Interrupt Swizzling Solution for Intel® 5000 Chipset Series based Platforms

Application Note

August 2006

Document Number: 314337-002

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Contents

Figures

Tables

Revision History

Revision Number	Description	Date
001	• Initial release of the document.	August 2006
002	• Updated product names.	August 2006

1 Introduction

Intel® 5000 Series Chipset supports PCIe devices attached to the MCH and/or the Intel® 631xESB / 632xESB I/O Controller Hub (including integrated PCIe* devices). Interrupt support for these PCIe devices is using PCI compatible INTx emulation scheme or using MSI/MSI-X scheme.

In case of INTx emulation scheme, the interrupts from the PCIe devices are signaled as inband messages that are eventually converted to system interrupts by the root complex. In order to better distribute INTx interrupts, the PCIe Base Specification 1.0a requires bridges to map interrupts from secondary side to primary side based on device number (per Table 2-13 of PCIe Base Specification 1.0a).

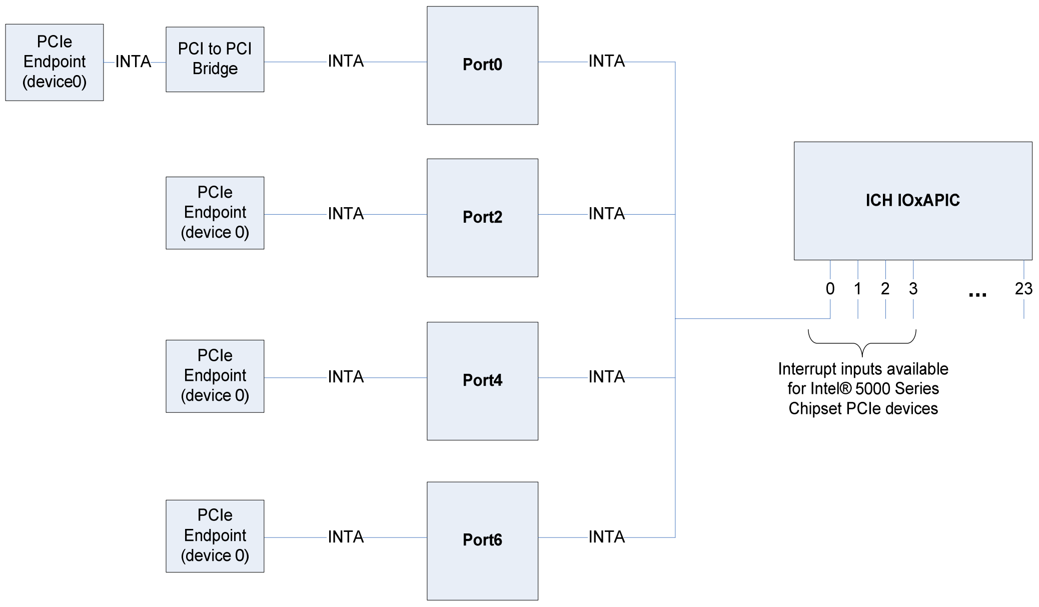
However, most ports have only one device with device number 0 that results in identity mapping of the interrupt (INTA → INTA, INTB → INTB,…). As a result, if root ports mapped the downstream interrupts messages as is to the interrupt controller, all PCIe interrupts will likely be mapped to a single input of the interrupt controller.

The Intel 5000 Series Chipset implements interrupt swizzling logic to rebalance and distribute the PCIe legacy interrupts for performance and load balancing. This document describes the interrupt swizzling scheme in detail and discusses the programming requirements to implement this scheme.

2 Interrupt Swizzling Overview

Consider a platform configuration where Port 0, 2, 4, 6 are populated with PCIe devices. The default mapping of PCI INTx messages to system interrupt controllers is as shown in Figure 1.

