Improve Xeon IRQ throughput with posted interrupt

LPC 2023 VFIO/IOMMU/PCI MC

Jacob Pan

jacob.jun.pan@intel.com

Acknowledgement

- Rajesh Sankaran and Ashok Raj for the original idea

- Thomas Gleixner for reviewing and guiding the upstream direction of PoC patches. Help correct my many misunderstandings of the IRQ subsystem.

- Jie J Yan(Jeff), Sebastien Lemarie, and Dan Liang for performance evaluation with NVMe and network workload

- Bernice Zhang and Scott Morris for functional validation

- Michael Prinke helped me understand how VT-d HW works☺

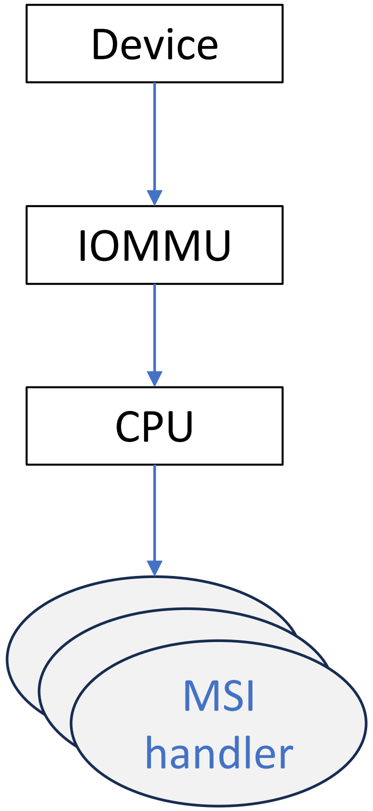
- Sanjay Kumar for providing the DSA IRQ test suite

Background

- IOMMU Interrupt remapping (IR) is required and turned on by default to support X2APIC
- VT-d and other IOMMUs support two IR modes
  - Remapped
  - Posted (used in VM only today, MSIs from the directly assigned devices can be delivered to the guest kernel)

Device MSI to CPU HW Flow (remappable mode)

1. Devices issue interrupt requests with writes to 0xFEEx_xxxx
2. The system agent accepts the IRQ and remaps/translates based on Interrupt Remapping Table Entries (IRTE)
3. Upon receiving the translation response, the system agent notifies the CPU with the translated MSI
4. CPU's local APIC accepts the MSI into its IRR/ISR registers
5. IDT delivery to the OS IRQ handler (MSI vector)



Device MSI to CPU HW Flow (posted mode)

3. Notifies the destination CPU with a notification vector

   - IOMMU suppresses CPU notification

   - IOMMU atomic swap IRQ status to memory (PID)

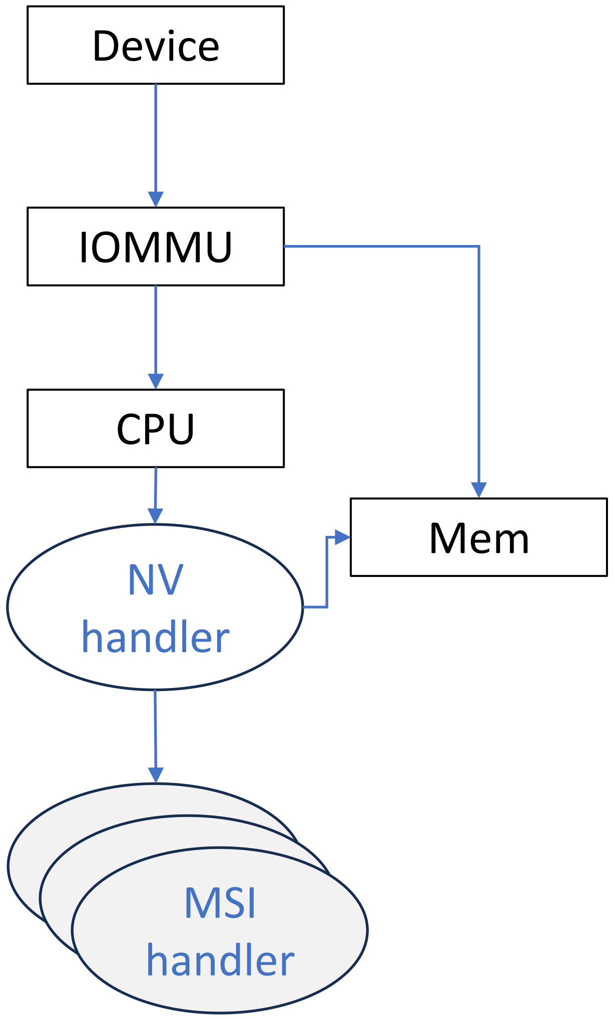
4. CPU's local APIC accepts the notification interrupt into its IRR/ISR registers

