2009 Sixth IFIP International Conference on Network and Parallel Computing

# Improved Forwarding Architecture and Resource Management for Multi-Core Software Routers

Norbert Egi*, Adam Greenhalgh‡, Mark Handley‡, Gianluca Iannaccone†,
Maziar Manesh†, Laurent Mathy*, Sylvia Ratnasamy†

†Intel Research Berkeley, *Lancaster University, ‡University College London

***Abstract*—Recent technological advances in commodity server architectures, with multiple multi-core CPUs, integrated memory controllers, high-speed interconnects and enhanced network interface cards, provide substantial computational capacity and thus an attractive platform for packet forwarding. However, to exploit this available capacity, we need a suitable software platform that allows effective parallel packet processing and resource management. In this paper, we at first introduce an improved forwarding architecture for software routers that enhances parallelism by exploiting hardware classification and multi-queue support, already available in recent commodity network interface cards. After evaluating the original scheduling algorithm of the widely-used Click modular router, we propose solutions for extending this scheduler for improved fairness, throughput and more precise resource management. To illustrate the potential benefits of our proposal, we implement and evaluate a few key elements of our overall design.**

## I. Introduction

Recent advances in server technology promise significant speedups to applications amenable to parallelization. Router workloads appear ideally suited to exploit these advances, which has led to a renewed interest in the applicability of software routers. Software routers offer several advantages, like a familiar programming environment and ease of extensibility, which offers the potential to serve as a single platform onto which one can consolidate many network functions typically implemented by various middleboxes. Examples are the single server-based routers running open-source router software as offered by Vyatta [1], which have even been touted as cheaper alternatives to other commercial routers. However, the limitation of software routers has always been performance. Which is why the possibility of leveraging recent server advances to further the reach of software routers has been of high interest [2] [3] [4] [5].

The recent trajectory of server advances has been of growing the available resource capacity through increased parallelism. Multi-core CPUs increase the available CPU resources; multiple memory controllers integrated in the CPU sockets increase the available memory capacity; multiple I/O links and PCIe buses do the same for I/O and – importantly, high-speed point-to-point interconnects offer high-capacity access between components.

However, precisely because the underlying hardware is parallel, achieving high performance will rely greatly on our ability to distribute packet-processing well across the available resources (not unlike routing across a network – the more alternate routes, the more a routing algorithm has to work to make sure the network capacity is well exploited). Traditionally, this is the job of the scheduler to decide which set of tasks are assigned to which cores and when (if at all) tasks must be moved between cores.

One of the major points of software routers is that they can simultaneously support very different forms of packet processing, with different resource characteristics, on a single hardware architecture. For example, one can imagine having multiple customers – for one we are performing encryption, and for the other deep packet inspection. Since certain operations are more expensive (in terms of the server resources they consume) than others we want to make sure that each customer receives a fair access to the server resources. In general, this requires proper abstractions for flexible and efficient flow differentiation as well as resource allocation. While, we also need scheduling techniques that make good use of the resources so as to achieve high performance, and can also support various scenarios of fair resource sharing and isolation.

In this paper, we observe that neither traditional forwarding architectures nor traditional schedulers are ideally suited to our task, for the following reasons: (1) Traditional forwarding architectures are not capable of taking advantage of recent server advances (e.g. multiple I/O links, hardware classification and multi-queueing in the network interface cards), and thus are limited in exploiting the highest performance achievable. (2) Schedulers are CPU centric. This can be problematic since one of the distinguishing features of a packet-processing workload is that it stresses more than just the CPU. This becomes even more likely as the number of cores increases resulting in bottleneck potentially being any of the CPU, the memory, or the I/O, and hence being "fair" in terms of CPU may not mean an adequate solution. (3) Schedulers fail to take into account the heterogeneity in workloads. (4) Schedulers also fail to consider the basic question of whether we can accurately measure the resource consumption for different packets/flows in a dynamically changing workload. We cannot hope to allocate resources appropriately if we cannot first accurately account for their consumption.

To this aim, we approach the limitations from two directions. On the one hand, we propose an improved forwarding architecture that outperforms, in terms of throughput and latency, the traditional way of packet forwarding, while it also provides additional advantages for resource management and scheduling. We then focus on understanding the limitations of traditional schedulers and on quantifying their impact, thus identifying the requirements for an improved scheduler design.

The paper is organized as follows. In Section II we discuss the limitations of traditional forwarding architectures and propose an architecture that overcomes these limitations. Section III focuses on the resource management. Finally Section IV concludes the paper.

## II. Forwarding Architectures

In this section at first we analyze the shortcomings of the traditional forwarding architecture widely used in commodity software routers and propose an improved forwarding architecture that outperforms, in terms of throughput and latency, this traditional way of packet forwarding, while it also provides additional advantages for resource management. Our

978-0-7695-3837-2/09 $26.00 © 2009 IEEE
DOI 10.1109/NPC.2009.39

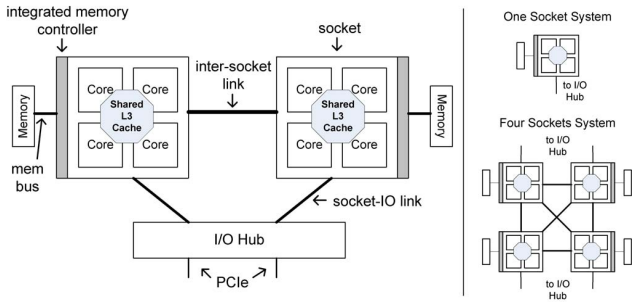


Fig. 1. *Server architectures based on point-to-point inter-socket links and integrated memory controllers.*

