THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
INDIA

Demystifying PCI Interrupt Handling in Linux: MSI/MSI-X

Shradha Gupta, Microsoft

Agenda

A packet arrives. How does your CPU find out?

1. **The old way** - shared wires (and why they broke)
2. **The fix** - MSI/MSI-X, memory writes instead of wires
3. **Who sets it up** - Linux kernel - sets IRQ domains, data structures the APIs
4. **What can go wrong** - debugging interrupts
5. **The payoff** - two real world case studies

How does Hardware talk to the CPU?

POLLING

Inefficient

- CPU repeatedly checks NIC

  check… check… check…
- CPU cycles wasted on empty checks
- No packet most of the time
- Higher latency, lower throughput

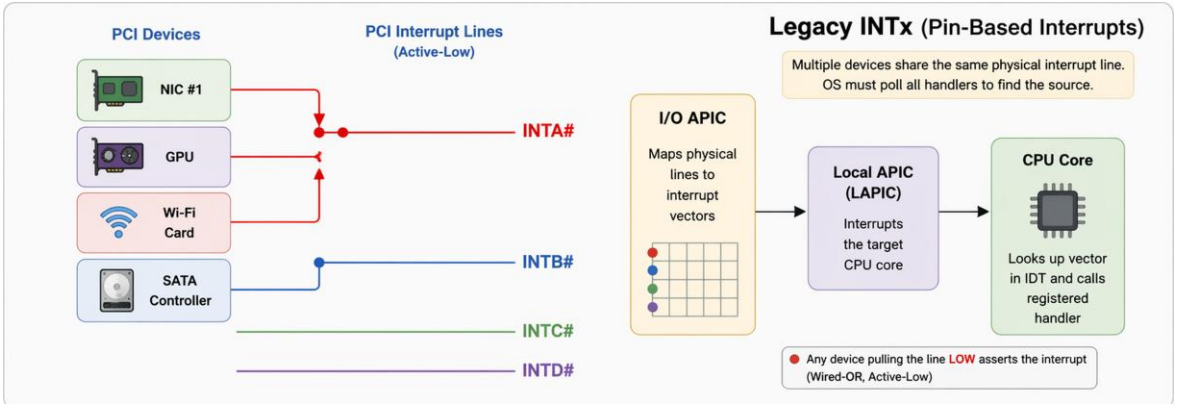
INTERRUPTS

Efficient

- Packet arrives at NIC
- NIC sends interrupt signal to CPU
- CPU wakes up only when needed
- Lower latency, better throughput

How Do Legacy Line-Based Interrupts Work?

- Asserts physical line (INTA#-INTD#)
- Active-low, level-triggered
- shared across devices
- Routed via IOAPIC -> LAPIC



What could go wrong?

Problem	Impact
Shared lines	All devices on same line must be polled to find source, latency and wasted work
Limited pins	Each function gets only one legacy pin-based interrupt; INTA#–INTD# are shared across the bus. Routing is bootleneck
Out-of-band signaling	Separate physical wires, racy
No per-queue interrupts	Cannot scale to multi-queue devices
Level-triggered	Level-triggered behaviour requires explicit acknowledgment and careful masking, higher latency

A fundamentally different approach needed.

Message Signaled Interrupts (MSI)

