Iron: Isolating Network-based CPU in Container Environments

Junaid Khalid¹, Eric Rozner², Wesley Felter²,³, Cong Xu², Karthick Rajamani², Alexandre Ferreira²,⁴, Aditya Akella¹

¹Now at State Street, ⁴Now at ARM
Virtualization

Virtual machines

Containers

Hypervisor/ Host OS

App

OS

VM

App

OS

VM

App

OS

VM

App

Container

OS

App

Container

App

Container

HW
Virtualization

Virtual machines

- App
- OS
- VM

Hypervisor/Host OS

HW

- ✗ Heavy weight
- ✗ Slow

Containers

- App
- Container

OS

HW

- ✔ Light weight
- ✔ Fast
Virtualization

**Virtual machines**

- App
- OS
- VM

Hypervisor/Host OS

- HW

- Heavy weight
- Slow

**Containers**

- App Container
- OS
- Container

- HW

- Light weight
- Fast

Supports new workloads such as microservices and serverless
Isolation

Containers

App1
Container

App2
Container

OS

HW
Containers require strong resource isolation

- Memory
- Network
- CPU
Containers require strong resource isolation

- Memory
- Network
- CPU

Administrators want to strongly control resource allocation in multi-tenant environments
Containers require strong resource isolation

- Memory
- Network
- CPU

Administrators want to strongly control resource allocation in multi-tenant environments

Strong isolation is important for performance, predictability and efficiency
Isolation: A container *shouldn’t consume* more than its assigned share of *resources*.
Isolation: A container shouldn’t consume more than its assigned share of resources

allocated share

50% 50%
Isolation: A container *shouldn’t consume* more than its assigned share of *resources*
**Isolation**: A container *shouldn’t consume* more than its assigned share of *resources*

cgroups ensures **CPU isolation** by *allocating, metering, and enforcing* resource usage in the kernel.
Isolation

Isolation: A container shouldn’t consume more than its assigned share of resources.

CPU isolation provided by Linux breaks down while handling the network traffic.

cGroup ensures resource isolation by allocating, metering, and enforcing resource usage in the kernel.
Outline

• How and by how much is isolation broken
Outline

• How and by how much is isolation broken
• Iron’s design
  • Accounting of per-packet processing cost
  • Ensuring isolation via enforcement
    • Integration with Linux scheduler
    • Hardware-based packet dropping
Outline

• How and by how much is isolation broken
• Iron’s design
  • Accounting of per-packet processing cost
  • Ensuring isolation via enforcement
    • Integration with Linux scheduler
    • Hardware-based packet dropping
• Evaluation
  • Controlled workload
  • Realistic workload
Isolation is Broken

Penalty factor = \frac{Time that job takes when competing with traffic}{Time that job takes when competing with compute}

Containers

OS

HW
Isolation is Broken

Penalty factor = \[
\frac{\text{Time that job takes when competing with traffic}}{\text{Time that job takes when competing with compute}}
\]
Isolation is Broken

Containers

\[
\text{Penalty factor} = \frac{\text{Time that job takes when competing with traffic}}{\text{Time that job takes when competing with compute}}
\]
Isolation is Broken

Penalty factor = \( \frac{\text{Time that job takes when competing with traffic}}{\text{Time that job takes when competing with compute}} \)

Containers

- **TeraSort Container**
- **WordCount Container**

- **OS**
- **HW**

compute intensive
Isolation is Broken

**Penalty factor** = \( \frac{\text{Time that job takes when competing with traffic}}{\text{Time that job takes when competing with compute}} \)

Containers

- **TeraSort Container**
- **WordCount Container**

Network intensive

Compute intensive

**OS**

**HW**
Isolation is Broken

Penalty factor = \frac{\text{Time that job takes when competing with traffic}}{\text{Time that job takes when competing with compute}}

Containers

Core 1
- TeraSort Container
- WordCount Container

OS

HW

network intensive
compute intensive
Isolation is Broken

**Penalty factor** = \( \frac{\text{Time that job takes when competing with traffic}}{\text{Time that job takes when competing with compute}} \)

**Wordcount** can take 1.5x longer when it shares the core with **TeraSort**
**Isolation is Broken**

**Containers**

\[ \text{Penalty factor} = \frac{\text{Time that job takes when competing with traffic}}{\text{Time that job takes when competing with compute}} \]

