教育部補助博士班研究生出席國際會議報告

研發處補助序號：

<table>
<thead>
<tr>
<th>姓 名</th>
<th>蔡文嚴</th>
<th>會議期間</th>
<th>2012/12/02 ~ 2012/12/07</th>
</tr>
</thead>
</table>

會議名稱
中文：2012 全球通訊會議
英文：IEEE Global Telecommunications Conference 2012

受補助項目
(敬請依核定申請表填寫)

<table>
<thead>
<tr>
<th>機票費：</th>
<th>註冊費：1,217</th>
<th>生活費：11,423</th>
</tr>
</thead>
<tbody>
<tr>
<td>合計總金額：12,640</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

出國報告
(至少兩頁以上)

1. 會議內容
今年於美國 Anaheim Disneyland Hotel 舉行的 IEEE Globecom 2012 從來自世界研究學者投稿的 2,560 篇論文中接受包含口頭 (oral) 及海報 (poster) 報告共 966 篇，總和接受率為 38.5%。會議第一天的主要活動為 welcome reception，透過簡單的晚宴歡迎來自各地的研究學者。接下來三天則由每天一早來自業界或學界 keynote speaker 的演講開始，接著 technical symposia 和與之平行的 industry forum，與會者可以自行選擇要參加的場次。值得一提的是要進入每一個會場之前，都需要讓門口的工作人員掃描識別證上的 2-D 條碼，安全措施做的比較嚴謹。

2. 心得
我的論文被分配到 NGNI 的 router & switch architecture 這個
session 中，而 session chair 是跟我研究領域相關的大師 Norbert Egi，其在 high-speed packet forwarding system/architecture 上發表許多經典的論文，但最近從學界轉到業界。會後與其交談得知其研究方向有了一些轉移，但還是持續關注原來的研究領域。這次 presentation 的結果因為時差的關係，還算差強人意，報告完之後也只有 session chair 問了一個問題。同一 session 中的報告比較感興趣的是有關 packet classification 中如何有效率的安排 memory layout 以達較好的 memory locality 及較高的 performance；另一個則是對傳統 RED 演算法的改良，其注重在 dynamic network 中利用 feedback 來調整 RED 的參數，以期在各個不同動態環境達到較好的效果。會議中也參加了一個關於 SDN (Software Defined Network) / openflow 的 industry forum，與會的講者包含美國 ISP、光學元件公司及 Juniper Networks 的業界專家，介紹其對 SDN/openflow 目前的技術及未來的研發規劃，會中看到的一些資料數據頗值得參考及啟發新的研究方向。對整體會議的心得是收穫豐富，有機會與同領域的大師及許多專家交流，同時也了解目前業界對 SDN/openflow 的動態；美中不足的是會場交通不方便，附近只有 Disneyland 及一個附屬的 downtown，在只靠步行的方式下，吃住方面沒有太多選擇。

3. 檢討
不足的地方還是英語聽說，如果對方速度太快或帶有口音，就很難聽懂他要表達的內容；另一方面，因為平常接觸的英文都是學術或技術文章，因此口語交談時都會用比較冗長或對外國人來說很奇怪的說法，雖然對方可以猜出我的意思，但溝通不免彆扭。日後在學術閱讀寫作之外還要加強英語口語聽說的訓練。另一點則是建立自己的學術人脈，擴大可用的研究資源。目前熟悉的研究同儕都是國內學者，藉由這次會議雖然可以認識不少如大陸或西方的同行，但通常侷限於初步會面及名片交換，回國之後就沒有繼續保持聯絡。事實上，不少高質量的論文都是來自不同國家的研究者協同合作的成果，因此我想更積極的與國外研究者建立人脈關係對自己研究的廣度及深度都會有很大幫助。

※受補助之學生請於回國後將此表填妥後上傳至校務資訊系統，並點選開放報告後，始得辦理後續核銷作業。
A Port-Configuration Assisted NIC IRQ Affinitization Scheme for Multi-Core Packet Forwarding Applications

Wen-Yen Tsai\textsuperscript{1}, Nen-Fu Huang\textsuperscript{1,2} and Hsien-Wei Hung\textsuperscript{3}

Institute of Communications Engineering\textsuperscript{1}, Department of Computer Science\textsuperscript{2}
National Tsing Hua University, Taiwan
Broadweb Corporation, Taiwan\textsuperscript{3}