How MSI Works

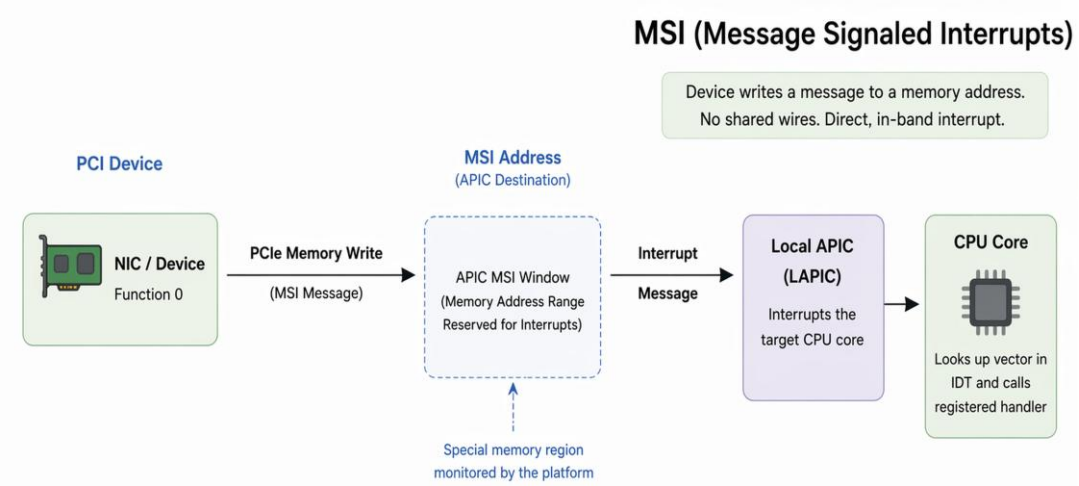
- Device writes data to a memory address
- Targets CPU's LAPIC directly
- No physical wires needed
- In-band, same as DMA data

MSI Structure

- PCI capability with 3 registers:
  Address · Data · Control
  Stored in PCI config space
- Up to 32 vectors per device

Advantages

- In-band signaling - no sideband wires
- Simpler hardware, better performance
- No sharing - each device gets its own vectors



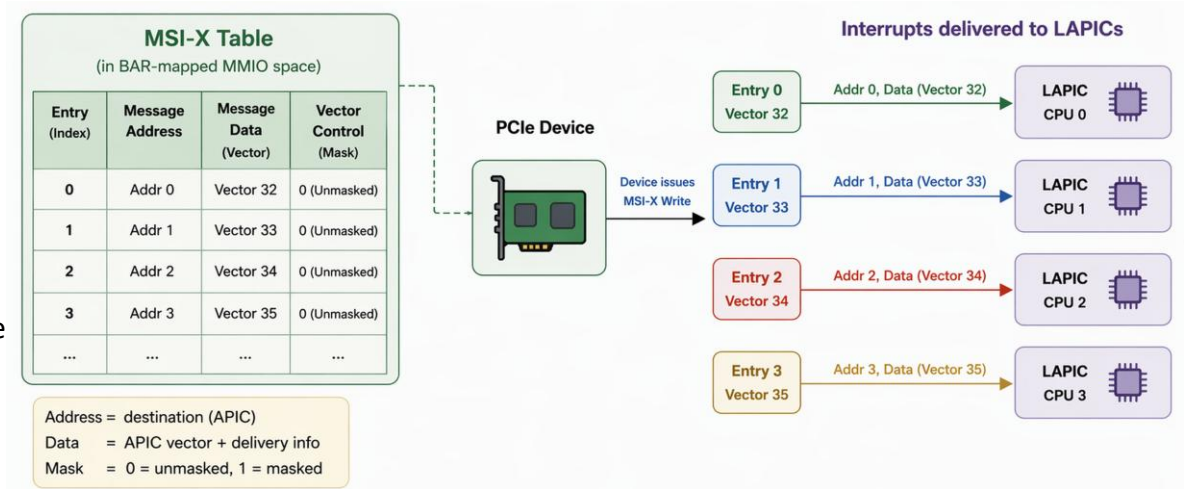
MSI-X - Scalable MSI

MSI Limitations

- Max 32 vectors, all share one address
- Data values must be consecutive

MSI-X Improvements

- Up to 2048 vectors, each independently configurable
- Table lives in BAR space, not config space
- Each entry targets a different CPU.
  - per-vector addressing

