial workloads.

Acknowledgements We thank the reviewers and our shepherd Boon Thau Loo. This work is supported by the National Science Foundation (grants CNS-1302041, CNS-1330308, CNS1345249, and CNS-1717039), and Aditya Akella is also supported by gifts from VMWare, Huawei, and the UW-Madison Vilas Associates.

References

[1] Docker swarm. https://github.com/docker/swarm. Accessed: 2017-09-25.

[2] Linux advanced routing and traffic control howto. http://lartc.org/lartc.html. Accessed: 2017-09-25.

[3] Linux control groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. Accessed: 2017-09-21.

[4] Networking napi. https://wiki.linuxfoundation.org/networking/napi. Accessed: 2017-09-25.

[5] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2017-09-21.

[6] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., MATUS, F., PAN, R., YADAV, N., VARGHESE, G., ET AL. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM* (2014).

[7] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[8] AMAZON WEB SERVIES, INC. AWS Lambda: Serverless computing. https://aws.amazon.com/lambda/.

[9] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 242–253.

[10] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *OSDI* (1999), vol. 99, pp. 45–58.

[11] BARTOLINI, D. B., SIRONI, F., SCIUTO, D., AND SANTAMBROGIO, M. D. Automated fine-grained cpu provisioning for virtual machines. *ACM Trans. Archit. Code Optim. 11*, 3 (July 2014), 27:1–27:25.

[12] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (CO, 2014), USENIX Association, pp. 49–65.

[13] BREWER, E. A. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 167–167.

[14] BRUTLAG, J. Speed Matters for Google Web Search. Tech. rep., 2009. https://services.google.com/fh/files/blogs/google_delayexp.pdf.

[15] CHERKASOVA, L., GUPTA, D., AND VAHDAT, A. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev. 35*, 2 (Sept. 2007), 42–51.

[16] CORBET, J. Jls2009: Generic receive offload. *Linux Weekly News (LWN)* (Oct. 2009). https://lwn.net/Articles/358910/.

[17] CORBET, J. Software interrupts and realtime. *Linux Weekly News (LWN)* (Oct. 2012). https://lwn.net/Articles/520076/.

[18] CORBET, J. Bulk network packet transmission. *Linux Weekly News (LWN)* (Oct. 2014). https://lwn.net/Articles/615238/.

[19] CRONKITE-RATCLIFF, B., BERGMAN, A., VARGAFTIK, S., RAVI, M., MCKEOWN, N., ABRAHAM, I., AND KESLASSY, I. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 230–243.

[20] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56* (2013), 74–80.

[21] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 77–88.

[22] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 127–144.

[23] DELIMITROU, C., AND KOZYRAKIS, C. Bolt: I know what you did last summer... in the cloud. *SIGARCH Comput. Archit. News 45*, 1 (Apr. 2017), 599–613.

[24] DOBRESCU, M., ARGYRAKI, K., AND RATNASAMY, S. Toward predictable performance in software packet-processing platforms. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 141–154.

[25] DRUSCHEL, P., AND BANGA, G. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *OSDI* (1996), vol. 96, pp. 261–275.

[26] FONSECA, R., DUTTA, P., LEVIS, P., AND STOICA, I. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 323–338.

[27] FUNARO, L., BEN-YEHUDA, O. A., AND SCHUSTER, A. Ginseng: Market-driven llc allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 295–308.

[28] Ganger, G. R., Engler, D. R., Kaashoek, M. F., Briceño, H. M., Hunt, R., and Pinckney, T. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst. 20*, 1 (Feb. 2002), 49–83.

[29] Ghodsi, A., Sekar, V., Zaharia, M., and Stoica, I. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 1–12.

[30] Gleixner, T. [announce] 3.6.1-rt1. *Linux Weekly News (LWN)* (Oct. 2012). https://lwn.net/Articles/518993/.

[31] Google Inc. Cloud functions. https://cloud.google.com/functions/.

[32] Gulati, A., Merchant, A., and Varman, P. J. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 437–450.

[33] Guo, C., Lu, G., Wang, H. J., Yang, S., Kong, C., Sun, P., Wu, W., and Zhang, Y. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Conference* (2010), ACM, p. 15.

[34] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. Enforcing performance isolation across virtual machines in xen. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (2006), Springer, pp. 342–362.

[35] Han, S., Marshall, S., Chun, B.-G., and Ratnasamy, S. Megapipe: A new programming interface for scalable network i/o. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 135–148.