**Wordcount** can take **1.5x** longer when it shares the core with **TeraSort** vs running alone.
Practical Implications

1) Insufficient isolation
   overcharging & high variance in the performance
Practical Implications

1) Insufficient isolation → overcharging & high variance in the performance

Containers

network intensive vs compute intensive

allocated share 50% 50%

App Container

App Container

OS

HW
Practical Implications

1) Insufficient isolation → overcharging & high variance in the performance

Containers

- Network intensive vs compute intensive

<table>
<thead>
<tr>
<th>App Container</th>
<th>App Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>OS</td>
</tr>
<tr>
<td>HW</td>
<td>HW</td>
</tr>
</tbody>
</table>

Allocated share: 50% vs 50%
Actual usage: 70% vs 30%
Practical Implications

1) Insufficient isolation
   - overcharging & high variance in the performance

2) Under provisioning
   - waste of potential revenue
Practical Implications

Containers

1) Insufficient isolation
   - overcharging & high variance in the performance

2) Under provisioning
   - waste of potential revenue
Practical Implications

Containers

1) Insufficient isolation
   overcharging & high variance
   in the performance

2) Under provisioning
   waste of potential revenue

allocated share: 100%
actual usage: 50%
1) Insufficient isolation → overcharging & high variance in the performance

2) Under provisioning → waste of potential revenue
How is Isolation Broken?

NIC ring buffer

OS

Tera Sort

Word Count

userspace

kernel

Interrupt handler

wordcount

Scheduled task/process

time

10
How is Isolation Broken?

Kernel

OS

NIC ring buffer

NIC

userspace

compute intensive

Tera Sort

Word Count

Interrupt handler

wordcount

Scheduled task/process

time

Word Count

Kernel

OS

NIC ring buffer

NIC

userspace

compute intensive

Tera Sort

Word Count

Interrupt handler

wordcount

Scheduled task/process

time

How is Isolation Broken?

How is Isolation Broken?
How is Isolation Broken?

- Network intensive
- Compute intensive

- Tera Sort
- Word Count

- NIC ring buffer
- NIC

- Userspace
- Kernel

- OS

- Interrupt handler: wordcount

- Scheduled task/process
How is Isolation Broken?

- NIC
- ring buffer
- Tera Sort
- Word Count
- OS
- userspace
- network intensive
- compute intensive
- NIC ring buffer
- Interrupt handler
- wordcount
- time
- Scheduled task/process
How is Isolation Broken?

- NIC
- buffer
- Tera
- Sort
- Word
- Count
- OS
- userspace
- kernel
- network intensive
- compute intensive

Interrupt handler
wordcount

Scheduled task/process

time

wordcount

wordcount

NIC ring buffer

NIC
How is Isolation Broken?

• Kernel processes network traffic via interrupts
How is Isolation Broken?

• Kernel processes network traffic via interrupts
How is Isolation Broken?

- Kernel processes network traffic via interrupts

Scheduler charges wordcount for “t”, instead of “t - Δt”
How is Isolation Broken?

- Kernel processes network traffic via interrupts

**Charging:** Reduction in the runtime of a container

Scheduler charges wordcount for “t”, instead of “t - Δt”

Interrupt handler wordcount

Scheduled task/process

Kernel processes network traffic via interrupts.
How is Isolation Broken?

- Kernel processes network traffic via interrupts
- Time spend in servicing interrupts is incorrectly charged

**Charging:** Reduction in the runtime of a container

Scheduler charges wordcount for “t”, instead of “t - Δt”

- Network intensive
- Compute intensive

NIC ring buffer

NIC

userspace

Tera Sort

Word Count
How did Linux get here?

“... [software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a **randomly chosen victim** w/o the ability to put any control on them.”

--Thomas Gleixner (Linux developer)

- Kernel processes network traffic via interrupts
- Time spent in servicing interrupts is not accounted properly

Scheduler consider is as CPU usage of wordcount

Scheduled task/process

NIC ring buffer

NIC

userspace

kernel

OS
How did Linux get here?

“... [software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a **randomly chosen victim** w/o the ability to put any control on them.”

--Thomas Gleixner (Linux developer)
"... [software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a **randomly chosen victim** w/o the ability to put any control on them."