5. Interrupt delivered through IDT (notification vector handler)

6. System SW allows new notifications.

Note:

- 1 & 2 are the same as the remappable mode
- APIC only sees notification vectors but not MSI vectors



Problem statement

Remappable interrupt mode is inefficient under high IRQ rates

- CPU notifications are often unnecessary when the destination CPU is already overwhelmed (with handling bursts of IRQs)

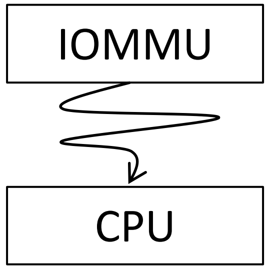
- CPU notifications are expensive/slow at least on Xeon

- Strong ordering between MSI writes and DMA.

As a result, slower IRQ rates can become a limiting factor for DMA IO performance.

Xeon sapphire rapids, FIO IOPS performance per disk on the same socket

# of disks	2	4	8
IOPS(million/disk/sec)	1.991	1.136	0.834



“THE LONG AND WINDING ROAD - The Beatles ”?

The proposal: Coalesced Interrupt Delivery (CID) with Posted MSI

```
Time --------------------------------------------------------------------->
     ^ ^ ^       ^ ^ ^ ^                ^   ^
MSIs A B C       D E F G                H   I

RI   N  N'  N'   N  N'  N'  N’          N   N

PI¹  N           N                      N   N

RI: remapped interrupt;  PI:  posted interrupt;
N: interrupt notification, N': superfluous interrupt notification
```

[1] CPU notifications are coalesced during IRQ bursts. N' eliminated in the flow above. We call this mechanism Coalesced Interrupt Delivery (CID).

Posted MSI: MSIs delivered as posted interrupts

What is really changing with posted MSI?

- All MSI vectors are multiplexed into a single notification vector for each CPU
- MSI vectors are then de-multiplexed by SW, driver handlers are dispatched without IDT delivery
- We lose the following compared to the remappable mode, but none of the below are used for device MSIs.
  - Fixed mode only, cannot choose delivery modes such as NMI
  - Physical X2APIC destination only
- MSI vectors can use the entire 0-255 range, not subject to local APIC restrictions

Performance with the patch

	Before	After	%Gain
FIO libaio (mil IOPS/sec/disk)	0.834[1]	1.943	**132%**
	1.136[2]	2.023	**78%**
DSA memfill[3] (mil IRQs/sec)	5.157	8.987	**74%**

[1] 1000 IOPS, 8 gen 5 NVMe disks on a single root port, Intel Xeon Sapphire Rapids, 48 cores 3.8/2.5GHz,

[2] Same as above but with 4 disks

[3] Two dedicated workqueues from separate Intel Data Streaming Accelerator (DSA) PCI devices, pin IRQ affinity of the two vectors to a single CPU. Queue size 128, batch size 128, buffer size 512B. Memory bandwidth gains are proportional to IRQ rate gains.

RFC patch: https://lore.kernel.org/lkml/20231112041643.2868316-1-jacob.jun.pan@linux.intel.com/T/#m6396a87995344345a9513a6b229b00972a5aeae8

So far, there has been no observable performance difference at the lower IRQ rates (<1M/sec/CPU)

However, the posted interrupt does require atomic xchg/swaps on PIDs from both CPU and IOMMU.