[36] He, K., Rozner, E., Agarwal, K., Felter, W., Carter, J., and Akella, A. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 465–478.

[37] He, K., Rozner, E., Agarwal, K., Gu, Y. J., Felter, W., Carter, J., and Akella, A. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 244–257.

[38] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. In *Proceedings of HotCloud* (June 2016).

[39] Herbert, T., and de Bruijn, W. Scaling in the linux networking stack, 2011. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[40] Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., and Reinhardt, S. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 25–36.

[41] Jeong, E., Wood, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 489–502.

[42] Jeyakumar, V., Alizadeh, M., Mazieres, D., Prabhakar, B., Kim, C., and Azure, W. Eyeq: Practical network performance isolation for the multi-tenant cloud. In *HotCloud* (2012).

[43] Jeyakumar, V., Alizadeh, M., Mazières, D., Prabhakar, B., Kim, C., and Greenberg, A. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI* (2013).

[44] Kasture, H., and Sanchez, D. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 729–742.

[45] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 67–81.

[46] Kopytov, A. Sysbench manual. *MySQL AB* (2012).

[47] Kulkarni, S. G., Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K. K., Wood, T., Arumaithurai, M., and Fu, X. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 71–84.

[48] Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., and Hyden, E. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun. 14*, 7 (Sept. 2006), 1280–1297.

[49] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst. 34*, 2 (May 2016), 6:1–6:33.

[50] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, CO, October 2014).

[51] MACE, J., BODIK, P., FONSECA, R., AND MUSUVATHI, M. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI* (2015), pp. 589–603.

[52] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 459–473.

[53] MATTHEWS, J. N., HU, W., HAPUARACHCHI, M., DESHANE, T., DIMATOS, D., HAMILTON, G., MCCABE, M., AND OWENS, J. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (New York, NY, USA, 2007), ExpCS '07, ACM.

[54] MCCULLOUGH, J. C., DUNAGAN, J., WOLMAN, A., AND SNOEREN, A. C. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference–ATC* (2010), pp. 47–60.

[55] MICROSOFT CORP. Azure functions. https://azure.microsoft.com/en-us/services/functions/.

[56] MUNDADA, Y., RAMACHANDRAN, A., AND FEAMSTER, N. Silverline: Data and network isolation for cloud services. In *HotCloud* (2011).

[57] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst. 33*, 4 (Nov. 2015), 11:1–11:30.

[58] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM* (2012).

[59] POPA, L., YALAGANDULA, P., BANERJEE, S., MOGUL, J. C., TURNER, Y., AND SANTOS, J. R. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 351–362.

[60] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. Rethinking the library os from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2011), ACM.

[61] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for End-host Rate Limiting. In *NSDI* (2014).

[62] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 7:1–7:13.

[63] RODRIGUES, H., SANTOS, J. R., TURNER, Y., SOARES, P., AND GUEDES, D. O. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV* (2011).

[64] SHARMA, P., CHAUFOURNIER, L., SHENOY, P., AND TAY, Y. C. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference* (New York, NY, USA, 2016), Middleware '16, ACM, pp. 1:1–1:13.

[65] SHIEH, A., KANDULA, S., GREENBERG, A. G., AND KIM, C. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud* (2010).

[66] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 182–196.

[67] TURNER, P., RAO, B. B., AND RAO, N. Cpu bandwidth control for cfs. In *Proceedings of the Linux Symposium* (2010), pp. 245–254.

[68] VASILESCU, L., OLTEANU, V., AND RAICIU, C. Sharing cpus via endpoint congestion control. In *Proceedings of the Workshop on Kernel-Bypass Networks* (New York, NY, USA, 2017), KBNets '17, ACM, pp. 31–36.

[69] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[70] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), FAST '07, USENIX Association, pp. 5–5.

[71] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 607–618.

[72] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 474–489.

[73] Yosifovich, P., Ionescu, A., Russinovich, M. E., and Solomon, D. A. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*, 7th ed. Microsoft Press, 2017.

[74] Zhang, W., Rajasekaran, S., Duan, S., Wood, T., and Zhuy, M. Minimizing interference and maximizing progress for hadoop virtual machines. *SIGMETRICS Perform. Eval. Rev. 42*, 4 (June 2015), 62–71.

[75] Zhang, Y., and West, R. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2006), RTSS '06, IEEE Computer Society, pp. 191–201.

[76] Zilberman, N., Audzevich, Y., Covington, G. A., and Moore, A. W. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro 34*, 5 (2014), 32–41.