--Thomas Gleixner (Linux developer)

- Kernel processes network traffic via interrupts
- Time spend in servicing interrupts is not accounted properly

```
Kernel processes network traffic via interrupts
```

```
...[software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.
```

```
Kernel processes network traffic via interrupts
```

```
Time spend in servicing interrupts is not accounted properly
```

```
...[software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.
```

```
Kernel processes network traffic via interrupts
```

```
Time spend in servicing interrupts is not accounted properly
```

```
...[software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.
```

```
Kernel processes network traffic via interrupts
```

```
Time spend in servicing interrupts is not accounted properly
```

```
...[software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.
```

```
Kernel processes network traffic via interrupts
```

```
Time spend in servicing interrupts is not accounted properly
```

```
...[software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.
```

```
Kernel processes network traffic via interrupts
```

```
Time spend in servicing interrupts is not accounted properly
```

```
...[software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.
```

```
Kernel processes network traffic via interrupts
```

```
Time spend in servicing interrupts is not accounted properly
```
How did Linux get here?

“... [software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a **randomly chosen victim** w/o the ability to put any control on them.”

--Thomas Gleixner (Linux developer)
How did Linux get here?

“... [software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.”

--Thomas Gleixner (Linux developer)

• Kernel processes network traffic via interrupts
• Time spend in servicing interrupts is not accounted properly
How did Linux get here?

“... [software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a **randomly chosen victim** w/o the ability to put any control on them.”

--- Thomas Gleixner (Linux developer)
How did Linux get here?

“... [software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.”

--Thomas Gleixner (Linux developer)

- Kernel processes network traffic via interrupts
- Time spend in servicing interrupts is not accounted properly

• Kernel processes network traffic via interrupts

... [software interrupts] a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.”

--Thomas Gleixner (Linux developer)
Sender Side

Sender stack

- Process
- TCP/IP stack
- vSwitch
- OS
- NIC ring buffer
- NIC
Sender Side

Sender stack

userspace
- Process

kernel
- TCP/IP stack
- vSwitch

NIC driver

NIC ring buffer

NIC
Sender Side

Sender stack

- Process
- TCP/IP stack
- vSwitch
- NIC driver
- NIC ring buffer
- NIC

userspace

kernel

Process context
No Problem!
Sender Side

- NIC
- NIC ring buffer
- NIC driver
- qdisc/tc
- vSwitch
- TCP/IP stack
- Process

- Rate limiter
- Userspace
- Kernel
Sender Side

- Process
- TCP/IP stack
- vSwitch
- qdisc/tc
- NIC driver
- NIC ring buffer
- NIC

Rate limiter
Sender Side

• Packet is enqueued in the process context

Process context
No Problem!
Sender Side

- Packet is enqueued in the process context
- System call exits after enqueuing the packet
Sender Side

- Packet is enqueued in the process context
- System call exits after enqueuing the packet
- Soft interrupt is responsible for dequeuing and delivering it to the NIC
Sender Side

Linux services a softirq

1) at the end of hardware interrupt processing, in the context of the currently scheduled process
Linux services a softirq

1) at the end of hardware interrupt processing, in the context of the **currently scheduled process**
Sender Side

Linux services a softirq

1) at the end of hardware interrupt processing, in the context of the currently scheduled process

2) through \textit{ksoftirqd thread} (a per core kernel thread)
Sender Side

Linux services a softirq

1) at the end of hardware interrupt processing, in the context of the currently scheduled process

2) through ksoftirqd thread (a per core kernel thread)
Sender Side

Softirq processing can be charged incorrectly or not charged at all to any container.

1) at the end of hardware interrupt processing, in the context of the current scheduled process.

