Improving Performance in a Combined Router/Server

Voravit Tanyingyong, Markus Hidell, Peter Sjödin
School of Information and Communication Technology
KTH Royal Institute of Technology
Kista, Sweden
Email: {voravit, mahidell, psj}@kth.se

***Abstract*—A modern PC-based router can provide as competitive service as a specialized hardware router while offering more flexibility and possibility to extend beyond routing. We focus on a use case in which the PC-based router also functions as a server. In this paper, we propose a multi-core based architecture for a combined router/server that efficiently provides simultaneous packet forwarding and server processing. We improve the overall performance by creating a fast path for packet forwarding through caching flow entries in on-board classification hardware on the NIC. We propose a generic design based on multi-core processors and multi-queue network interface cards. We describe a prototype implementation and present an experimental evaluation of this design. We also devise a strategy for how to efficiently map packet forwarding and application processing tasks onto the multi-core architecture.**

I. Introduction

Advancements in PC technology enables a modern PC-based router to emerge as a cheap and competitive alternative to a specialized hardware router [1], [2], [3]. Recent research [4], [5], [6] has shown that today's PC-based routers are capable of forwarding several million packets per second, corresponding to up to 10 gigabit/s wire-speed performance, thanks to several improvements in PC architectures:

- Multiple general-purpose multi-core processors tremendously increase computational power.
- Multi-queue network interface cards (NICs) enable parallel processing with a smaller number of cache-misses through CPU affinity mapping of each queue to a CPU core.
- Dedicated memory node for each physical processor allows CPUs to have direct access to memory.
- High-speed interconnection through PCIe offers better system throughput and I/O bandwidth.

PC-based routers offer several advantages over commercial routers. They are normally built using commodity hardware, which is generally cheap and commonly available compared to specialized hardware. Moreover, the forwarding engines of PC-based routers are usually implemented in software on a generic platform making them programmable. These attributes offer more flexibility and foster new forwarding services. For example, researchers can develop and introduce new functionalities using open APIs and standard programming languages, something which is not possible in commercial routers with proprietary interfaces and closed software.

With the generic-purpose nature of a PC-based architecture, it can be extended to offer services beyond solely routing. In this context, we focus on a scenario in which a PC-based router also functions as an application server. One example where such devices are useful is in data centers using e.g. BCube [7], DCell [8] and FiConn [9] interconnection structures. In such scenarios, each server acts as an end host as well as a relay host for other servers. Another example is community-level gateways in residential networks in which a PC-based router can be responsible for packet forwarding as well as for providing local services such as community web portal, mail, media streaming, and directory services.

In this paper, we propose a suitable architecture for a combined router/server that integrates two main functional tasks–packet forwarding and server processing–into a single system. We introduce a simple way to efficiently combine the packet forwarding task and the server processing task as well as to provide the required resources for each task. We utilize parallelism of the multi-core architecture to boost the performance. Moreover, we improve the overall performance by offloading the CPU through a hardware classifier feature on the NIC.

The idea for this work comes as a spin-off from our previous work [10] that aims at improving lookup performance of PC-based OpenFlow [11] using NIC hardware classification to offload the CPU from the lookup processing task. The rest of this paper is organized as follows; Section II describes the architecture for multipurpose PC-based router and motivation behind it. Section III gives an overview of the implementation of our prototype. Section IV covers experiments and evaluations. Finally, we conclude our work and identify potential future work in Section V.

II. Architectural Design

To cope with the increasing demand for services of the future Internet, the underlying architecture of a PC-based router should be flexible without sacrificing performance. We believe that the architecture we proposed in our previous works [10], [12] is an important step in this direction. We intend to extend it further to provide a basis for a multipurpose PC-based router, which we call a *combined router/server*. We adhere to the idea of keeping our design open and accessible by using open source software and standard PC hardware components.

The main challenge with a combined router/server, which integrates packet forwarding and server processing, is how to efficiently combine these two tasks. In this context, we classify the incoming packets into two types: pass-through packets to be forwarded and local-delivery packets to be delivered to applications running on the combined router/server. Our goal is to exploit the multi-core architecture to provide a suitable amount of CPU resources for the packet forwarding task under various conditions.