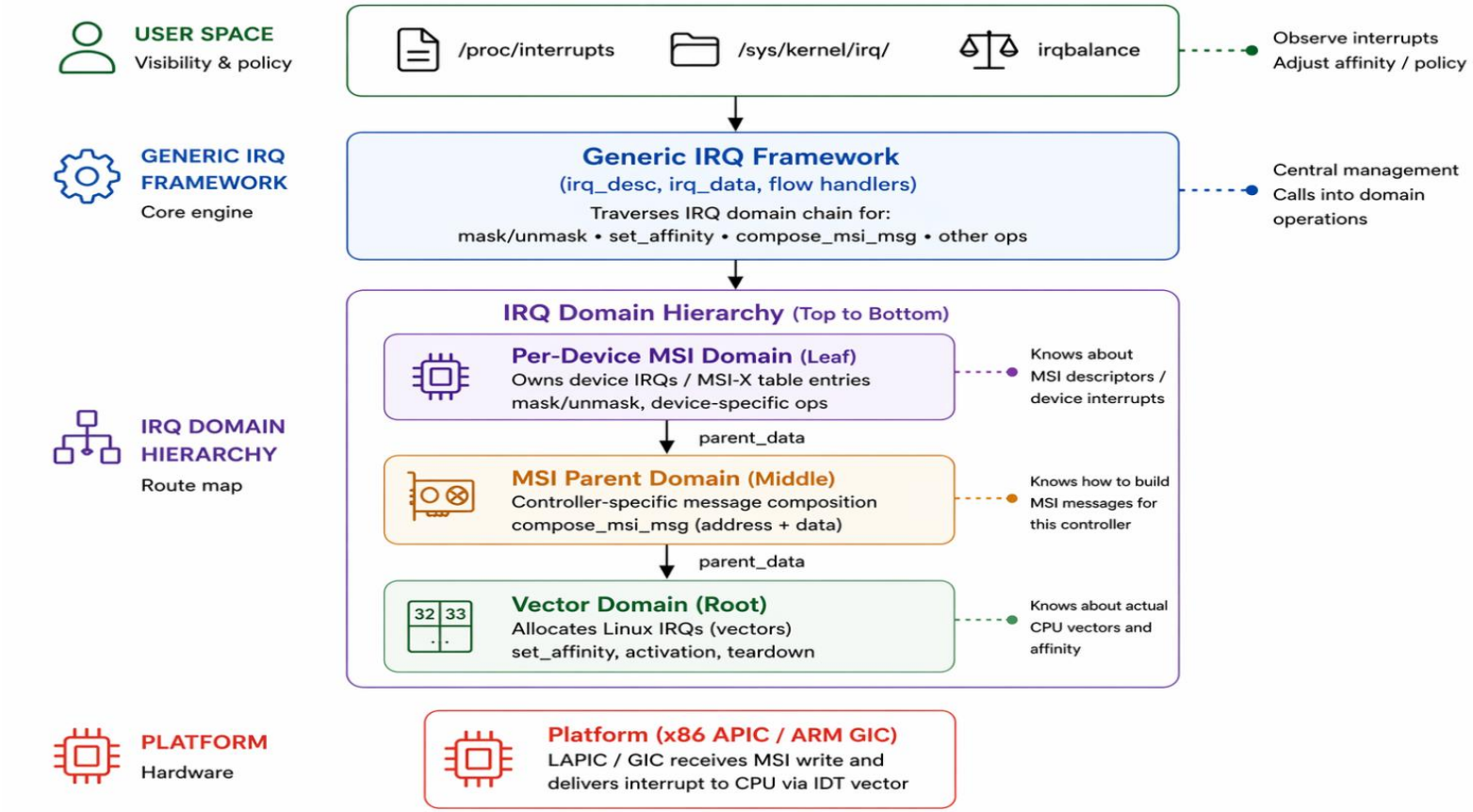


2048 independently targetable vectors → requires a kernel framework to manage allocation