\textbf{Abstract} — Interrupt affinitization of network interface cards (NICs) is a fundamental composition that defines how packets are processed by which CPU-cores on multi-core platforms. In this paper, we propose a simple port-configuration assisted scheme to attain an optimal affinitization for packet forwarding applications. Experiments ranging from bridging, routing, flow tracking to deep packet inspection are conducted to show the performance impacts utilizing different affinitization approaches. As a result, our proposed scheme achieves the same performance level as the best fixed affinitization scheme. In addition, the effectiveness of interrupt balancing is demonstrated for our scheme to be superior to the widely-deployed \textit{irqbalance} with varying network settings.

\textbf{Index Terms} — IRQ affinitization, multi-core computing, IRQ balancing.

\section{I. INTRODUCTION}

Multi-core powered platforms view their pervasive deployment in today’s computing environments from a wide range of small mobile devices to supercomputers. To utilize the massive computing power, both hardware-IO components and system software need to be enhanced in order to cooperate with the CPU-cores in a more efficient way. Interrupt affinitization is the key that decides which CPU-core is responsible for processing an interrupt from the peripheral. Take a NIC for example, once a packet is received, the NIC hardware uses the Direct Memory Access (DMA) channel to transfer data to system memory and then raises its interrupt request line (IRQ) to notify the programmable interrupt controller (PIC) of the packet reception. The PIC is programmed in advance according to the IRQ affinitization setting and knows how to direct the interrupt to the target CPU-core. Besides the affinitization process that maps interrupts to CPU-cores, the interrupt processing model is basically the same on both single and multi-core platforms. On single-core platforms, all interrupts are affinitized to the only core; however, it is nontrivial to decide an optimal affinitization on multi-core platforms for packet forwarding applications in face of several challenges. First, state-of-art CPU and NIC hardware equips with advanced features that complicate the affinitization process.

- Modern NICs incorporate hardware features like multiple Rx/Tx queues, advanced interrupt mechanisms, i.e., PCI MSI and MSI-X, and flow classifier for steering packets belonging to a same flow to a single Rx-queue. Besides, the receive side scaling (RSS) innovation provide users the flexibility to configure the number of Rx queues and how to direct packets of a same flow to a queue based on some hashing calculation results. From the perspective of IRQ affinitization, each NIC queue is designated by an IRQ number that is to be affinitized to some CPU-core(s).

- Modern systems arrange CPU-cores in a hierarchical way. Starting from the machine board level, to avoid a single memory channel shared by all the CPUs becoming the bottleneck, scalable systems employ Non-Uniform Memory Access (NUMA) architectures that separate each CPU-package to a node with its own memory channel. Inside each CPU-package, CPU-cores that share the same last level cache (LLC) are in the same cache domain. Moreover, if hardware threading is supported, threads (logical cores) that belong to a same physical core are in the same core level.

In addition, system software and/or the operating system accommodate the facility to configure the IRQ affinitization via divergent interfaces and mechanisms. Also, owning to different requirements between performance and flexibility of network applications, the packet processing may occur in either user space or directly in the OS kernel. Last, the behavior of different networking applications varies dramatically, therefore, an optimal affinitization for packet forwarding application in the kernel is deemed far from optimal for a server application running in the user space. To conquer these difficulties, we propose a systematic approach to affinitize NIC IRQs to CPU-cores with the prior knowledge of port configuration for packet forwarding platforms. Applications ranging from bridging, routing, firewalling to deep packet inspection are studied to show the performance gains of our scheme. Besides, dynamic interrupt balancing is demonstrated to get insight of how the proposed affinitizer works for unbalanced traffic models.

The rest of this paper is organized as follows. In Section II, we introduce the existing IRQ affinitization schemes. Then, our proposed port-configuration assisted affinitization approach is elaborated in Section III. Both the affinitization cost and the operation complexity of our proposed scheme is analyzed in Section IV. In Section V, the experimental results are presented. Finally, we conclude our work in Section VI.