In Linux, packet forwarding, which is performed in the kernel space, has a higher priority than application-level processing. When the system is under high load and is busy doing intensive packet forwarding, the applications might not get enough computational capacity to perform their task correctly. To address this issue, it has been suggested that some CPU cores should be reserved for application-level processing [4]. This is done by assigning NIC interrupts to a subset of the CPU cores. The kernel scheduler will automatically execute the applications on the remaining CPU cores when the system is under traffic load. We take a different approach to address the problem. We bind all CPU cores to NIC interrupts but we selectively assign a set of the CPU cores to perform packet forwarding. We do this by using on-board hardware classification to direct packets to CPU cores used for packet forwarding. This method allows us to increase the number of the CPU cores used for packet forwarding if there is a need for more forwarding capacity. Not only can we exploit the multi-core architecture to distribute the packet forwarding among the CPU cores for increasing performance, but we can also use it to ensure that there are available resources for server processing.

Apart from efficiently managing CPU resources, we would like to improve the overall throughput of a combined router/server. One way to do this is by offloading packet forwarding to a hardware component. We use commodity hardware to assist the CPUs in processing the pass-through packets.

Modern commodity hardware components are very capable yet relatively cheap making them attractive choices for our purpose. For instance, PacketShader [13] uses Graphics Processing Unit (GPU) acceleration to offload computation and memory-intensive workload. ServerSwitch [14] uses a commodity switching chip to build a customized NIC that can perform various customized packet forwarding in recent proposed data center network (DCN) designs, and leverages the server CPU for control and data plane packet processing. RouteBricks [15] builds a scalable software router using clusters of general-purpose PC hardware that parallelizes router functionality across multiple servers and across multiple cores within a single server. FIBIUM [16], RouteFlow [17], and Flowstream [18] leverage packet forwarding to the switching hardware by consolidating commodity PC hardware and programmable commodity switching hardware. We choose the commodity NIC with hardware classification to assist

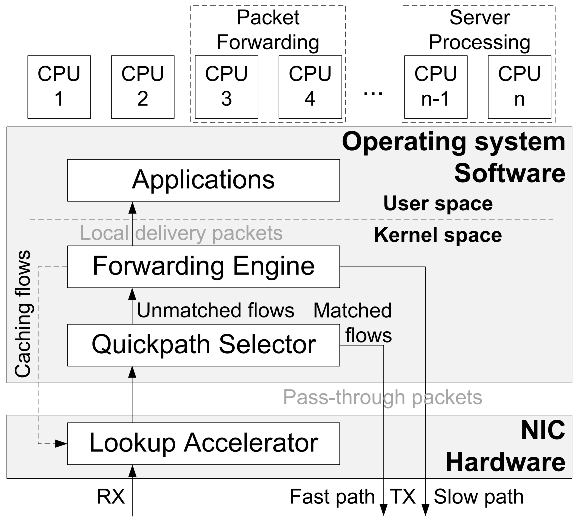

Fig. 1. Generic Architectural Design

the CPUs in processing the pass-through packets. We aim at using unaltered commodity NICs and only make changes in software. This method is easy to adopt and requires no change in the existing infrastructure. With reduced workload, the CPUs will be able to do more packet forwarding and/or server processing.

We propose an architecture as depicted in Fig. 1, where we reserve a set of CPU cores for each functional task. Note that the number of CPU cores reserved as shown in Fig. 1 is just an example, the actual number of reserved CPU cores should be based on the number of available CPU cores and the desired balance between packet forwarding and server processing. The applications in user space are server processes as well as routing protocol processes running on the combined router/server. The forwarding engine is a kernel space networking subsystem that makes forwarding decisions for pass-through packets. To offload the CPU from packet processing, we introduce a fast path in the lookup process to bypass the software based forwarding. This is done by caching active flow table entries in the on-board NIC classification hardware, which functions as a lookup accelerator [10]. In general, commodity NICs have no capability to forward the packet completely by themselves. Thus, the Quickpath Selector is introduced in the kernel as a decision point to determine which path a received packet should take. A pass-through packet belonging to a cached flow in the lookup accelerator will find a match in the Quickpath Selector and gets forwarded without further software processing. A packet that does not belong to a cached flow will be forwarded along the standard software path. A packet destined for a server process on the system will be identified in the forwarding engine and will be passed on to the corresponding process in the user space. The design in Fig. 1 is intended to be generic to allow flexibility and should support any types of hardware classification NICs as well as any types of lookup process in the forwarding engine.