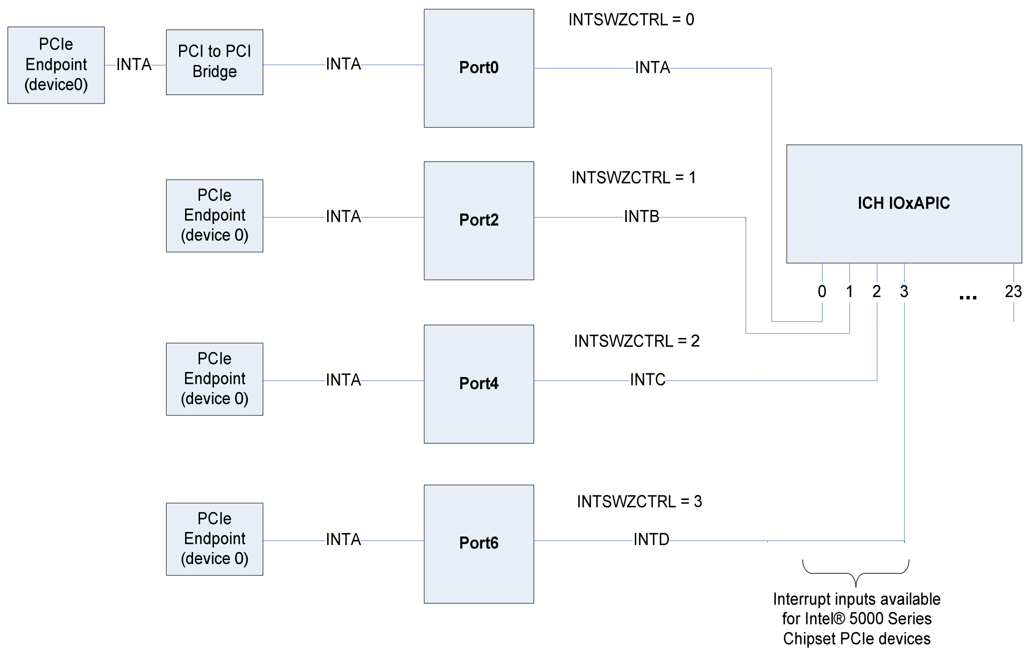
Figure 1. Interrupt Routing without Interrupt Swizzling



As illustrated above, the default mapping results in mapping of all 4 PCIe devices (assigned to device number 0) to the same interrupt although Intel® 5000 Series Chipsets supports 4 unique interrupts.

The interrupt mapping for the same platform configuration with optimal interrupt swizzling enabled is shown in Figure 2.

Figure 2. Interrupt routing with Optimal Interrupt Swizzling



3 Interrupt Swizzling Scheme Limitations

The effectiveness of Interrupt swizzling scheme is dependent on the following:

System BIOS that programs the interrupt swizzling logic must comprehend the PCIe configuration of the platform and customize the programming of the swizzling logic accordingly. This includes comprehending all populated PCIe ports, devices as well as hot plug-capable slots supported on the platform.

The Intel 5000 Series chipset supports a maximum of 4 unique interrupt messages to the system interrupt controller (ICH IOxAPIC). As a result, the interrupt swizzling can still result in sharing of interrupts between devices if the PCIe hierarchy contains more than 4 devices.

4 Programming for Interrupt Swizzling

The Intel 5000 Series chipset supports 6 x4 PCIe ports and an ESI port for a total of 7 ports. For each of these 7 ports, the interrupt mapping for that port to the system interrupt controller is controlled by programming the corresponding INTXSWZCTRL register. The register is described Table 1.