2) through ksoftirqd thread (a per core kernel thread)
Experiment Setup
Experiment Setup

Container 1
Q = Period/N

Container 2
Q = Period/N

Container 3
Q = Period/N

Container N
Q = Period/N

Core
Experiment Setup

Container 1
Q = Period/N
Victim
(sysbench)

Container 2
Q = Period/N

Container 3
Q = Period/N

Container N
Q = Period/N

Core
Experiment Setup

Container 1
Q = Period/N
Victim
(sysbench)

Container 2
Q = Period/N
(sysbench)

Container 3
Q = Period/N
(sysbench)

Container N
Q = Period/N
(sysbench)

Core

Interferers
Experiment Setup

Container 1
Q = Period/N
Victim (sysbench)

Container 2
Q = Period/N
Sender/Receiver

Container 3
Q = Period/N
Sender/Receiver

Container N
Q = Period/N
Sender/Receiver

Core

Interferers

Penalty factor = \( \frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}} \)

Container 1
Q = Period/N
Victim (sysbench)

Container 2
Q = Period/N
(sysbench)

Container 3
Q = Period/N
(sysbench)

Container N
Q = Period/N
(sysbench)

Core

Interferers
Impact Of Network Traffic

Penalty factor = \frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}}
Impact Of Network Traffic

HTB is used for traffic shaping @ 5Gbps

TCP Sender

Penalty Factor = \[
\frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}}
\]

<table>
<thead>
<tr>
<th>Penalty Factor</th>
<th>Number of containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10 flows</td>
</tr>
<tr>
<td>1.5</td>
<td>50 flows</td>
</tr>
<tr>
<td>1.75</td>
<td>100 flows</td>
</tr>
</tbody>
</table>

HTB is used for traffic shaping @ 5Gbps
Impact Of Network Traffic

HTB is used for traffic shaping @ 5Gbps

**Penalty factor** = \[
\frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}}
\]

TCP Sender

<table>
<thead>
<tr>
<th>Number of containers</th>
<th>Penalty Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 flows</td>
<td>0.5</td>
</tr>
<tr>
<td>50 flows</td>
<td>1.0</td>
</tr>
<tr>
<td>100 flows</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Higher is worse
Impact Of Network Traffic

HTB is used for traffic shaping @ 5Gbps

Maximum penalty factor is around 1.85
Impact Of Network Traffic

**TCP Sender**

Penalty factor = \( \frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}} \)

-Higher is worse

HTB is used for traffic shaping @ 5Gbps

Maximum penalty factor is around **1.85**

Look at our paper for the impact of UDP traffic

<table>
<thead>
<tr>
<th>Number of containers</th>
<th>10 flows</th>
<th>50 flows</th>
<th>100 flows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 0
- 0.5
- 1
- 1.5
- 2
Receiver Side

- Receiver side problem is much worse than the sender
Receiver Side

• Receiver side problem is much worse than the sender

• Packet is processed in non-process context until copied to application’s socket
Impact Of Network Traffic

TCP Receiver

Penalty factor = Time that victim takes when competing with traffic / Time that victim takes when competing with sysbench

<table>
<thead>
<tr>
<th>Number of containers</th>
<th>10 flows</th>
<th>50 flows</th>
<th>100 flows</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Penalty Factor
Impact Of Network Traffic

**TCP Receiver**

Penalty factor = \( \frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}} \)

Higher is worse

Penalty Factor

Number of containers

- 10 flows
- 50 flows
- 100 flows
Impact Of Network Traffic

Maximum penalty factor is around 6

TCP Receiver

Number of containers

Penalty Factor

Higher is worse

Penalty factor = \frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}}
Scenarios When Isolation Breaks

OS

Compute intensive vs Network intensive
Scenarios When Isolation Breaks

Compute intensive vs Network intensive

Low network workload vs High network workload
Scenarios When Isolation Breaks

- **Compute intensive vs Network intensive**
- **Low network workload vs High network workload**
- **Network intensive vs Network intensive with kernel bypass**
Iron

A scheme that ensures and enforces accounting of network-based CPU consumed in the kernel on the behalf of a container.
Outline

• How and by how much is isolation broken
• Iron’s design
  • Accounting of per-packet processing cost
    • Ensuring isolation via enforcement
      • Integration with Linux scheduler
      • Hardware-based packet dropping
• Evaluation
  • Controlled workload
  • Realistic workload
Network Call Stack – Background

NIC interrupt
Network Call Stack – Background

- Kernel IRQ Handler
- Driver interrupt Handler
- Softirq Handler
- NAPI poll Handler
- Network stack Handlers

NIC interrupt

- do_IRQHandler
- napi_schedule
- do_softirq
- net_rx_action
- netif_receive_skb
- ip_rcv
- ...
Network Call Stack – Backgound

Kernel IRQ Handler

NIC interrupt

Driver interrupt Handler

Softirq Handler

NAPI poll Handler

Network stack Handlers

do_IRQ

napi_schedule

do_softirq

net_rx_action

netif_receive_skb

ip_rcv

...
Network Call Stack – Background

- NIC interrupt
  - Kernel IRQ Handler
    - do_IRQ
  - Driver interrupt Handler
    - napi_schedule
  - Softirq Handler
    - do_softirq
      - net_rx_action
        - netif_receive_skb
          - ip_rcv
            - ...
          - ...
          - ...