III. Prototype Implementation

We have created a prototype of the combined router/server, where each component in our architecture is implemented as follows:

1) Multi-core system + Multi-queue NIC: Research studies [4], [15] show that the coupling between hardware and software has large impact on the performance of a multi-core PC-based router. Although the packet forwarding task can be distributed among the CPU cores to provide parallelization, this may not always lead to improved performance. For instance, when one receive queue is made available to multiple CPU cores, each core must lock the receive queue before accessing it. This is an expensive operation since the CPU cores have to contend for locking the queue. Furthermore, the study in [4] also shows that an application may suffer from performance problems when executed on a CPU core that is also doing packet forwarding.

In order to efficiently exploit parallelism, multi-queue NICs can be used to reduce locking. [15] suggests two rules: 1) each queue should be accessed by a single CPU core 2) each packet should be handled by a single CPU core. For our prototype, we bind one receive queue and one transmit queue to each CPU core that can be involved in packet forwarding. This is done by mapping receive queue 1 and transmit queue 1 to CPU 1, receive queue 2 and transmit queue 2 to CPU 2, and so on. Doing so would ensure that there is no contention among CPU cores across receive queues and transmit queues and that both rules are enforced. To ensure that applications running on the combined router/server will not suffer when the system is under high packet processing, we limit the packet processing to a specific set of CPU cores.

2) Hardware Classification: We use a NIC with the Intel 82599 10 Gigabit Ethernet (GbE) controller [19] as our lookup accelerator. This Intel chipset offers a hardware classification support called Flow Director filter, which can direct received packets, according to their flows, to queues for classification purposes. We describe the operation of the Flow Director filter in more detail in [10]. We use the Flow Director filter to classify received packets into two types; matched flows and unmatched flows. Matched flows will be forwarded via the fast path while unmatched flows will be either forwarded via the slow path (pass-through packets) or passed on to the applications running on the combined router/server (local-delivery packets).

3) Quickpath Selector: We create a kernel module that implements the Quickpath Selector. We modify the Linux kernel to add a hook to our Quickpath kernel module so that the received packet will be matched against the Quickpath Selector before the general protocol handler. The Quickpath kernel module contains a simple index lookup table with receive interface and receive queue as the lookup key to identify the outgoing interface and outgoing queue.

4) Forwarding Engine: To provide flexible forwarding, we use an OpenFlow implementation as our forwarding engine. Open vSwitch [20], which is a multilayer software switch that supports many features including OpenFlow, matches well with our purpose. We use the Open vSwitch kernel module, which offers high-performance forwarding, for our forwarding engine prototype.

IV. Experimental Evaluation

To evaluate our architecture, we use our experimental prototype as described in section III. We adopt a standard experiment setup in conformance with the RFC 2544 [21] using three PCs to evaluate the performance characteristics of our architecture; PC1 is a traffic generator (source), PC2 is a device under test (DUT), and PC3 is a traffic receiver (sink). Pktgen [22] is used as a packet generator on the source, and a patch for receiver side traffic analysis [23] is used for receiving traffic on the sink. The DUT is configured as a combined router/server with Open vSwitch kernel module. All three PCs have identical hardware; TYAN S7002 Motherboard with Intel Xeon Quad Core 5520, 2.26 GHz, 3GB RAM, one 10 GbE dual-port NIC with Intel 82599 10 GbE controller. The operating system we use is Bifrost/Linux [24] version 7.0 with the Linux net-next 3.0.0 kernel.

To investigate the performance characteristics of our architecture, we carry out various experiments as follows.

A. Baseline performance

This is a test to find the baseline forwarding performance of the Open vSwitch running on our hardware and to investigate how much performance increase we can achieve with Quickpath forwarding in our prototype. We are interested in the maximum throughput as well as the effect from the number of entries in the lookup table.

To keep the test simple, we set up the DUT to use only one CPU core and two receive queues. Both receive queues are mapped to the CPU core. One queue is used for matched flows and the other queue is used for unmatched flows. We carry out a test for a setup with unmodified Open vSwitch and a setup with our architecture. The forwarding rule is simply to forward packets that match entries in the lookup table in each architecture. For a setup with Open vSwitch, we run one round of test to warm the cache so that all possible packet flows generated by our traffic generator exists in the lookup table. All incoming packets will be put in the queue for unmatched flows. This is to make sure that only one queue is used in both setup to provide fair comparison. For our architecture, we preconfigure all possible packet flows generated by the source as 5-tuple (source/destination IP addresses, port numbers, and IP protocol) flow entries in the Flow Director filter, which subsequently redirects them to the queue for matched flows.