This work was supported by National Science Council (NSC) of Taiwan under the grant numbers NSC-100-2221-E-007-104 and NSC-100-2221-E-007-123.
Although in recent Linux kernels the in-kernel IRQ balancer is irqbalance implementations, the Linux in-kernel IRQ balancer and Intel’s systems is discussed in [10] and [11]. Two practical problems of adaptive load balancing on multiprocessor systems is also suffer performance degradation if the interrupt load is dynamically distributed among CPU-cores to achieve communication-intensive processes. While their research is similar to us, they focus on network applications on end-hosts, i.e., servers, but not packet forwarding platforms. A transport-friendly NIC design by Wu et al. [8] intelligently affinitizes IRQs of active NICs to one or more CPU-cores statically. The major drawback of static affinitization lies in its lack of dynamic interrupt balancing. Jang and Jin [7] proposed the MiAMI process scheduler that utilizes knowledge of NIC affinity to help select the optimal CPU-core to execute protocol-intensive processes.}

**II. RELATED WORK**

Salehi et al. are among the first researchers to take affinity into account for effective process scheduling for protocol processing on multiprocessor platforms. Foong et al. [5][6] have analyzed how NIC affinity impacts the network performance by employing a static affinitization setup, which affinitizes IRQs of active NICs to one or more CPU-cores statically. The major drawback of static affinitization lies in its lack of dynamic interrupt balancing. Jang and Jin [7] proposed the MiAMI process scheduler that utilizes knowledge of NIC affinity to help select the optimal CPU-core to execute communication-intensive processes. While their research is similar to us, they focus on network applications on end-hosts, i.e., servers, but not packet forwarding platforms. A transport-friendly NIC design by Wu et al. [8] intelligently affinitizes interrupts of NICs to CPU-core(s) on which the target server or client application resides by passing hints from the application to the NIC driver. Their idea is later implemented in Intel’s 82599 10G Ethernet controller and termed “flow director” [9]. However, it is hard to coordinate this smartness with in-kernel packet forwarding applications since there is no concept of process when the packet processing takes place in the interrupt context (more precisely, softirq context in Linux). On the other spectrum, dynamic affinitization schemes distribute loads of interrupt processing among CPU-cores to obtain balanced overall system utilization. However, to mitigate the problem of CPU cache-line bounces, dynamic affinitizers prefer previously pinned cores except those cores are over-utilized. Dynamic affinitization schemes also suffer performance degradation if the interrupt load is distributed in a careless way without considering the relationship between hardware peripherals and CPU topology. The problem of adaptive load balancing on multiprocessor systems is discussed in [10] and [11]. Two practical implementations, the Linux in-kernel IRQ balancer and Intel’s irqbalance [1] tool are widely-deployed dynamic affinitizers, although in recent Linux kernels the in-kernel IRQ balancer is permanently dropped in favor of irqbalance.

**III. PORT-CONFIGURATION ASSISTED IRQ AFFINITIZATION**

For the reason of maximum performance, many high-speed packet processing applications execute their code in kernel mode. Therefore, it is impossible to affinitize the interrupt of a NIC to a CPU-core where a user-mode packet processing application resides. Nevertheless, due to the relative invariance of the port-configuration of these applications, NICs in a same functional group of a specific port-configuration should be affinitized to CPU-core siblings because 1) no extra CPU-core has to be dedicated to dispatch the incoming packets to other cores for processing, 2) connection and cache localities can be reserved, 3) improper cross NUMA-node or CPU-package affinitization can be alleviated. Based on this reasoning, we enhance irqbalance by factoring in the knowledge of port configuration along with the hardware information of CPU and NIC. The resulting affinitizer achieves optimal packet forwarding performance while preserving the benefit of dynamic interrupt balancing.

We first collect and analyze information of all the NIC interrupts and CPU topology on the packet forwarding platform. Information of each interrupt is represented by a nic_interrupt structure. To facilitate the representation of each CPU-core in the system, we define the cpu_mask_t type which is a bitmap with each bit indicating a core, i.e., bit- \( n \) for the \( (n+1) \)-th core. Ergo, a numa_mask is the bitmap of cores that are “local” to an interrupt from the NUMA’s perspective on a specific system. The pref_mask contains the preferred CPU-cores according to the current port-configuration. For example, Fig. 1 illustrates a quad-core system with a single CPU-package within which two cores share a same LLC. The system is configured as (a) two bridges or (b) a 3-leg firewall. In Fig. 1 (a), we set the pref_mask of NIC groups belonging to the two bridges to 0x03 and 0x0c to favor CPU-cores \{1, 2\} and \{3, 4\}, respectively. On the other hand, for the 3-leg firewall configuration, the pref_mask of NIC groups in the first bridge is set to 0x0f and let one of NIC (NIC55) share a single CPU-core with its counterpart as illustrated by dash lines.