Table 1 Interrupt Swizzle Control Register on PCIe Ports of Intel® 5000 Chipset

PCIe Port	DID	BDF Mapping	INTXSWZCTRL offset
0 (ESI)	SKU dependant	B:0, D:0, F:0	4Fh
2 (PCIe)	25E2h	B:0, D:2, F:0	4Fh
3 (PCIe)	25E3h	B:0, D:3, F:0	4Fh
4 (PCIe)	25E4h	B:0, D:4, F:0	4Fh
5 (PCIe)	25E5h	B:0, D:5, F:0	4Fh
6 (PCIe)	25E6h	B:0, D:6, F:0	4Fh
7 (PCIe)	25E7h	B:0, D:7, F:0	4Fh

INTXSWZCTRL register (bits 1:0, other bits reserved) supports 4 combinations of interrupt mapping using the barber-pole stride mechanism.

4.1 BIOS

The default value of INTXSWZCTRL at reset is 0h corresponding to identity mapping of interrupts (INTA → INTA,…). System BIOS is required to program the INTXSWZCTRL register for each port to ensure that the interrupts load is balanced across the available system interrupts. The Intel® 5000 Series Chipset supports a maximum of 4 unique system interrupts for the PCIe hierarchy. System BIOS must program the INTXSWZCTRL register of each populated port on the platform for optimal performance and load balancing.

For example, in a platform where Ports 0, 2, 4, 6 are populated with PCIe devices, the system BIOS could program the INTXSWZCTRL register for ports 0, 2, 4, and 6 as described in Table 2.

Table 2 Example Interrupt Swizzling configuration

Port	INTXSWZCTRL value	Resulting Interrupt mapping
0	00h	INTA→INTA INTB→INTB INTC→INTC INTD→INTD
2	01h	INTA→INTB INTB→INTC INTC→INTD INTD→INTA
4	02h	INTA→INTC INTB→INTD INTC→INTA INTD→INTB
6	03h	INTA→INTD INTB→INTA INTC→INTB INTD→INTC

4.2 OS

The mapping of PCIe virtual interrupt (INTx) to system interrupt is indicated to the OS using firmware tables. OSes that comprehend ACPI _PRT method, parse _PRT to identify the mapping. Older OSes use MPS1.4 table method to identify the mapping.

System BIOS that implement interrupt swizzling must comprehend the effect of swizzling when constructing the _PRT or MPS1.4 interrupt mapping table. OS itself does not require any special handling to support interrupt swizzling mechanism.

4.3 Performance Benefits of Interrupt Swizzling on Intel 5000 Series Chipset

This section utilizes a networking stack running on Linux2.4 is to illustrate the performance benefits incurred by adopting interrupt swizzling scheme on platforms based on the Intel 5000 series chipset. However, the benefits of the interrupt swizzling scheme is neither limited to a specific IO stack nor to a specific OS.

With highly threaded applications, such as database and web server applications, the OS is responsible for balancing the active (ready) threads among the available CPU cores. Software developers optimize the granularity of application threads with the goal of minimizing the overheads of multithreading (for example, thread switching, migrating threads to balance the use of CPUs) while maximizing parallelism offered by multithreading. Kernel level applications and especially networking stacks are architected to be single threaded for various reasons:

Networking is layered directly above the hardware (that is, networking interface controllers, or NICs).

Networking stacks typically run in the most privileged mode allowed by the operating system (for example, SoftIRQ mode in Linux and Deferred Procedure Call mode in Windows).

In order to keep pace with high speed HW interfaces (for example, Gigabit Ethernet), networking stacks can not afford to be burdened with the overheads associated with multithreading.

Note: The following discussion assumes Linux (2.4 and later versions) as the underlying Operating system. Please note, Linux 2.4 does not support MSI (PCISIG defined Message Signaled Interrupt).

In a typical network processing scenario, a network packet is received over an Ethernet port. The NIC controlling that port deposits the packet in the memory and signals the CPU through a HW interrupt. The CPU runs an interrupt service routine (ISR) to attend the HW interrupt and update the HW status of the network device. At the end of the ISR, the CPU queues the follow-up work for processing the received packet(s) by signaling a SoftIRQ. SoftIRQs, being highly privileged threads in Linux, are closely guarded resources. The networking stack has one SoftIRQ permanently assigned in the architecture of Linux kernel. SoftIRQs are a per CPU resource, hence there is one networking SoftIRQ per available core in SMP platforms.