INTx vs MSI vs MSI-X Comparison

Feature	INTx	MSI	MSI-X
Signal type	Out-of-band wire	In-band write	In-band write
Max vectors	4 (shared)	32	2048
Per-vector CPU targeting	No	No (shared addr)	Yes
Per-vector masking	No	Optional	Yes (mandatory)
Location	Wires	Config space	BAR space
Sharing	Forced	None	None
DMA ordering	Racy	Interrupt cannot pass prior DMA writes	Interrupt cannot pass prior DMA writes

Interrupt delivered, who sets it up? - Linux Kernel



IRQ Domain Hierarchy

Purpose: Separate concerns - each layer handles one level of translation

Per-Device MSI Domain (auto-created per PCI device)
MSI descriptor management · masking

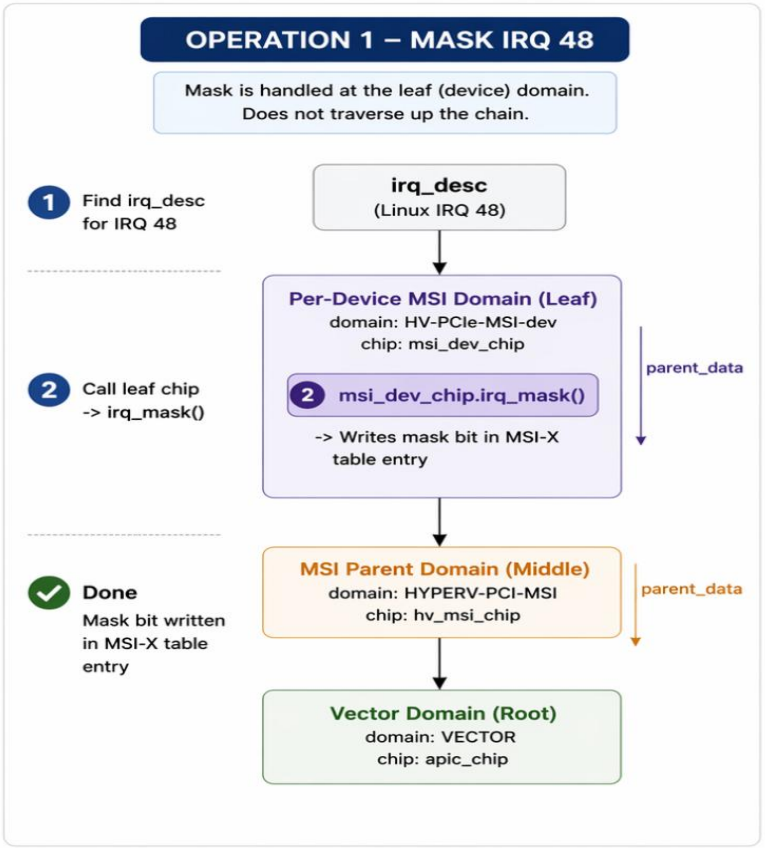
MSI Parent Domain (created by host controller driver)
Controller-specific logic · compose msg · set affinity