Similarly, the CPU topology is abstracted by three structures, package, cache_domain, and cpu_core with their most important fields listed in Table I. In addition, we employ the
workload field in each structure as the major metric for performing dynamic interrupt balancing.

In Scheme I, information of all CPU-packages and NIC-interrupts in the system is packed in the nic_interrupt and package structures and they are linked in the L_packages and L_interrupts list. In setup_intr pref_mask(), for each port group, grp, we set the value of grp.pref_mask according to the number of ports in grp and the CPU topology. A bottom-up approach is followed to set grp.pref_mask from the CPU-core level to gradually increase 1’s in grp.pref_mask if the number of ports in grp, N(grp), is larger than the number of CPU-cores on the current level of CPU topology. Here the hweight() function is used to calculate the Hamming weight, i.e., the numbers of 1’s, of a bitmask. After the for loop, we have to check if N(grp) is zero, which means a single CPU-package doesn’t have sufficient cores to afford a one-to-one affinitization for all the ports in grp. In this case, we just confine the affinitization of this port group to the CPU-package level and rely on interrupt balancing to dynamically distribute load of the package if it is overloaded. To explicitly enable interrupt balancing in the latter affinitization phase, the global flag, do irq balance, is set on line 11. Finally, for each interrupt, intr, in L_interrupts, we find the port group containing the NIC from which intr originates and set intr.pref_mask accordingly.

Each interrupt in Linterrupts is then classified as one of the three classes: IRQ_ETH, IRQ_GETH, and IRQ_TGETH for fast, Gigabit and 10G Ethernet NIC, respectively. Also, for a given interrupt, in compute_workload(), we compute the delta of the number of interrupts generated on it and, more specifically, the volume of received and transmitted packets (rxtx_count) to update its workload. For systems handling millions of packets per second, interrupt-based packet reception mechanism is considered inefficient and results in the “livelock” problem. To deal with this difficulty, polling-based packet reception approach is proposed and adopted in modern OS. Take Linux for example, it provides a new API (NAPI) interface to NIC drivers to facilitate implementation of this polling paradigm. Since interrupt is replaced by polling, the number of interrupts of a NIC (irq_count) can no longer fully expresses the load of a NIC, we therefore have to account for the packet count when

### TABLE I Structures used in the proposed affinitization scheme

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>old_mask, mask</td>
<td>cpumask_t</td>
<td>bitmaps of the previously and currently pinned cores</td>
</tr>
<tr>
<td>numa_mask</td>
<td>cpumask_t</td>
<td>bitmap of “local” cores specific to the system’s NUMA config.</td>
</tr>
<tr>
<td>pref_mask</td>
<td>cpumask_t</td>
<td>bitmap of preferred cores according to port-configuration</td>
</tr>
<tr>
<td>class</td>
<td>int</td>
<td>interrupt class by the NIC type</td>
</tr>
<tr>
<td>workload</td>
<td>uint64_t</td>
<td>load in terms of the number of generated interrupts and transceived packets</td>
</tr>
</tbody>
</table>

### Field specific to the package structure

| cache_domain | list | linked list of cache domains belonging to this package |

### Field specific to the cache_domain structure

| cpu_cores | list | linked list of cores belonging to this cache_domain |

### Auxiliary methods for Scheme I

**Method: setup_intr pref_mask(L_interrupts)**
1. port_grp, ∈ all interrupts of NICs in the same port group
2. PG = { port_grp, } for PG = PG
3. for grp in PG
4. grp.pref_mask 0
5. for i in { cpu_core, cache_domain, package }
6. if N(grp) ≤ hweight(mask)
7. grp.pref_mask lmask
8. break
9. if grp.pref_mask is zero
10. grp.pref_mask package.mask
11. do irq balance ω true
12. for each intr in L_interrupts
13. grp the port group to which intr belongs
14. intr.pref_mask = grp.pref_mask