As mentioned, all packet processing, that is, queueing and dequeing packets from the network interface, IPv4/IPv6 processing and TCP/UDP processing in Linux happens in the SoftIRQ context. In fact, several layer 3 and layer 4 networking functions such as firewall, VPN, proxy and intrusion detection are processed in the same SoftIRQ context upon receiving a packet.

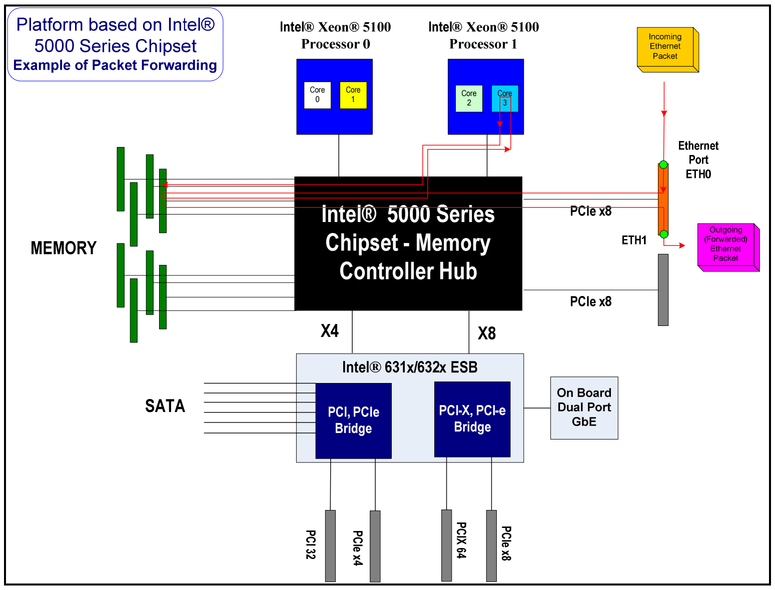
Since SoftIRQs are CPU specific and are triggered through ISRs for specific HW devices, functions such as routing (or layer 3 forwarding) have affinity at the software level to a specific CPU core. This is the same core that receives the hardware interrupt upon receipt of a packet.

Linux on Intel architecture platforms offers three choices for routing interrupts from network devices to CPU cores:

1. ***Default routing***: all interrupts are routed the same CPU core (typically core 0).
2. ***Stack interrupt routing***: Interrupts from a HW device are preferentially routed to the same CPU core always.
3. ***IRQ Balance***: the IOAPIC in the chipset distributes all interrupts (from all devices) among available cores in a round robin fashion.

While it is intuitive that IRQ Balance should offer the best load balance and optimal throughput under high network throughput conditions, computationally light weight functions such as IPv4 forwarding are observed to achieve peak throughput through static interrupt routing. Also, for statically defined packet flows (that is, the receiving and transmitting NICs are fixed), static binding of both NICs (that is, their HW interrupts to CPU cores in same socket) is known to maximize performance. This is due to the hardware overhead involved in synchronizing the Transmit (TX) and Receive (RX) sides of the flow in an SMP context. Figure 3 shows the flow of a packet through an IPv4 forwarding stack on a uniprocessor (UP) IA32/Linux platform: a packet arrives through Ethernet interface eth0 (receiving port), is processed by the CPU and then sent through the transmitting (TX) Ethernet interface eth1.