We vary the number of the lookup entries between 100 and 5000 entries and measure the performance. The performance metric we focus on is the throughput in terms of packet per second (pps). In each test, the source sends 64-byte packets uniformly distributed from 100 different UDP flows. The DUT receives all traffic on one port and forwards them onto another port to the sink. All generated flows from the source will match entries in the lookup tables in both setups. We vary the load from 1 Kpps up to 1.5 Mpps.

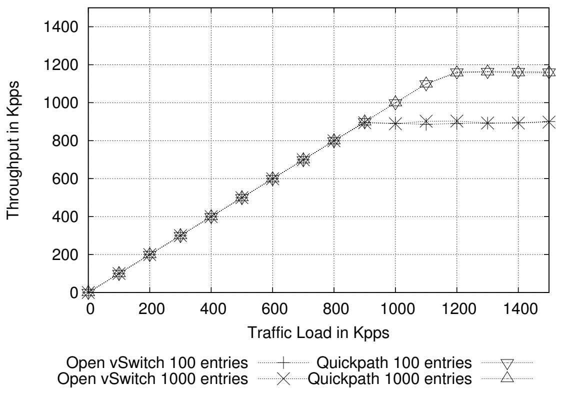

Fig. 2. Baseline Performance Test

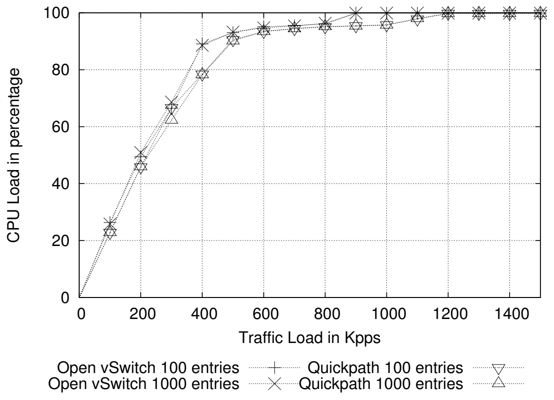

Fig. 4. CPU load

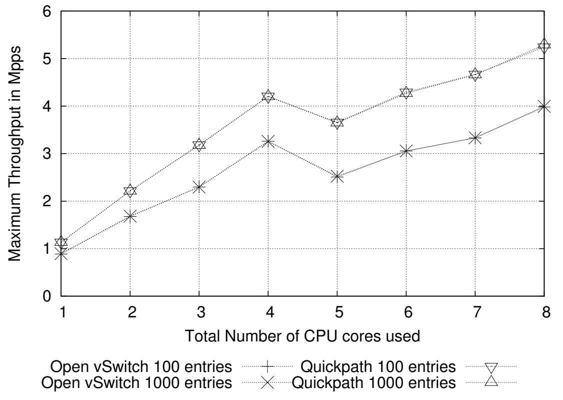

Fig. 3. Performance Scalability in relation to the number of CPU cores

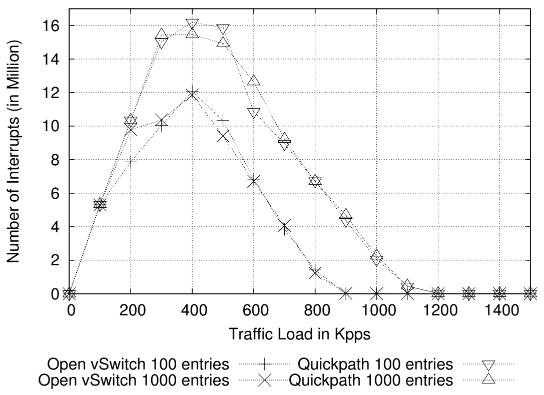

Fig. 5. Number of Interrupts

The results show that for both setups the throughput is independent of the number of entries in the lookup table. For clarity, we plot only the result of when the lookup table has 100 and 1000 entries as shown in Fig. 2. This is because the lookup tables in both setups are based on a hash table. Our architecture increases the forwarding throughput on average by 29% compared to the unmodified Open vSwitch.

For both setups, the overall performance will increase as the number of CPU cores used for forwarding pass-through packets increases as shown in Fig. 3. The performance per CPU core degrades when Hyper-Threading is used (i.e. when using 5-8 CPU cores) due to the contention among logical processors. We have observed the same behavior in previous experiments [10]. For the performance of 1-4 CPU cores, our architecture maintains the average throughput improvement of roughly 32% compared to unmodified Open vSwitch.