**Method: compute_workload(intr)**
1. rxtx_count # received and transmitted packets on intr
2. intr.rxtx_delta (rxtx_count – intr.prev_rxtx) / 2
3. intr.workload (rxtx_count – intr.ance_rxtx) + 1/3
4. intr.prev_rxtx = rxtx_count

**Method: compute_affinitization(intr, L_packages)**
1. best_pkg p’ s.t. ω(p’, intr) = min {ω(p, intr), ∀ p in Lpackages}
2. Lcache.dom best_pkg.cache.domains
3. best_dom d’ s.t. ω(d’, intr) = min {ω(d, intr), ∀ d in Lcache.dom}
4. Lcpu.core best_dom.cpu.cores
5. best_core c’ s.t. ω(c’, intr) = min {ω(c, intr), ∀ c in Lcpu.core}
6. return best_core.mask

### Scheme I. Port-configuration assisted affinitization scheme

**Method: affinitize_IRQ**
1. L_packages list of all package structures of the system
2. L_interrupts list of all nic_interrupt structures of the system
3. while true
4. setup_intr pref_mask(L_interrupts)
5. for each interrupt intr in L_interrupts
6. intr.class interrupt class of intr
7. intr.workload compute_workload(intr)
8. Linterrupts sort L_interrupts by class and workload
9. for each interrupt intr in Linterrupts
10. intr.mask compute_affinitization(intr, L_packages)
11. activate_affinitization(intr)
12. sleep(AFF_INTL)
TABLE II. Affinitization cost parameters for an interrupt, \( intr \)

<table>
<thead>
<tr>
<th>Cost parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \omega_{\text{topology}} )</td>
<td>1,000 on core level</td>
</tr>
<tr>
<td>( \omega_{\text{topology}} )</td>
<td>1,500 on cache_domain level</td>
</tr>
<tr>
<td>( \omega_{\text{topology}} )</td>
<td>3,000 on package level</td>
</tr>
<tr>
<td>( \omega_{\text{NUMA}} )</td>
<td>500</td>
</tr>
<tr>
<td>( \omega_{\text{port-config}} )</td>
<td>50,000 on package level for all other cases</td>
</tr>
<tr>
<td>( \omega_{\text{class}} )</td>
<td>1,000 * ( N_{\text{class}} )</td>
</tr>
</tbody>
</table>

Calculating the workload of a NIC. However, to prevent this large value of packet count from outweighing other factors that influence the affinitization, it is smoothed by cut in half. As a result, \( intr.\text{workload} \) is a weighted value of the current number of interrupts and its history plus the smoothed packet count.

Then, we affinitize the NIC interrupts in a prioritized manner by sorting \( L_{\text{interrupts}} \) to \( L_{\text{interrupts}} \), according to the class and workload of each interrupt. In other word, NICs capable of higher speed or possessing heavy workload are affinitized first. For each \( intr \) in \( L_{\text{interrupts}} \), \( \text{compute_affinitization()} \) manipulates its various mask fields and those of the package/cache_domain/cpu_core structures derived from \( L_{\text{packages}} \) to compute a mask for optimal affinitization. On each level of the CPU hierarchy, we select a target with the minimal affinitization cost by a formula that is to be elaborated in section IV. The exact way to activate the affinitization depends on the OS. On Linux, we can affinitize an interrupt to the intended CPU-core(s) by simply writing a file in the special \( \text{proc}/\text{irq} \) pseudo-filesystem. For instance, to affinitize NIC#1 to CPU-core#1, one single command suffices: $ echo 01 > /proc/irq/IRQ_of_NIC#1/smp_affinity. Finally, the affinitization interval, \( \text{AFF\_INTL} \), is a trade-off amid the computing overhead of this periodical affinitization process and the effectiveness of interrupt balancing. It can be configured dynamically or left unchanged depending on the characteristic of system load.