Figure 3 . Example of Network Packet Forwarding



In a multiprocessor/multi-core platform with multiple NICs, peak performance requires packets received over all NICs be processed by distributing them evenly among the available cores. Based on empirical data, we recommend the following rules of static interrupt binding for NICs to maximize the performance of kernel level networking functions:

1. Each NIC would have its HW interrupt/bound to a different CPU core. The transmitting and receiving NICs of a given flow would be assigned to a pair of cores in the same CPU socket.
2. When the number of NICs exceeds the number of CPU cores, then assign the IRQs of the transmitting and receiving NICs of a given flow to the same CPU core, while distributing different flows among available cores.
3. The transmitting and receiving sides of a flow should be allocated to cores in different CPU sockets only when (1) and (2) cannot be satisfied completely.

Figure 4 illustrates rule (1) with the schematic of a platform based on Intel 5000 series chipset.

Figure 5 illustrates rules (1) and (2) with the schematic of a platform based on Intel 5000 series chipset.[1]

[1] We observe that rule (3) can be put into use only the rare case when the available flows are some how skewed in CPU assignments & hence it might be necessary to allocated cores from different socket. Hence case (3) is not illustrated here.

Figure 4 Example of Static Affinity assignment for IRQs (each NIC assigned a different core)

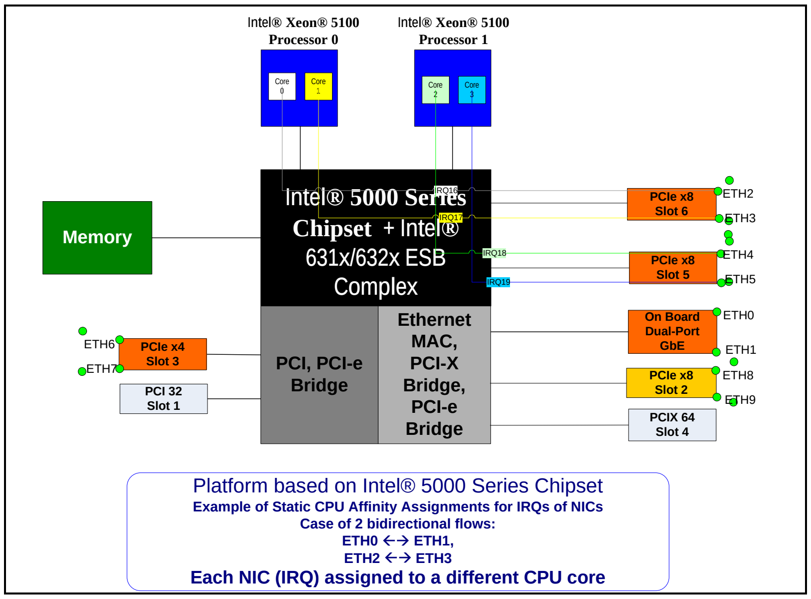
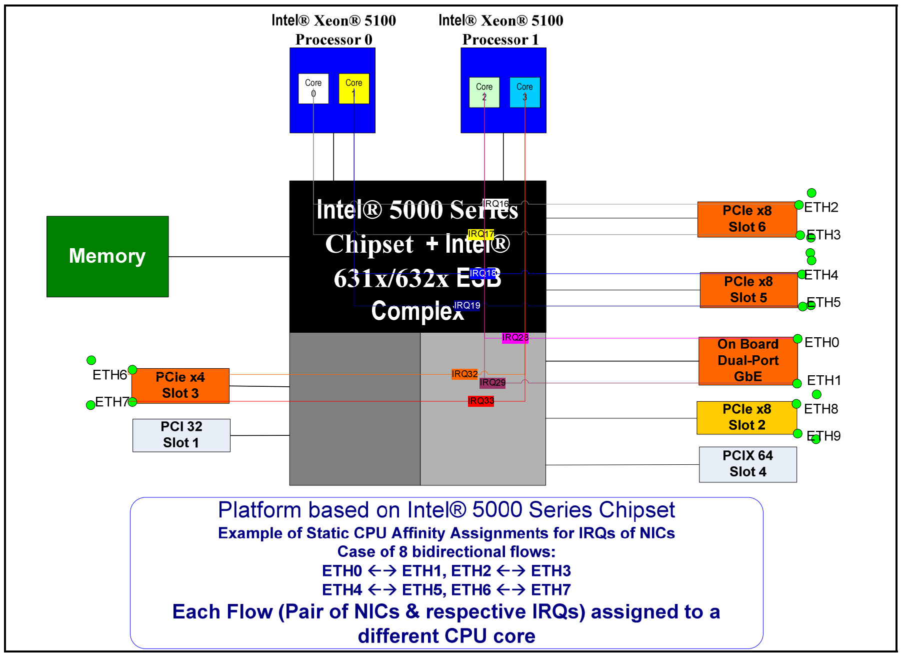