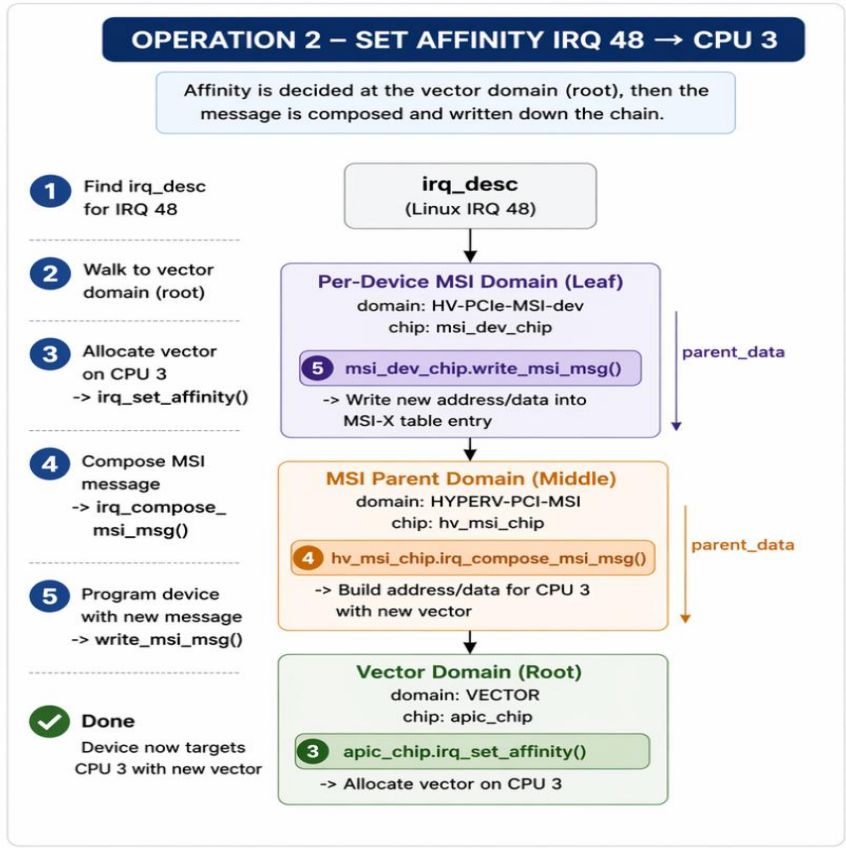
Root/Platform Domain (x86 vector domain / ARM GIC)
Vector allocation · CPU targeting · activation

Traversal on interrupt delivery:

Hardware MSI write → LAPIC → IDT vector →irq_desc lookup → installed flow handler invokes resolved chip callbacks (ack, eoi)

IRQ traversal through Domains





IRQ lifecycle (MSI-X)

Phase	Kernel Function	What Happens
1. Alloc vectors	`pci_alloc_irq_vectors()`	Creates per-device domain, reserves vectors
2. Request IRQ	`request_irq()`	Registers handler, calls irq_startup
3. Activate	`irq_domain_activate_irq()`	IRQD_ACTIVATED, Assigns APIC vector (managed: deferred to here)
4. Compose	`irq_chip_compose_msi_msg()`	Formats MSI address/data pair
5. Write	`pci_write_msi_msg()`	Programs address/data into MSI-X table
6. Unmask	`pci_msi_unmask_irq()`	Writes MSI-X table, interrupt is live
7. Runtime	*Hardware*	Device writes → LAPIC → ISR runs
8. Free	`free_irq()` / `pci_free_irq_vectors()`	Mask → deactivate → free → destroy domain

All of the above works perfectly - until it doesn't…

How do we debug it?

Practical Debugging - Tools Overview

Tool	What it shows
`/proc/interrupts`	Per-CPU interrupt counts, IRQ → device mapping
`/proc/irq/<N>/smp_affinity`	CPU affinity mask for IRQ N
`/proc/irq/<N>/effective_affinity`	Actual CPU the IRQ is delivered to
`/sys/kernel/debug/irq/irqs/<N>`	Detailed IRQ state (domain, chip, flags) *Requires CONFIG_GENERIC_IRQ_DEBUGFS=y*
`lspci -vvv`	MSI/MSI-X capability, vector count, BAR info
`perf stat -e irq_vectors:*`	Interrupt-related perf events
`ftrace` (irq events)	`irq_handler_entry`, `irq_handler_exit`
`bpftrace`	Dynamic tracing of IRQ paths
`mpstat -I ALL`	Per-CPU hardware IRQ + softIRQ time

Our Running Example: MANA on Hyper-V

MANA (Microsoft Azure Network Adapter)
- Virtual NIC for Azure/Hyper-V VMs
- Each queue gets its own MSI-X vector
- Uses non-managed IRQs

PCI Host Controller: pci-hyperv
- MSI parent domain: hv_msi_domain
- Compose requires hypercall to host
  (hv_compose_msi_msg → VMBus round-trip)

IRQ Domain Chain for MANA:

Per-Device (MANA) → HV-PCIe-MSI (parent) → VECTOR (root)

Is it working?

msi-x-diagnostics ~ bash

Step 1: Are interrupts arriving and spread across CPUs?

```
$ cat /proc/interrupts | grep mana
  46   0  0  1082  0  HV-PCIe-MSI 2-edge mana_q0@pci:7870:00:00.0
  47  857  0    0  0  HV-PCIe-MSI 3-edge mana_q1@pci:7870:00:00.0
  48   0  821   0  0  HV-PCIe-MSI 4-edge mana_q2@pci:7870:00:00.0
  49   0  0   804  0  HV-PCIe-MSI 5-edge mana_q3@pci:7870:00:00.0
```

✓ Counts incrementing ✓ Even per CPU

Step 2: Is affinity correct?

```
$ cat /proc/irq/46/effective_affinity_list → 2
$ cat /proc/irq/47/effective_affinity_list → 0
$ cat /proc/irq/48/effective_affinity_list → 1
```

✓ Each queue pinned to different CPU, no clustering

Step 3: Is the device configured correctly?

```
$ lspci -s 7870:00:00.0 -vvv | grep -A5 "MSI-X"
  MSI-X: Enable+ Count=1024 Masked-
  Vector table: BAR=4 offset=00003000
```

✓ MSI-X enabled ✓ Not masked

Step 4: Is the domain hierarchy intact?

```
$ cat /sys/kernel/debug/irq/domains/HV-PCIe-MSI-7870:00:00.0
  mapped: 4  parent: HYPER-V-PCIe[4] → VECTOR
```

✓ Domain chain verified ✓ All vectors mapped

How fast?

perf-diagnostics ~ bash

Step 1: Per-CPU interrupt rate during network load:

```
$ perf stat -e irq:irq_handler_entry -a --per-cpu -- iperf3 -c 10.0.0.1 -t 10
  CPU0              12,847      irq:irq_handler_entry
  CPU1              13,102      irq:irq_handler_entry
  CPU2              12,956      irq:irq_handler_entry
  CPU3                 214      irq:irq_handler_entry   ← imbalanced!
```

→ If one CPU shows far fewer/more interrupts, check affinity settings.

Step 2: Handler latency via ftrace:

```
$ echo 1 > /sys/kernel/debug/tracing/events/irq/irq_handler_entry/enable
$ echo 1 > /sys/kernel/debug/tracing/events/irq/irq_handler_exit/enable
  <idle>-0 [002] d.h. 123.456: irq_handler_entry: irq=48 name=mana_q2@pci:7870:00:00.0
  <idle>-0 [002] d.h. 123.457: irq_handler_exit:  irq=48 ret=handled
```

→ 1 µs handler duration = healthy (NAPI kick + return)
→ 50+ µs = problem (cache thrash, excessive work in hardirq)

What broke?


```
irq-debug-patterns ~ bash

PATTERN 1: MSI-X partial allocation - silent performance drop
Symptom:     Device throughput well below expected, no errors in dmesg.
             Driver got fewer vectors than requested, silently created fewer queues.
             Even interrupt distribution on existing queues
Diagnose:    $ ethtool -l <dev>           → requested queue count
             $ grep <device> /proc/interrupts → actual IRQs allocated
Investigate: Was the vector domain exhausted at probe time?
             Did another device claim vectors first?
             Did the driver handle partial allocation correctly?

PATTERN 2: No interrupts after hotplug/migration
Symptom:     Device silent - no errors, no panic, just no interrupts.
Diagnose:    $ dmesg | grep compose → "hv_compose_msi_msg: timed out"
             debugfs irqs/<N>: chip_data = NULL (compose/setup failure)
Investigate: Did the compose round-trip to the host complete?
             Was the device re-probed or was compose retried?
             Is this a managed IRQ composing in atomic context?

PATTERN 3: Managed IRQ silently dead after CPU hotplug
Symptom:     I/O queue stops completing after CPU offline/online cycle.
Diagnose:    debugfs irqs/<N> → VECTOR domain hwirq = 0x00 (no vector assigned)
             IRQD_ACTIVATED flag not set
Investigate: Did irq_domain_activate_irq() run on CPU online?
             Is the CPU online notifier triggering re-activation?
             Was the vector returned to the pool and never reclaimed?
```