IV. ANALYSIS

A. Affinitization Cost

We calculate the affinitization cost, \( \omega \), of an interrupt, \( intr \), for each level of the CPU hierarchy, \( l \), with the formula:

\[
\omega = \frac{(\omega_{\text{topology}} + \omega_{\text{NUMA}} + \omega_{\text{port-config}})}{adj + \omega_{\text{class}}} + \frac{(\text{intr.\text{workload}} + \text{intr.\text{load}}) \times \text{irq.balance}}{\text{hweight}_{\text{intr} \cdot \text{pref} \cdot \text{mask}}} \times \text{hweight}_{\text{intr} \cdot \text{pref} \cdot \text{mask}}
\]

(\( \omega_{\text{topology}} \)) represents the penalty of migrating an interrupt from its currently pinned CPU cores to different ones, i.e., \( \text{intr.\text{old}} \cdot \text{mask} \neq \text{intr.\text{mask}} \). \( \omega_{\text{NUMA}} \) denotes the cost of not honoring the specific NUMA locality for a NIC, that is, \( \text{intr.\text{mask}} \) isn’t in \( \text{intr.\text{numa}} \cdot \text{mask} \). Similarly, we use \( \omega_{\text{port-config}} \) to protect invalid affinitization from violating the

B. Operation Complexity

For most applications the capability of hardware hot-plugging is not needed, and, therefore, the expense of gathering and analyzing system information to construct \( L_{\text{packages}} \) and \( L_{\text{interrupts}} \) is amortized and negligible. In \( \text{setup} \cdot \text{intr} \cdot \text{pref} \cdot \text{mask}() \), to find the appropriate \( \text{pref} \cdot \text{mask} \) of each defined port group, grp, we have to find the level, \( l \), of the CPU hierarchy for the condition, \( N_{\text{grp}}(*) \leq \text{hweight}_{\text{intr} \cdot \text{pref} \cdot \text{mask}} \), to hold. The cost of \( \text{hweight}() \) is considered constant time since most CPUs feature built-in instructions for Hamming weight calculation. Thus, the cost of \( \text{setup} \cdot \text{intr} \cdot \text{pref} \cdot \text{mask}() \) is \( O(n^3) \), or \( O(n) \), for \( n \) port groups. Next, the cost of updating workload of one interrupt is constant time since it takes fixed time to calculate the packet count and the formula to compute workload is simple. To find the affinitization mask of one interrupt, \( \text{intr.\text{mask}} \), we have to traverse lists on each level of
the CPU hierarchy. For a system have \( p \) packages, \( d \) cache domains in each package, and \( c \) cores in each cache domain, the cost to find the best core with minimal affinitization cost is \( O(p \times d \times c) \), which for small system can be treated as \( O(1) \). Finally, the activation cost of affinitizing an interrupt is also constant because the time to program the APIC is fixed. As a result, the total cost of each affinitization iteration is \( O(n) + O(m \times p \times d \times c) \) or \( O(m) \) on small multi-core system with \( m \) NICs.

### V. EXPERIMENTAL MEASUREMENTS

In this section, we investigate the performance of our port-configuration assisted affinitization scheme by comparing it to two static and the original irqbalance affinitizers. The experimental environment is a dual-socket industrial PC with two quad-core Intel Xeon E5540 CPUs and 8 GB memory; we run Linux 2.6.32 on this machine. The system architecture is NUMA in nature and cross node accesses introduce large performance overhead. We use eight Intel 82580 NICs and an IXIA chassis to connect them back-to-back for performance assessment. In addition, because each of the E5540 CPU-core is powerful enough to handle a single gigabit NIC, we don’t enable the multi-queue feature of the 82580 NICs.

#### A. Affinitization without Interrupt Balancing

First, to investigate the performance impacts of NIC interrupt affinitization on different packet forwarding applications, we simply don’t activate the dynamic interrupt balancing capacity of the proposed scheme. We employ two typical static affinitization schemes, IP-Aff and CP-Aff, to affinitize NIC interrupts to CPU-cores statically with the former trying to pin interrupts of NICs belonging to the same port group to CPU-cores in a single CPU-package (In-Package), while the latter distributing interrupts to different packages (Cross-Package). Take the platform configured as two firewalls for example, in Fig. 3 and 4, NICs designated as eth-\{2,3,4,5\}-TxRx-0 and eth-\{6,7,8,9\}-TxRx-0 are ports belonging to these two firewalls, respectively. The four-digit hex numbers represent CPU-core numbers and \{01,02,04,08\} and \{10,20,40,80\} are sets of cores in package 0 and 1. As the commands shown in Fig. 3, IP-Aff affinitizes ports of a same firewall group into a CPU-package while, in Fig. 4, CP-Aff does a cross-package affinitization. For the latter two affinitizers, we have irqbalance operate by its default settings and pc_irqb is our port-configuration assisted version of irqbalance.