Intel® Xeon® 5100 Processor 0
Intel® Xeon® 5100 Processor 1
Core 0
Core 1
Core 2
Core 3
Memory
Intel® 5000 Series Chipset + Intel® 631x/632x ESB Complex
IRQ16
IRQ17
IRQ18
IRQ19
PCI, PCI-e Bridge
Ethernet MAC, PCI-X Bridge, PCI-e Bridge
PCIe x8 Slot 6
PCIe x8 Slot 5
On Board Dual-Port GbE
PCIe x8 Slot 2
PCIX 64 Slot 4
PCIe x4 Slot 3
PCI 32 Slot 1
ETH0
ETH1
ETH2
ETH3
ETH4
ETH5
ETH6
ETH7
ETH8
ETH9
Platform based on Intel® 5000 Series Chipset
Example of Static CPU Affinity Assignments for IRQs of NICs
Case of 2 bidirectional flows:
ETH0 <-> ETH1,
ETH2 <-> ETH3
Each NIC (IRQ) assigned to a different CPU core

Figure 5 Example of Static Affinity assignment for IRQs (each flow assigned a different core)


Intel® Xeon® 5100 Processor 0
Intel® Xeon® 5100 Processor 1
Core 0
Core 1
Core 2
Core 3
Memory
Intel® 5000 Series Chipset + Intel® 631x/632x ESB Complex
IRQ16
IRQ17
IRQ18
IRQ19
IRQ28
IRQ32
IRQ29
IRQ33
PCIe x8 Slot 6
PCIe x8 Slot 5
On Board Dual-Port GbE
PCIe x8 Slot 2
PCIX 64 Slot 4
PCIe x4 Slot 3
PCI 32 Slot 1
ETH0
ETH1
ETH2
ETH3
ETH4
ETH5
ETH6
ETH7
ETH8
ETH9
Platform based on Intel® 5000 Series Chipset
Example of Static CPU Affinity Assignments for IRQs of NICs
Case of 8 bidirectional flows:
ETH0 <-> ETH1, ETH2 <-> ETH3
ETH4 <-> ETH5, ETH6 <-> ETH7
Each Flow (Pair of NICs & respective IRQs) assigned to a different CPU core

Since IRQs and their static CPU affinities are the basis for balancing available CPU cores among packet flows, it is easy to see why a poor IRQ assignment by the firmware could result in a poor utilization of CPU cores and poor forwarding throughput in turn. In the extreme case when all the NICs are assigned to the same IRQ, any IRQ binding policy (whether static, default or IRQ balance) would result in all packets (from all the NICs) being processed by the same CPU core – netting a 75% wastage of CPU resources in a Woodcrest/Bensley platform.

Table 3 shows the IRQ assignments on a sample platform based on Intel® 5000 series chipset that contains three of Intel 82571 dual port PCI Express NICs plugged into the three available PCIe slots, besides the on board dual port NIC on board.