Case Study: Networking - MANA on Azure

mana-case-studies ~ bash

The design:

- Each RX/TX queue pair gets dedicated MSI-X vector
- Non-managed IRQs: driver controls affinity via irq_set_affinity_and_hint()
- Interrupt -> NAPI poll -> batch packet processing, all on one CPU

1. Dynamic MSI-X allocation (2025, Shradha Gupta - net-next)

Problem: Driver statically allocated Hardware max QUEUE_COUNT MSI-X vectors at probe time -> In cases where hardware offers less that the MAX queues we overallocated and waste vectors

Symptom: device works fine functionally, but allocated vectors sit unused. Failures in other devices' vector requests

Tools: /proc/interrupts -> to check all allocated MANA interrupts
/proc/interrupts counts to indicate only subset of interrupts actually triggered

Fix: Allocate 1 vector for HWC -> query hardware -> dynamically add needed vectors
Uses pci_alloc_irq_vectors() incrementally

Result: Precise resource usage; flexible VM resizing;

2. Optimize IRQ affinity for low vCPU configs (2026, Shradha Gupta)

Problem: IRQs > core and SMP enabled, Topology aware affinity logic would distribute IRQs unevenly (some vCPUs got more, while other sibling vCPU was free)
-> softIRQ imbalance and throughput collapse

Symptom: throughput collapses under parallel TCP load. Some vCPUs are saturated with softIRQ load while others are mostly idle

Tools: /proc/irq/N/smp_affinity_list -> gives driver provided affinity,
/proc/irq/N/effective_affinity_list -> gives actual affinity,
mpstat –I ALL -> gives HW IRQ and softIRQ overhead

Fix: Spread IRQs across all vCPUs regardless of NUMA topology;

Result: 7.73 → 15.65 Gbps (2× throughput, 4 vCPU VM, 20480 TCP connections)

Case Study: Storage - NVMe Interrupt Scaling

nvme-case-studies ~ bash

The design:
- 1 completion queue per CPU = 1 MSI-X vector per CPU
- Managed IRQs (PCI_IRQ_AFFINITY): kernel auto-spreads, handles hotplug

1. Quirk for broken MSI (2024, Sean Anderson – nvme-pci)

Problem: Some NVMe devices advertises MSI but implementation is faulty; every command times out -> 20+ minute boot, device unusable

Symptom: device hangs on boot; NVMe commands timeout endlessly

Tools: dmesg | grep nvme -> shows I/O timeouts, /proc/interrupts show IRQs stuck
lspci –vvv -> shows MSI-X not available

Fix: Add device quirk in nvme-pci to detect broken MSI at probe; fall back to legacy INTx / emulated pin interrupts

Result: Device functions reliably; demonstrates need for reliable fallback when MSI/MSI-X is broken or unavailable

2. Queue affinity respecting isolated CPUs (2026, Daniel Wagner – nvme-pci)

Problem: NVMe IRQ affinity ignores isolcpus= parameter; interrupts land on CPUs reserved for RT workloads -> latency spikes

Symptom: Latency spikes on isolated cores; isolation guarantee violated

Tools: /sys/devices/system/cpu/isolated -> gives list of isolated CPUs
/proc/irq/N/effective_affinity_list -> shows isolated CPUs

Fix: genirq/affinity: cpumask helpers to exclude isolated CPUs
blk-mq: builds queue-to-CPU maps respecting isolcpus
nvme-pci: uses block layer helpers for queue setup

Result: RT workloads on isolated cores see consistent latency

The interrupt architecture gives you the tools; the driver must use them correctly

Key Takeaways

Scalability

Per-queue interrupts reduce lock contention, allowing systems to handle increased loads efficiently.