In each application, we run IXIA’s IXAutomate software several times to generate UDP traffic with varying frame size from 64 to 1518 bytes and present the averaged performance results. We first use the brctl tool and bridge module of Linux to turn the machine into a four-bridge system. The port-configuration of each bridge is detailed in TABLE III. We simulate a LAN environment for each bridge by manipulating the generated frames to use hundreds of different source MAC addresses. In this simple test, we can see that pc_irqb achieves the maximal throughput as IP-Aff by affinitizing the NICs to adjacent CPU-core siblings. Then, to evaluate IP packet routing performance, we enable the Linux’s built-in IP forwarding support and configure the eight NICs to be in two firewalls. By firewalls, we partition the NICs into two groups and configure one NIC in each group to be the “external” interface with a public IP address that is supposed to connect to the WAN in a typical firewall setup. The other three NICs are configured to be the “trusted” or “optional” interfaces with private IP addresses for LAN connectivity. Besides, routing of each firewall operates in a static manner without support for dynamic routing protocol since we focus on packet forwarding here. As we can observe from Table IV, it is beneficial to affinitize NICs in the same firewall to a single NUMA-node. However, the expected deterioration of cross-node affinitization in CN-Aff is not obvious because Linux employs a routing cache to accelerate the process of route lookup. The routing cache is small enough to fit into the CPU L2 cache which avoids the cross-node memory access penalty. Flow tracking is a fundamental mechanism to attribute each individual packet to its associated connection. We use the same firewall setup with Netfilter’s connection tracking module to track each UDP flow passing through each NIC. We can see from Table V that throughput of CN-Aff drops compared to IP-Aff and pc_irqb because the large flow-tracking table can no longer fit into the L2 cache and cross-node memory access overhead emerges. Finally, to perform full packet scanning, we

#### TABLE III. Linux Bridge Performance (Mbps)

<table>
<thead>
<tr>
<th>Bridging</th>
<th>NIC[1,2] → bridge#1; NIC[3,4] → bridge#2</th>
<th>NIC[5,6] → bridge#3; NIC[7,8] → bridge#4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schemes \ Frame size</td>
<td>64 128 256 1518</td>
<td></td>
</tr>
<tr>
<td>IP-Aff</td>
<td>2.286 5.405 7.420 7.896</td>
<td></td>
</tr>
<tr>
<td>CP-Aff</td>
<td>1.524 3.459 7.420 7.896</td>
<td></td>
</tr>
<tr>
<td>irqbalance</td>
<td>1.524 3.836 6.463 7.896</td>
<td></td>
</tr>
<tr>
<td>pc_irqb</td>
<td>2.286 5.297 7.420 7.896</td>
<td></td>
</tr>
</tbody>
</table>

#### TABLE IV. Linux IP Routing Performance (Mbps)

<table>
<thead>
<tr>
<th>Static routing</th>
<th>NIC[1,2,3,4] → firewall#1; NIC[5,6,7,8] → firewall#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schemes \ Frame size</td>
<td>64 128 256 1518</td>
</tr>
<tr>
<td>IP-Aff</td>
<td>3.583 6.774 7.420 7.896</td>
</tr>
<tr>
<td>CP-Aff</td>
<td>3.528 6.723 7.420 7.896</td>
</tr>
<tr>
<td>irqbalance</td>
<td>1.539 3.069 6.210 7.896</td>
</tr>
<tr>
<td>pc_irqb</td>
<td>3.578 6.773 7.420 7.896</td>
</tr>
</tbody>
</table>

#### TABLE V. Linux Flow Tracking Performance (Mbps)

<table>
<thead>
<tr>
<th>Static routing with Netfilter’s conn_track</th>
<th>NIC[1,2,3,4] → firewall#1; NIC[5,6,7,8] → firewall#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schemes \ Frame size</td>
<td>64 128 256 1518</td>
</tr>
<tr>
<td>IP-Aff</td>
<td>2.879 5.730 7.420 7.896</td>
</tr>
<tr>
<td>CP-Aff</td>
<td>2.743 3.459 7.420 7.896</td>
</tr>
<tr>
<td>irqbalance</td>
<td>1.724 3.466 5.725 7.896</td>
</tr>
<tr>
<td>pc_irqb</td>
<td>2.885 5.730 7.420 7.896</td>
</tr>
</tbody>
</table>