- NAPI poll Handler
- Network stack Handlers
Iron – Accounting

Receiver stack

- do_softirq
- net_rx_action
- netif_receive_skb
- ip_rcv
- ...

Iron – Accounting

Receiver stack

- do_softirq
- net_rx_action
- netif_receive_skb
- ip_rcv
- ...
Iron – Accounting

Receiver stack

do_softirq

net_rx_action

netif_receive_skb

ip_rcv

... run

start_time = localtime()

end_time = localtime()
Iron – Accounting

- Measuring time difference is non-trivial
  - Kernel is preemptable
  - Function in the call stack can be interrupted at any time

```
start_time = localtime()
```

```
end_time = localtime()
```

Receiver stack:
- do_softirq
  - net_rx_action
    - netif_receive_skb
      - ip_rcv
        - ...
      - run
        - interrupted
        - end_time = localtime()
Iron – Accounting

• Measuring time difference is non-trivial
  • Kernel is preemptable
  • Function in the call stack can be interrupted at any time
Iron – Accounting

- Measuring time difference is non-trivial
  - Kernel is preemptable
  - Function in the call stack can be interrupted at any time

```
start_time = cumtime + (localtime() – swaptime)
```
Iron – Accounting

- Measuring time difference is non-trivial
  - Kernel is preemptable
  - Function in the call stack can be interrupted at any time

```
start_time = cumtime + (localtime() – swaptime)
```
Iron – Accounting

• Measuring time difference is non-trivial
  • Kernel is preemptable
  • Function in the call stack can be interrupted at any time

Receiver stack

```
do_softirq

net_rx_action

netif_receive_skb

ip_rcv

...```

```
run

interrupted

run

cumulative execution time

\[ \text{start\_time} = \text{cumtime} + (\text{localtime()} - \text{swapt ime}) \]

last swapped in

26```
Iron – Accounting

• Measuring time difference is non-trivial
  • Kernel is preemptable
  • Function in the call stack can be interrupted at any time

Receiver stack

- `do_softirq`
- `net_rx_action`
- `netif_receive_skb`
- `ip_rcv`

... last swapped in

Cumulative execution time

\[
\text{start\_time} = \text{cumtime} + (\text{localtime}() - \text{swaptime})
\]

\[
\text{end\_time} = \text{cumtime} + (\text{localtime}() - \text{swaptime})
\]
Iron – Accounting

Receiver stack

- do_softirq
  - NET_RX_SOFTIRQ
- net_rx_action
- netif_receive_skb
- ip_rcv
  - ...
Iron – Accounting

Receiver stack

- do_softirq
- net_rx_action
- netif_receive_skb
- ip_rcv

\[ P_i = \text{per}_\text{pkt}_\text{cost} \]

\[ \text{pktcost}_i = P_i \]
Iron – Accounting

Receiver stack

- do_softirq
  - NET_RX_SOFTIRQ
- net_rx_action
- netif_receive_skb
  - ip_rcv
  - ...

\[ P_i = per_{pkt\_cost} \]

\[ pktcost_i = P_i \]

\[ batch\_cost \]
Iron – Accounting

Receiver stack

- do_softirq
- net_rx_action
- netif_receive_skb
- ip_rcv
- ...

\[
\text{batch\_cost} = \begin{cases} 
\text{per\_pkt\_cost} \\
\end{cases}
\]

\[
pktcost_i = P_i + \frac{\text{batch\_cost}}{|P|}
\]
Iron – Accounting

Receiver stack

\[
\text{do\_softirq} \rightarrow \text{NET\_RX\_SOFTIRQ} \rightarrow \text{net\_rx\_action} \rightarrow \text{netif\_receive\_skb} \rightarrow \text{ip\_rcv} \rightarrow \ldots
\]