Latency

Maintaining correct affinity ensures processing occurs on the appropriate CPU, minimizing delays.

Simplicity & Debugging

Modern frameworks simplify driver code, and tools like /proc/interrupts, perf, and ftrace facilitate quick troubleshooting.

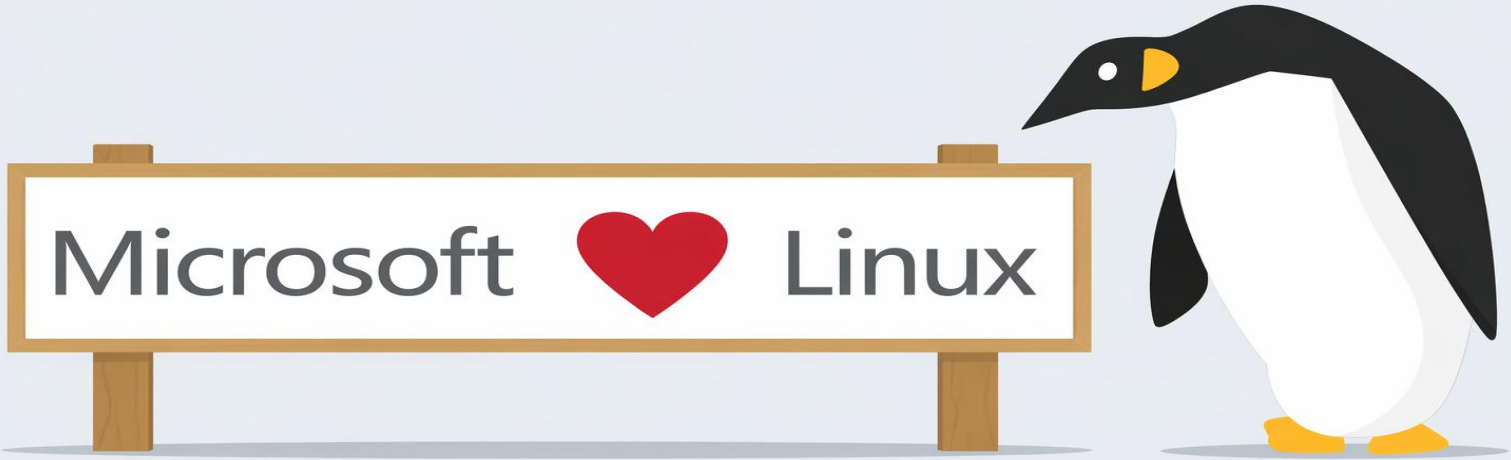
References

- Documentation/core-api/irq - Kernel IRQ subsystem documentation
- kernel/irq/msi.c - MSI domain core implementation
- drivers/irqchip/irq-msi-lib.c - MSI library helpers
- arch/x86/kernel/apic/vector.c - x86 vector domain
- drivers/pci/controller/pci-hyperv.c - Hyper-V PCI controller
- drivers/net/ethernet/microsoft/mana - MANA network driver
- Commit 5f83d6337c9c4 - “PCI: hv: Switch to msi_create_parent_irq_domain()”
- PCI Local Bus Specification - Chapter 6.8 (MSI/MSI-X)
- Thomas Gleixner, “Genirq: The Linux Generic Interrupt Infrastructure” (LWN)
- Case study commit discussion:
- Dynamic MSI-X IRQ allocation in MANA shradhagupta@linux.microsoft.com
- Linear IRQ affinity for lower configs MANA shradhagupta@linux.microsoft.com
- Add Quirks for broken MSI sean.anderson@linux.dev
- Restrict managed IRQ affinity to housekeeping CPUs

Questions?

Microsoft

Contact: shradhagupta@linux.microsoft.com



THE LINUX FOUNDATION

OPEN SOURCE SUMMIT INDIA