Implementation choices

- Transparent to the device drivers
- System-wide option instead of per-device or per-IRQ opt-in, i.e. once enabled all device MSIs are posted.
  - Excluding IOAPIC, HPET, and VT-d’s own IRQs
- Limit the number of polling/demuxing loops per CPU notification event
- In IRQ domain hierarchy VECTOR/APIC->INTEL-IR-POST->PCI-MSI
- X86 Intel only so far, can be extended to other architectures with posted interrupt support (ARM and AMD)
- Bare metal only

Posted MSI de-mux loop

```
DEFINE_IDTENTRY_SYSVEC(sysvec_posted_msi_notification)
{
            pid = this_cpu_ptr(&posted_interrupt_desc);
            inc_irq_stat(posted_msi_notification_count);
            irq_enter();
            while (i++ < max_posted_msi_coalescing_loop)
{
                        handle_pending_pir(pid, regs); *
                        if (is_pir_pending(pid))
                                    continue;
                        else
                                    break;
            }
            apic_eoi();
            pi_clear_on(pid);**
            irq_exit();
}
```

* Call driver handler for each bit posted in PID.PIR (pending vectors):

** Actual code does one more round to address the race where the IRQs are posted while we do pi_clear_on()

How to tell posted MSI is running?

```
IRQ debugfs:
domain:  IR-PCI-MSIX-0000:6f:01.0-12
 hwirq:   0x8
 chip:    IR-PCI-MSIX-0000:6f:01.0
  flags:   0x430
              IRQCHIP_SKIP_SET_WAKE
              IRQCHIP_ONESHOT_SAFE
 parent:
    domain:  INTEL-IR-12-13
     hwirq:   0x90000
     chip:    INTEL-IR-POST
      flags:   0x0
     parent:
        domain:  VECTOR
         hwirq:   0x65
         chip:    APIC
```

```
cat /proc/interrupts | grep PMN
PMN: 1387 Posted MSI notification event
```

No change to the device MSI accounting.

Runtime behavior comparison for 3 MSIs

BEFORE

```
interrupt
    irq_enter()
    handler() /* EOI */
    irq_exit()
        process_softirq()
system interrupt() /* e.g. timer */
interrupt
    irq_enter()
    handler() /* EOI */
    irq_exit()
        process_softirq()

interrupt
    irq_enter()
    handler() /* EOI */
    irq_exit()
        process_softirq()
```

AFTER

```
interrupt /* Posted MSI notification vector */
    irq_enter()
        handler()
        handler()
        handler()
        apic_eoi()
    irq_exit()
        process_softirq()
system interrupt() /* e.g. timer */
```

Higher priority system interrupt and softIRQs are blocked inside the MSI demuxing loop!

Attempt #1: limit the max loop count for polling pending posted interrupts

(Implemented in the RFC patch)

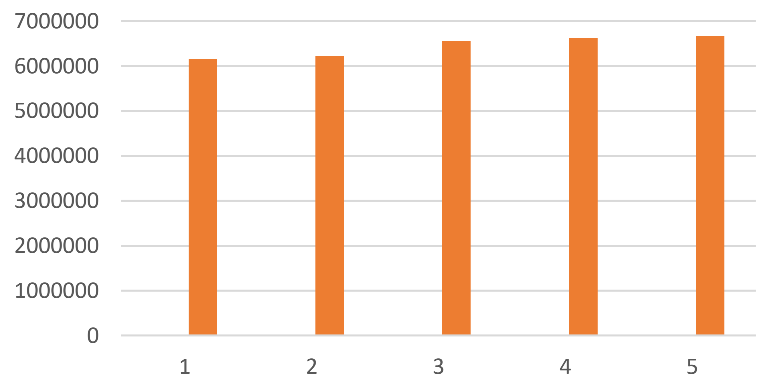
Data shows max coalescing loop = 3 gets 90+% performance benefit, should we make it tunable? E.g. /sys/kernel/max_posted_msi).