HI, TX, RX, TIMER, SCSI & TASKLET
cost of all the softirqs

\[
\text{batch\_cost} = \left( \text{do\_softirq\_cost} - \sum S_i \right) \times \frac{S_{RX}}{\sum S_i}
\]

\[
P_i = \text{per\_pkt\_cost}
\]

\[
pktcost_i = P_i + \frac{\text{batch\_cost}}{|P|}
\]
Transmitter stack

1. do_softirq
2. net_tx_action
3. qdisc_run
4. dequeue_skb

...
Iron – Accounting

Transmitter stack

- **do_softirq**
  - NET_TX_SOFTIRQ
- **net_tx_action**
  - qdisc_run
  - dequeue_skb
  - ...

- Linux batches packets for transmission
Iron – Accounting

Transmitter stack

- `do_softirq`
  - `NET_TX_SOFTIRQ`
- `net_tx_action`
- `qdisc_run`
- `dequeue_skb`

- Linux batches packets for transmission
Iron – Accounting

Transmitter stack

- **do_softirq**
  - NET_TX_SOFTIRQ
- **net_tx_action**
- **qdisc_run**
- **dequeue_skb**
  - \( \text{batch} \)

### Linux batches packets for transmission

- We measure the cost of the batch and charge each packet within the batch an equal share

\[
pktcost = \frac{\text{batch_cost}}{\text{batch_size}}
\]
Iron – Accounting

Transmitter stack

- **do_softirq**
- **NET_TX_SOFTIRQ**
- **net_tx_action**
- **qdisc_run**
- **dequeue_skb**

- Linux batches packets for transmission
- We measure the cost of the batch and charge each packet within the batch an equal share
- To identify the container to charge at dequeue
  - We encode the container information in the skb while enqueueing the packet

\[
\text{pktcost} = \frac{\text{batch\_cost}}{\text{batch\_size}}
\]
Outline

• How and by how much is isolation broken
• Iron’s Design
  • Accounting of per-packet processing cost
  • **Ensuring isolation via enforcement**
    • **Integration with Linux scheduler**
      • Hardware-based packet dropping
• Evaluation
  • Controlled workload
  • Realistic workload
Iron – Enforcement
Scheduler Integration

• *Return the accounted time* to the container which was *incorrectly* charged
Iron – Enforcement
Scheduler Integration

• *Return the accounted time* to the container which was *incorrectly* charged
Iron – Enforcement
Scheduler Integration

• *Return the accounted time* to the container which was *incorrectly* charged
Iron – Enforcement
Scheduler Integration

- **Return the accounted time** to the container which was *incorrectly* charged
- **Charge the accounted time** to the container which was *responsible* for the network traffic
Iron – Enforcement
Scheduler Integration

• **Return the accounted time** to the container which was *incorrectly* charged
• **Charge the accounted time** to the container which was *responsible* for the network traffic
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler

At the end of the period, *running_time* is refilled by *quota*. 
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler

At the end of the period, running_time is refilled by quota.
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler

task on each core borrows time slices from the global scheduler

Core 1

Global Scheduler

running_time

local_running_time
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler

task on each core borrows time slices from the global scheduler
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler
Reuse infrastructure from cgroup and Linux scheduler

Iron – Enforcement
Scheduler Integration

Tracks the time container should get back

\[ \text{gained} \quad \text{running\_time} \]

local\_running\_time
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler

- **Global Scheduler**
  - **task**
    - Container
    - local_running_time
    - additional_cpu_usage
  - tracked time container should get back
  - running_time
  - gained

Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler

- Additional CPU usage:
  - < 0: not charged
  - > 0: charged incorrectly

- Local running time

- Gain tracks the time container should get back

- Core 1

- Task

- Container

- Global Scheduler

- Diagram illustrating task, container, and scheduler relationships.
Reuse infrastructure from cgroup and Linux scheduler

- Global Scheduler
  - Tracks the time container should get back
  - \(\text{gained} \rightarrow \text{running_time}\)

- Core 1
  - Container
    - task
  - local_running_time
  - additional_cpu_usage
    - \(< 0; \text{not charged}\)
    - \(> 0; \text{charged incorrectly}\)
Reuse infrastructure from cgroup and Linux scheduler