B. CPU load and the effect of NAPI

In this experiment, we investigate the CPU load as a function of the traffic served by the CPU. We compare the CPU load of the unmodified Open vSwitch with our Quickpath forwarding. In addition, we examine the effect of the new API (NAPI) [25] in packet forwarding.

We use the same setup with one CPU core as in the baseline performance test, but this time we focus on measuring the CPU load instead of the forwarding throughput. We vary the load from 1 Kpps to 1.5 Mpps. In each test round, we send traffic for 60 seconds. During each round, we take 30 samples of the CPU load, one sample every second, and calculate the average value as the CPU load for each incoming traffic load.

We use cyclesoak[1] to measure the CPU load of the system in this experiment. Cyclesoak is an application to measure system resource utilization through a subtractive method, which measures how much system capacity is available rather than how much is consumed. This gives a more accurate measurement of the kernel subsystems than the conventional process accounting.

We plot the average CPU load versus traffic load as shown in Fig. 4. Each curve has two parts - at low traffic load, the CPU load increases quickly with the traffic load, and at higher load, CPU loads increases only a little with increasing traffic load. Despite the fact that we achieve a significant throughput improvement as shown in the baseline performance test, the CPU load at low traffic load is only reduced by less than 13%. Moreover, the CPU load at higher load is merely a marginal difference. It is clear that we do not reduce the CPU load at the same level as the gain in the throughput. This is expected

[1]source from http://www.tux.org/pub/sites/www.zip.com.au/%257Eakpm/linux/#zc - accessed February 2012

since the lookup itself is not very CPU intensive. However, the Quickpath gives faster per-packet processing (less time required for processing each packet) since the amount of software processing is kept at a minimum.

At higher load, a further increase in the traffic load only gives a small increase in the CPU load. This is due to the effect of using NAPI in the NIC device driver. NAPI stops the NIC from generating interrupt while it handles the incoming packets in the receive queue. With NAPI, a significant performance increase is obtained by switching over from interrupt handling to polling under high traffic load. Once NAPI has processed all the packets in the poll list, it re-enables the interrupt again. As the load increases the system will spend more time processing packets in polling mode relieving a portion of the interrupt handling burden from the CPU core. The CPU core can use the spare cycles to process more packets. More explanation about NAPI is described in [25], [26].

To verify that NAPI does reduce the number of interrupts in our experimental prototype, we repeat the experiment and record the total number of generated interrupts at different traffic load. The result is shown in Fig. 5. We observe the expected behavior of NAPI. By relieving a portion of the interrupt handling from the CPU, it becomes more efficient at processing the packets making it possible to forward more packets with marginal increase in the CPU load. The interrupt rate continues to decrease to the point when the system is saturated. At the saturation point, the incoming packets arrive much faster than the system can process. When the receive queue is full, the NIC will drop incoming packets.

From the results, we can conclude that when NAPI is used it would be better to have just one CPU to forward the packets than to distribute the workload to more CPU cores given that the CPU can forward without dropping any packet. To verify this hypothesis, we carry out an experiment with two CPUs. We investigate how the application processing performance is affected by how the two CPUs are used for packet forwarding.

C. Application processing performance in a combined router/server

This experiment is to verify our hypothesis that we would get better application processing performance when having one CPU core at high traffic load instead of several CPU cores at lower traffic load as long as one core is enough to serve the incoming traffic. We use a setup with two CPU cores to carry out an experiment on two cases; one when a single CPU core is used for forwarding all the pass-through packets and another when both CPU cores are used for forwarding. For the latter case, we distribute the traffic load evenly to both CPU cores. The two cases are illustrated in Fig. 6. We run a server process on each CPU core to simulate services offered by the combined router/server. We vary the traffic load from no load up to 1.5 Mpps. We observe how the traffic load affects the application processing performance and compare the application processing performance of the two cases.

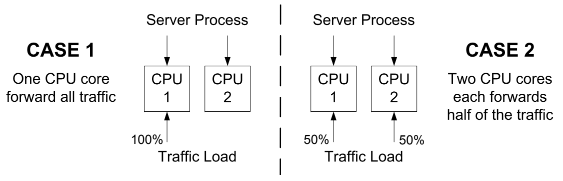

Fig. 6. Test Cases for Distribution of Packet Forwarding Task

We use nbench[2] to simulate the server process in this experiment. Nbench is a Linux/Unix ported version of release 2 of BYTE Magazine's BYTEmark benchmark program. It runs through ten different tasks, each producing a result in terms of number of iterations per second. Nbench uses these numbers to calculate a geometric mean to produce three overall indices: integer index, memory index, and floating-point index. These indices are relative scores compared to a baseline system based on an AMD K6/233 with 32 MB RAM and 512 KB L2-cache running Linux 2.0.32 and using GNU gcc version 2.7.2.3 and libc-5.4.38. To provide a comparable representation of the application processing performance, we normalize each nbench index into a ratio relative to when there is no traffic load (and only nbench occupies the CPU cores) according to (1). N is the normalized index, B_i is the nbench index of each CPU core, Z_i is the nbench index at no traffic load of each CPU core, and i is the CPU core number. We use the normalized indices as our performance metric.