Today, we already allow one low-priority MSI to block system interrupts, can we tolerate more?

```
Intel data streaming accelerator (DSA) one dedicated work queue IRQ MEMORY_FILL
performance w.r.t. coalescing loop limit. (128 queue size, batch size 128,
buffer size 512B)

MaxLoop              IRQ/sec              bandwith Mbps
---------------------------------------------------------
1                    6157107              25219
2                    6226611              25504
3                    6557081              26857
4                    6629683              27155
5                    6662425              27289
#of IRQ coalesced is max_loop + 1 since we have to one more round
after the loop.
```

DSA IRQs/sec/wq/CPU vs. Max Loop



Attempt #2:
Make the posted MSI notification IRQ handler preemptable but not reentrant

(Evaluating, not included in RFC)

- Choose the notification vector in a lower priority class than all other system vectors
- All MSIs are multiplexed into one vector, there will be no MSI nesting.
- The nesting is one deep. Little risk for IRQ stack overflow? Or use an IST entry for a separate stack.

```
+       local_irq_enable();
        while (i++ < MAX_POSTED_MSI_COALESCING_LOOP) {
                handle_pending_pir(pid, regs);
         }
+       local_irq_disable();
        pi_clear_on; /* enables new notifications */
```

AFTER

```
interrupt
    irq_enter()
        local_irq_enable()
        system interrupt()
        handler()
        handler()
        handler()
        apic_eoi()
        local_irq_disable()
    irq_exit()
        process_softirq()
```

Attempt #3:Use self-IPI to invoke MSI handler

Take away the majority of the performance benefits, not preferred

Opens

Should we extend to other architectures with PI capability?

More testing suggestions?

- IRQ affinity change, migration
- CPU offlining
- Multi vector coalescing
- Low IRQ rate, general no-harm test
- VM device assignment

Summary

- On Intel Xeon CPUs, posted MSI on bare metal can improve IRQ throughput significantly. No need to buy new HW!
- To achieve maximum performance, there may be a transitory, limited, delay in processing system IRQ and softIRQs, or making notification IRQ preemptable

Alternatives to CID - tuning and SW stack change

- NVMe IRQ coalescing, E.g. nvme set-feature /dev/nvme0n1 -f 8 -v 100set-feature:08 (Interrupt Coalescing), value:0x000064
  - Longer latency for low throughput workload
  - Cannot coalesce among different vectors
  - Device-specific coalescing factor (may not be required on slower NVMe disk)
- IO_URING polling

Preemption ftrace in IRQ

```
: funcgraph_entry:                     |  __sysvec_posted_msi_notification() {
: hrtimer_cancel:       hrtimer=0xff1100103d1b47f0
: hrtimer_expire_entry: hrtimer=0xff1100103d1b47f0 now=472164000266 function=tick_sched_time

: hrtimer_expire_exit:  hrtimer=0xff1100103d1b47f0
: hrtimer_start:        hrtimer=0xff1100103d1b47f0 function=tick_sched_timer/0x0 expires=472

: bputs:                __sysvec_posted_msi_notification: timer pending!!!      //IRR set for timer vec 0xec
: bprint:               __sysvec_posted_msi_notification: TPR 10 : 0
: bprint:               __sysvec_posted_msi_notification: PPR 10 : 0
: funcgraph_entry:                     |    irq_chip_ack_parent() {
: funcgraph_entry:        0.112 us     |      apic_ack_irq_no_eoi();
: funcgraph_exit:         0.316 us     |    }
: funcgraph_entry:                     |    handle_irq_event() {
: funcgraph_entry:                     |      handle_irq_event_percpu() {
: irq_handler_entry:    irq=76 name=idxd-portal
: irq_handler_exit:     irq=76 ret=handled
: funcgraph_exit:         0.271 us     |      }
: funcgraph_exit:         0.478 us     |    }
: funcgraph_entry:                     |    irq_chip_ack_parent() {
: funcgraph_entry:        0.075 us     |      apic_ack_irq_no_eoi();
: funcgraph_exit:         0.276 us     |    }
: funcgraph_entry:                     |    handle_irq_event() {
: funcgraph_entry:                     |      handle_irq_event_percpu() {
```