#### TABLE VI. Linux Deep Packet Inspection Performance (Mbps)

<table>
<thead>
<tr>
<th>Static routing with Netfilter’s conn_track and string matching</th>
<th>NIC[1,2,3,4] → firewall#1; NIC[5,6,7,8] → firewall#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schemes \ Frame size</td>
<td>64 128 256 1518</td>
</tr>
<tr>
<td>IP-Aff</td>
<td>1.947 3.861 7.420 7.896</td>
</tr>
<tr>
<td>CP-Aff</td>
<td>1.524 3.459 7.420 7.896</td>
</tr>
<tr>
<td>irqbalance</td>
<td>1.161 2.270 4.462 7.896</td>
</tr>
<tr>
<td>pc_irqb</td>
<td>1.929 3.828 7.420 7.896</td>
</tr>
</tbody>
</table>
utilize iptables’ string matching extension with its Netfilter kernel module to match payloads of packets against some predefined pattern. Again, both \textit{pc\_irqb} and \textit{IP-Aff} attain the maximal throughput. From the results present in Table II to V, our port-configuration assisted affinitization scheme performs much better than \textit{irqbalance} that, in the worst case, blindly affinitizes interrupts of NICs in the same port group to CPU-cores in different nodes of a NUMA system. Consequently, \textit{pc\_irqb} achieves the best performance as the optimal static affinitization scheme for the various combinations of packet forwarding applications with different frame sizes.

B. Affinitization with Interrupt Balancing

To demonstrate the interrupt balancing capacity of \textit{pc\_irqb} and \textit{irqbalance} under unbalanced traffic load, we plug one more add-on card with 4 NICs and configure two bridges on the system. We enroll NIC\{1,2,3,4,5,6\} and NIC\{7,8,9,10\} in bridges 1 and 2, respectively. Here, \textit{m8-Aff} performs affinitization in a one-to-one consecutive manner by CPU-core number regardless of the package boundary with a \textit{modulus-4} assignment since we have total eight cores. However, \textit{m4-Aff} confines NIC interrupts of bridges 1 and 2 to CPU-cores of packages 1 and 2, respectively, with a \textit{modulus-4} assignment, i.e., interrupt of NIC\_i is assigned to CPU-core (i \% 4), because each package consists of four cores. To simulate the unbalance traffic conditions, two experiments are conducted: In Test-I, NIC \{1,2\} and NIC \{9,10\} are filled with a large volume of traffic while the other NICs are left idle. On the contrary, in Test-II, we stress all six ports of bridge 1 while pass no traffic to bridge 2. As we can observe from Fig. 5, in Test-I, \textit{m8-Aff} suffers inefficient CPU-core sharing compared to \textit{m4-Aff} since the four ports contend two cores (core 1 and 2) even though other cores are idle. Nevertheless, in Test-II, \textit{m4-Aff} deteriorates the performance by not utilizing CPU-cores on the other package but keeping all the interrupts in a single package. On the other spectrum, although both \textit{irqbalance} and \textit{pc\_irqb} dynamically balance interrupts among busy and lightly-loaded cores, \textit{irqbalance} cannot utilize the knowledge of port-configuration to affinitize interrupts of NICs in a CPU-topology friendly manner. As depicted in Fig. 5, despite a little performance degradation due to the overhead of dynamic interrupt balancing, the throughput of our proposed scheme is close to the best fixed affinitizer for both the simulated tests.

VI. CONCLUSION AND FUTURE WORK

The contribution of this paper is twofold. First, we demonstrate the influence of IRQ affinitization schemes on packet forwarding applications. Second, we factor in the knowledge of port-configuration in attaining an optimal affinitization. Experimental results show improvements of our proposed scheme over the widely-deployed \textit{irqbalance} and typical static affinitization approaches. To better investigate the affinitization problem, our future works include developing a general approach to derive the cost parameters and extending our framework to cover thread-level affinitization and asymmetric multi-processing (AMP) architectures.

REFERENCES

[1] [Online] \textit{irqbalance}, \url{http://irqbalance.org/}
[9] [Online] \textit{ irqbalance}, \url{http:// irqbalance.org/}