$$N = \frac{\sum_{i=1}^{n} B_i}{\sum_{i=1}^{n} Z_i} \tag{1}$$

After normalizing the three indices, their relative ratios turn out to be almost identical. Thus, we choose to present only the normalized results of the integer index. The result is shown in Fig. 7. From the experiment, all three indices decrease as the traffic load increases for both scenarios. This is expected since the overall application processing performance can be expected to decrease as the CPU cores begin to spend more and more cycles on packet processing. We observe that the application processing performance in case 1, where a single CPU core is used for forwarding all packets, is always higher than in case 2, where both CPU cores share the packet processing burden. The application processing performance stays above 0.5 under high load in case 1 whereas it continues decreasing in case 2. The gap in performance between the two cases is gradually increased as the load increases. This indicates that we can achieve better application processing performance when we allocate a single CPU core to carry out the packet forwarding task, which confirms our hypothesis. Note that our system in case 1 is saturated after 1100 Kpps, since this is the maximum forwarding performance of a single CPU core.

[2]source from http://www.tux.org/%7Emayer/linux/bmark.html - accessed February 2012

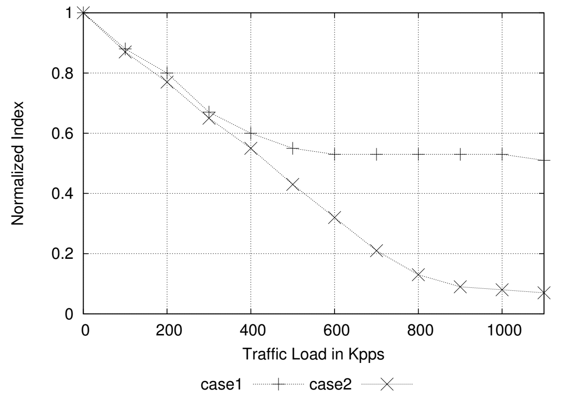

Fig. 7. Application Processing Performance with 2 CPU cores

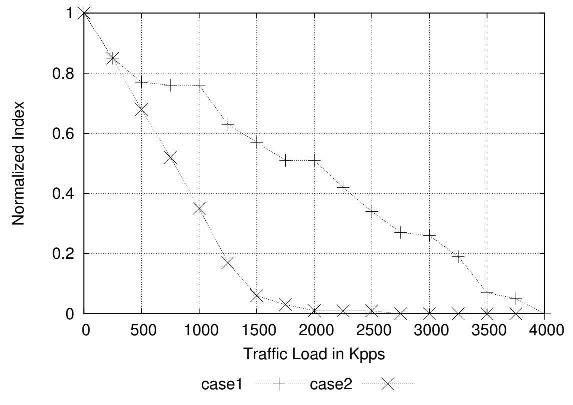

Fig. 8. Application Processing Performance with 4 CPU cores

From the result, it is clear that we should assign one CPU core to process all the incoming pass-through packets as long as the incoming load is less than or equal to what the CPU core can forward without dropping packets. However, it must be possible to invoke an additional CPU core when needed in order to keep up with the forwarding under higher traffic loads. We wish to exploit the fact that the packet forwarding is more efficient when NAPI goes into polling mode (as described in the previous experiment). To do this, we suggest a strategy to invoke an additional CPU core only after the CPU load reaches a certain threshold.

D. Making efficient use of multiple CPU cores

In the previous experiment, we found that it is better to use a single CPU core to forward all packets than to distribute the packets over two CPU cores. However, when all the packets cannot be forwarded by a single CPU core, we need to introduce additional CPU cores to forward the packets. In this experiment, we want to verify that our hypothesis is still valid under multi-core conditions and if the mechanism of adding additional CPU cores to the forwarding task would create any undesired behavior. Thus, we carry out an experiment where we gradually increase the traffic load from zero up to a point where four CPU cores are saturated with packet forwarding. We investigate how the server processing is affected during the increasing traffic load.