Table 3 IRQ Assignment on Platform based on Intel ® 5000 series chipset - with and without Interrupt Swizzling

Swizzle Enabled?	PCIe Slot Assignments to NIC Ports								IRQ Assignments to NIC Ports			
	On Board Dual-Port GbE NIC		Intel® 5000 Series Chipset PCIe x8 Slot (6)		Intel® 5000 Series Chipset PCIe x8 Slot (5)		631x/632x PCIe x4 Slot (4)		IRQ16	IRQ17	IRQ18	IRQ19
NO	eth0	eth1	eth2	eth3	eth4	eth5	eth6	eth7	eth6 eth2 eth4	eth7 eth3 eth5	eth0	eth1
YES	eth0	eth1	eth2	eth3	eth4	eth5	eth6	eth7	eth6 eth2	eth7 eth3	eth0 eth4	eth1 eth3

In the original assignment when swizzle is not implemented, the Ethernet ports (eth2, eth3, eth4 and eth5) attached to Intel 5000 Series Chipset are assigned to IRQs 16 and 17. Thus, any flow configuration involving ports eth2, eth3 and eth4 and eth5 can take advantage of only two of the 4 available cores. Even when all the ports eth0 through eth7 are involved, it is not possible to engage all four cores without overloading two of the cores with two additional flows, while two other cores would be handle only one flow (together). By swizzling (row 2), the IRQ assignments to eth4 and eth2 are spread out to IRQ18 and IRQ19. This makes for an even spread of servicing NICs among the cores; even when considering only the 4 ports attached to Intel® 5000 Series Chipset, it is now possible to engage all the 4 CPU cores evenly (consequently maximizing forwarding throughput). Table 4 illustrates this performance improvement in forwarding throughput on Linux 2.4.21 (RHEL3).

Table 4 IPv4 Forwarding Throughput in Packets Per Second on Platform using Intel® 5000 series chipset (with and without Interrupt Swizzling)

Color Code for Core Assignments					
	⬜ Core 1	🟨 Core 1	🟩 Core2	🟦 Core3	▨

BIOS Version	# Ethernet Ports	Flows	Assignment of cores to Ethernet ports								Throughput Small (64B) Packets/ Sec	Comment
			On Board Dual-Port GbE NIC		INTEL® 5000 Series Chipset PCIe x8 Slot (6)		INTEL® 5000 Series Chipset PCIe x8 Slot (5)		631x/632x PCIe x4 Slot (4)			
NO SWIZZLE	4	eth2 ↔ eth4 eth3 ↔ eth5	▨ eth0	▨ eth1	⬜ eth2	🟨 eth3	⬜ eth4	🟨 eth5	▨ eth6	▨ eth7	1,413,690	
NO SWIZZLE	4	Eth0 ↔ eth1 Eth2 ↔ eth3	⬜ eth0	🟨 eth1	🟩 eth2	🟦 eth3	▨ eth4	▨ eth5	▨ eth6	▨ eth7	2,585,565	
SWIZZLE	4	eth2 ↔ eth3 eth4 ↔ eth5	▨ eth0	▨ eth1	⬜ eth2	🟨 eth3	🟩 eth4	🟦 eth5	▨ eth6	▨ eth7	2,626,023	Improved Interrupt assignment facilitates the use of all 4 cores.
NO SWIZZLE	8	eth0 ↔ eth4 eth1 ↔ eth5 eth2 ↔ eth6 eth3 ↔ eth7	Due the concentration of NIC ports IRQs 16 and 17, it is not possible to distribute the flows evenly among the available cores. Best achievable throughput occurs with 6 ports & 3 flows (Not shown here)									
SWIZZLE	8	eth0 ↔ eth4 eth1 ↔ eth5 Eth2 ↔ eth6 eth3 ↔ eth7	⬜ eth0	🟨 eth1	🟩 eth2	🟦 eth3	⬜ eth4	🟨 eth5	🟩 eth6	🟦 eth7	2,836,681	

While it is possible to achieve near peak throughput of 2.6 Mpps (with 4 cores) without interrupt swizzling, swizzling allows a wider choice of flow configurations that achieve peak performance up to 2.8 Mpps.