Iron – Enforcement
Scheduler Integration

Tracks the time container should get back

Tasks:
- Task
- Container
- Global Scheduler

Core 1

- local_running_time
- additional_cpu_usage

- < 0; not charged
- > 0; charged incorrectly

- gained
- running_time
Reuse infrastructure from cgroup and Linux scheduler

Global Scheduler

Tracks the time container should get back
- gained
- running_time

Core 1

Local running time
additional_cpu_usage

< 0; not charged
> 0; charged incorrectly
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler

**Throttling** a sender ensures isolation! Because throttled sender (runtime < 0) *cannot generate* outgoing traffic.
Iron – Enforcement
Scheduler Integration

Reuse infrastructure from cgroup and Linux scheduler

**Throttling** a sender ensures isolation! Because throttled sender (runtime < 0) **cannot generate** outgoing traffic.

If the receiver is throttled, incoming traffic can still arrive and consume CPU.
Outline

• How and by how much is isolation broken
• Iron’s Design
  • Accounting of per-packet processing cost
  • Ensuring isolation via enforcement
    • Integration with Linux scheduler
    • Hardware-based packet dropping
• Evaluation
  • Controlled workload
  • Realistic workload
Iron – Enforcement

Dropping Packets

Modifies the Linux’s polling mechanism (NAPI)
- Assigns a queue (ring buffer) to a container
Iron – Enforcement

Dropping Packets

Modifies the Linux’s polling mechanism (NAPI)
- Assigns a queue (ring buffer) to a container
- Iron strips the throttled queue from the polling list
Iron – Enforcement

Dropping Packets

Modifies the Linux’s polling mechanism (NAPI)
- Assigns a queue (ring buffer) to a container
- Iron strips the throttled queue from the polling list
  - From kernel’s point of view, there are no more interrupts – no packets
Iron – Enforcement

Dropping Packets

Modifies the Linux’s polling mechanism (NAPI)
• Assigns a queue (ring buffer) to a container
• Iron strips the throttled queue from the polling list
  • From kernel’s point of view, there are no more interrupts – no packets
  • From NIC’s point of view, kernel is busy and is not polling packets from the queue, so it stays in the polling mode.
Iron – Enforcement

Dropping Packets

Modifies the Linux’s polling mechanism (NAPI)

• Assigns a queue (ring buffer) to a container
• Iron strips the throttled queue from the polling list
  • From kernel’s point of view, there are no more interrupts – no packets
  • From NIC’s point of view, kernel is busy and is not polling packets from the queue, so it stays in the polling mode.
  • If a new packet arrives and the ring buffer is full, it gets dropped
Iron – Enforcement
Dropping Packets

Modifies the Linux’s polling mechanism (NAPI)
  • Assigns a queue (ring buffer) to a container
  • Iron strips the throttled queue from the polling list
    • From kernel’s point of view, there are no more interrupts – no packets
    • From NIC’s point of view, kernel is busy and is not polling packets from the queue, so it stays in the polling mode.
    • If a new packet arrives and the ring buffer is full, it gets dropped
Outline

• How and by how much is isolation broken
• Iron’s design
  • Accounting of per-packet processing cost
  • Ensuring isolation via enforcement
    • Integration with Linux scheduler
    • Hardware-based packet dropping
• Evaluation
  • Controlled workload
  • Realistic workload
Experiment Setup

Container 1
Q = Period/N
Victim (sysbench)

Container 2
Q = Period/N
(sysbench)

Container 3
Q = Period/N
(sysbench)

Container N
Q = Period/N
(sysbench)

Core

Interferers
**Experiment Setup**

Container 1
Q = Period/N
Victim *(sysbench)*

Container 2
Q = Period/N
Sender/Receiver

Container 3
Q = Period/N
Sender/Receiver

Container N
Q = Period/N
Sender/Receiver

Core

Interferers

Container 1
Q = Period/N
Victim *(sysbench)*

Container 2
Q = Period/N *(sysbench)*

Container 3
Q = Period/N *(sysbench)*

Container N
Q = Period/N *(sysbench)*

Core

Interferers
Experiment Setup

**Container 1**
\[ Q = \text{Period}/N \]
Victim (sysbench)

**Container 2**
\[ Q = \text{Period}/N \]
Sender/Receiver

**Container 3**
\[ Q = \text{Period}/N \]
Sender/Receiver

**Container N**
\[ Q = \text{Period}/N \]
Sender/Receiver

**Core**

**Interferers**

**Penalty factor**
\[ \text{Time that victim takes when competing with traffic} \]
\[ \text{Time that victim takes when competing with sysbench} \]