We compare two different cases; case 1 in which we start with a single CPU core doing packet forwarding and then gradually invoke additional cores when required due to the traffic load, and case 2 in which the traffic load is always evenly distributed across all four CPU cores. We run nbench on each CPU core to simulate services offered by the combined router/server. We vary the traffic load in each round of test from no load up to 4 Mpps and observe how the traffic load affects the application processing performance. We also compare the application processing performance of the two cases.

The result is shown in Fig. 8. We observe that the application processing performance in case 1 is always better than the application processing performance in case 2. Thus, we confirm that our hypothesis is still valid. We can efficiently increase the forwarding capacity by increasing the number of CPU cores used for forwarding in the combined router/server by adding an additional CPU core to forward the packets when the existing CPU cores are saturated. Despite the fact that we generate the same traffic, we observe that case 2 starts to drop packets after 3750 Kpps whereas case 1 does not drop any packets. The explanation for this can be found in how NAPI polling works with multiple receive queues. Case 1 and case 2 are different in the way flows are mapped to receive queues. In case 1, longer bursts of consecutive packets will be mapped to the same receive queue. Therefore, there is likely to be a larger number of packets to fetch from one receive queue during a polling interval. As a result, a higher load can be served without dropping packets in case 1.

This experiment serves as an example of how good we can achieve with our devised strategy. In reality, we would need some dynamic way to monitor the traffic load and to determine when to invoke an additional core. We would also need to release a core that is no longer needed for packet forwarding, when the traffic load decreases.

V. Conclusions and Future Work

In this paper, we propose an open architecture to enable the PC-based router to support multiple tasks beyond solely routing and forwarding. We outline a generic and flexible architectural design. The design provides a simple mechanism for reserving CPU resources for different functional tasks of a combined router/server. The overall performance is also enhanced by offloading packet forwarding task from CPU cores to the NIC through a solution based on caching of flows and creating a fast path in the lookup process.

We implement a prototype of our architecture using a regular PC with a NIC with the Intel 82599 10 Gigabit Ethernet (GbE) controller, which provides a lookup acceleration function. The classification function is done through a feature called Flow Director filter, which can direct received packets, according to their flows, to queues for classification purposes. We introduce a Quickpath Selector, which uses a simple index lookup table with the receive interface and the receive queue as the lookup key to identify the output port. For flexible forwarding, we use the Open vSwitch implementation of OpenFlow switching.

We present an experimental evaluation to investigate the improved performance of our architecture for a combined router/server as well as to study the relative gain in term of application processing performance we can achieve from careful CPU allocation for packet forwarding. We use a NAPI-enabled driver to efficiently handle high traffic loads and exploit NAPI in our design to map packet forwarding onto specific CPU cores. The results show that the CPU offloading of our architecture increases the throughput by 32% on average compared to unmodified Open vSwitch forwarding. The results also show that for NAPI-enabled systems it is more efficient to have a CPU core forwarding as many packets as possible rather than to distribute packets among multiple cores. However, more research is required to determine whether this result would hold for a system that does not have NAPI-like interrupt management.

Our experiments indicate that careful planning of how to use CPU cores for packet forwarding and server processing significantly affects the overall performance of a combined router/server. We have presented a proof of concept and a future challenge is to introduce a run-time monitoring and dynamic allocation of CPU cores for packet forwarding. In the current prototype, we focus on allocating CPU resources for packet forwarding and assume that the remaining CPU resources will be used for server processing. More research on how best to allocate the available CPU resources for server processes in a combined router/server is needed.

Combining router and server functionalities into a single unit might lead to vulnerabilities. Further studies on isolation and security aspects are required to ensure integrity and accountability of the system and its services. We also plan to investigate the aspect of power efficiency for our PC-based system.

Moreover, in our experiments we cache each flow individually. Although this gives a fine-grained granularity, this might not be realistic since not all flows have good locality. For practical usage, aggregated flows can be used to improve traffic locality. For instance, we can cache flows based on AS level or address blocks (for example /24 subnets) instead of individual source and destination IP addresses. More experiments are needed in order to get insights on how to find a suitable level of granularity.

Finally, the overall performance of the combined router/server could be further improved through the enhancement of the software process. For instance, in most operating system, packets are copied when crossing the kernel/user boundary in the operating system. This is a slow and expensive operation since it involves additional memory allocation/deallocation. Netmap [27] improves this process by introducing a new structure that allows user space applications to directly access the packet buffer. Similarly, Click [28] improves this process through the use of polling and a specialized memory management system that allocates/deallocates packet buffers. Such techniques can be used to enhance the overall performance further.

References

[1] R. Olsson *et al.*, "Open source routing in high-speed production use," in *Proceedings of the 2008 Linux Kongress*, October 2008.

[2] The Tolly Group, "Vyatta 1.1.2, competitive gigabit ethernet lan routing throughput evaluation versus cisco 2821 integrated services router," Test Report, march 2007.

[3] G. Lawton, "Routing faces dramatic changes," *Computer*, vol. 42, no. 9, pp. 15 –17, sept. 2009.

[4] R. Bolla and R. Bruschi, "Pc-based software routers: high performance and application service support," in *PRESTO '08*. ACM, pp. 27–32.

[5] N. Egi *et al.*, "Towards high performance virtual routers on commodity hardware," in *CoNEXT '08*. NY, USA: ACM, 2008, pp. 20:1–20:12.

[6] O. Hagsand, R. Olsson, and B. Gördén, "Towards 10gb/s open-source routing," in *Proceedings of the 2008 Linux Kongress*, October 2008.

[7] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," pp. 63–74, 2009.

[8] C. Guo *et al.*, "Dcell: a scalable and fault-tolerant network structure for data centers," *SIGCOMM CCR*, vol. 38, pp. 75–86, August 2008.

[9] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, S. Lu, and J. Wu, "Scalable and cost-effective interconnection of data-center servers using dual server ports," *IEEE/ACM Trans. Netw.*, vol. 19, pp. 102–114, February 2011.

[10] V. Tanyingyong, M. Hidell, and P. Sjödin, "Using hardware classification to improve pc-based openflow switching," in *High Performance Switching and Routing (HPSR), 2011 IEEE 12th International Conference on*, July 2011, pp. 215 –221.

[11] N. McKeown *et al.*, "Openflow: enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[12] V. Tanyingyong, M. Hidell, and P. Sjödin, "Offloading packet processing in a combined router/server," in *7th Swedish National Computer Networking Workshop SNCNW 2011*, June 2011.

[13] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," *SIGCOMM CCR*, vol. 41, pp. 195–206, August 2010.

[14] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang, "Serverswitch: a programmable and high performance platform for data center networks," in *NSDI'11*.

[15] K. Fall *et al.*, "Routebricks: enabling general purpose network infrastructure," *SIGOPS OSR*, vol. 45, pp. 112–125, February 2011.

[16] N. Sarrar, A. Feldmann, S. Uhlig, R. Sherwood, and X. Huan, "Fibium: Towards hardware accelerated software routers," Deutsche Telekom Laboratories, Tech. Rep., November 2010, techical Report No 9.

[17] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. A. Corrêa, S. C. de Lucena, and M. F. Magalhães, "Virtual routers as a service: the routeflow approach leveraging software-defined networks," in *Proceedings of the 6th International Conference on Future Internet Technologies*, ser. CFI '11. New York, USA: ACM, 2011, pp. 34–37.

[18] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy, "Flow processing and the rise of commodity network hardware," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 20–26, March 2009.

[19] Intel, "Product brief: Intel 82599 10 gigabit ethernet controller," 2009.

[20] B. Pfaff *et al.*, "Extending networking into the virtualization layer," in *Proc. ACM Hotnets-VIII*, New York City, NY. USA., 2009.

[21] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices," IETF, RFC 2544, Mar. 1999.

[22] R. Olsson, "pktgen the linux packet generator," in *Linux Symposium*, vol. 2, 2005, pp. 11–24.

[23] D. Turull, "Open source traffic analyzer," Master's thesis, KTH Information and Communication Technology, 2010.

[24] "Bifrost project." [Online]. Available: http://bifrost.slu.se/

[25] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *Proceedings of the 5th annual Linux Showcase & Conference - Volume 5*. Berkeley, CA, USA: USENIX Association, 2001, pp. 18–18.

[26] K. Salah and A. Qahtan, "Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme," *Computer Communications*, vol. 32, no. 1, pp. 179 – 188, 2009.

[27] L. Rizzo and M. Landi, "netmap: memory mapped access to network devices," in *SIGCOMM '11*. NY, USA: ACM, 2011, pp. 422–423.

[28] E. Kohler *et al.*, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, pp. 263–297, August 2000.