**Victim**
(sysbench)

**Sender/Receiver**

**Interferers**

\(42\)
Sender Interference With Iron

**Penalty factor** = $\frac{\text{Time that victim takes when competing with sysbench}}{\text{Time that victim takes when competing with traffic}}$

<table>
<thead>
<tr>
<th>Traffic rate (Gbps)</th>
<th>Penalty Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.05</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.95</td>
</tr>
<tr>
<td>3</td>
<td>0.90</td>
</tr>
<tr>
<td>4</td>
<td>0.85</td>
</tr>
<tr>
<td>5</td>
<td>0.80</td>
</tr>
<tr>
<td>6</td>
<td>0.75</td>
</tr>
</tbody>
</table>

2 containers per core
Sender Interference With Iron

Penalty factor remains below 1.04, significant decrease from 1.85

Penalty factor = \frac{Time that victim takes when competing with traffic}{Time that victim takes when competing with sysbench}
**Sender Interference With Iron**

**TCP Sender**

Penalty factor remains below 1.04, significant decrease from 1.85

**UDP Sender**

Penalty factor remains below 1.04, significant decrease from 1.18
Receiver Interference With Iron

\[
\text{Penalty factor} = \frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}}
\]
Receiver Interference With Iron

\[
\text{Penalty factor} = \frac{\text{Time that victim takes when competing with \textit{traffic}}}{\text{Time that victim takes when competing with \textit{sysbench}}}
\]

TCP traffic

![Graph showing the impact of number of flows on penalty factor. The x-axis represents the number of flows (1, 10, 25, 50, 75, 100), and the y-axis represents the penalty factor (0.94 to 1.04). There are different colors representing 1 rcv, 4 rcv, and 7 rcv, with error bars showing variability.](image-url)
**Receiver Interference With Iron**

Penalty factor = \( \frac{\text{Time that victim takes when competing with traffic}}{\text{Time that victim takes when competing with sysbench}} \)

### TCP traffic

![TCP traffic chart]

### UDP traffic

![UDP traffic chart]
Receiver Interference With Iron

Penalty factor never exceeds 1.05, significant decrease from 6 for TCP and 4.45 for UDP
Outline

• How and by how much is isolation broken
• Iron’s Design
  • Accounting of per-packet processing cost
  • Ensuring isolation via enforcement
    • Integration with Linux scheduler
    • Hardware-based packet dropping
• Evaluation
  • Controlled workload
  • Realistic workload
Impact on Big Data Applications

Setup
- 48 containers spread over 6 machines
- Each job runs over 24 containers
Impact on Big Data Applications

Setup
• 48 containers spread over 6 machines
• Each job runs over 24 containers

MapReduce jobs as victim:
• **wordcount**: counts word frequency
• **pi**: computes the value of pi
• **grep**: searches for a given word

Trace based Interferer:
• Shuffle phase of TeraSort job with
  115GB input file
Impact on Big Data Applications

Setup
• 48 containers spread over 6 machines
• Each job runs over 24 containers

MapReduce jobs as victim:
• **wordcount**: counts word frequency
• **pi**: computes the value of pi
• **grep**: searches for a given word

Trace based Interferer:
• Shuffle phase of TeraSort job with 115GB input file
Impact on Big Data Applications

Setup
- 48 containers spread over 6 machines
- Each job runs over 24 containers

MapReduce jobs as victim:
- **wordcount**: counts word frequency
- **pi**: computes the value of pi
- **grep**: searches for a given word

Trace based Interferer:
- Shuffle phase of TeraSort job with 115GB input file

Penalty factor never exceeds **1.04**
Summary

• Evaluated the interference caused by network-based containers.

• Provided hardened isolation for network-based processing in containerized environment.

• Ensures accurate accounting of the time spent processing network traffic in softirq.

• Integrated with Linux scheduler with minimal changes.

• Novel packet dropping mechanism to limit the interference.