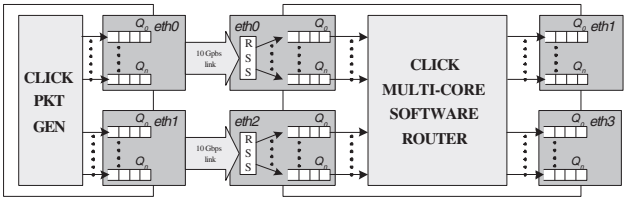


Fig. 2. *Experimental configuration.*

proposed architecture builds on the observation that recently released network interface cards (NICs) [6] provide multiple receive and transmit queues to support server virtualization, but these multi-queue NICs can also be used to achieve better parallelism and hence improve resource management and scheduling in the forwarding engine of software routers.

Before going into the details of the forwarding architectures we briefly overview the server architecture and software environment we used to implement our extensions and carry out our measurements.

### A. Server Architecture

The hardware environment we use for our study is an early prototype of the dual-socket 2.8GHz Intel Nehalem™server [7], since it implements the most recent advances in server architecture. Figure 1 illustrates the high-level architecture of our server. Multiple processing cores[1] are arranged in "sockets"; our server has two sockets with four cores per socket. All cores in a socket share the same 8MB L3 cache, while every core also has on its own a 256KB L2 and a 64KB L1 cache. A memory controller is integrated within each socket and connects to a portion of the overall memory space via a memory bus. The use of multiple sockets with integrated memory controllers means that memory accesses are non-uniform (NUMA). Dedicated high-speed point-to-point links serve to connect the two sockets directly, as well as to connect each socket to the I/O hub. Finally, the I/O hub connects to the NICs via a set of PCIe buses. Our server has 2 PCIe1.1 x8 slots which we populate with 2 NICs, each holding two 10Gbps ports [6]. These network cards can provide us upto 32 transmit (Tx) and 64 receive (Rx) queues, while they support both the Receive Side Scaling (RSS) [8] as well as the Virtual Machine Device queue (VMDq) [9] mechanisms for distributing the arriving packets into the desired Rx queues.

These servers represent the next-generation replacement for the widely deployed Xeon servers, *i.e.*, these servers conform to the informal notion of a "commodity" server.

Our server runs Linux 2.6.24.7 with the Click Modular Router [10] in polling mode—*i.e.*, the CPUs poll for incoming packets rather than being interrupted. We use Click to perform the packet forwarding and processing functions of our router as it has been shown to offer a good tradeoff between ease of programming and performance.

Click is a modular software architecture that offers a flexible approach for implementing software routers on Linux and FreeBSD. We use Click only in the Linux kernel-mode due to the need for the best performance we can get from our architecture. From a high-level perspective the Click implementation consists of packet processing elements and packet queues, connected in a data-flow like configuration. Elements include the code for performing a packet processing function. A number of elements that can be connected in any structure and order needed. Typically, a configuration will have a set of input, forwarding, and output elements, each separated by packet queues.

In addition, Click supports multi-threading inside the Linux kernel [11], that is, different elements in the same configuration can be scheduled on separate cores, providing an excellent platform for developing and experimenting with our forwarding and resource management methods. To further justify our choice of Click over native Linux we refer to the results reported in [12] that show how limited the standard Linux kernel is, as far as its ability of effectively parallelizing the packet forwarding process, and then of exploiting the overall capacity of multi-core systems.

Figure 2 illustrates the topology of our experimental configuration. In our experiments we used two servers, one acted as traffic generator while the other as the actual router. With regard to the generated traffic, we use a synthetic workload of min-sized (64B) packets as it stresses the system the most.

For performance measurements we instrument our server with a proprietary performance tool similar to Intel VTune [13] and somewhat to Oprofile [14].

### B. Traditional Forwarding Architecture

Figure 3 illustrates the packet forwarding configuration used in traditional software routers with single-queued NICs [15]. In this configuration a separate input process ($IN_x$) and output process ($OUT_x$) is needed to move the packets between the Rx and Tx rings and the packet processing part ($PP_x$) of the software. Besides polling the packets in, the input process also classifies the packets and determines to which flows the packets belong to, and places them into the appropriate down-stream queue where the packet waits to be processed by the packet processing and forwarding function(s).

After a packet gets scheduled to be processed by the next "stage" (i.e. $PP_x$), which in the simplest case includes manipulating the TTL and checksum field of the IP header and modifying the link-level header, it gets enqueued into the second down-stream queue where it again waits until all the packets in front of it are removed by the output process, and the output process gets scheduled to dequeue the given packet and move it to the output queue of the card.

The disadvantages of this configuration can easily be spotted: (1) to forward a packet it needs to be processed by three separate software sections, thus it waits three times to be scheduled, resulting in high latency; (2) besides high latency, the throughput is also going to be unavoidably below what might be achieved by the given hardware architecture: on the one hand, because scheduling and context-switching of the

[1]We use the terms "CPU," "core" and "processor" interchangeably.

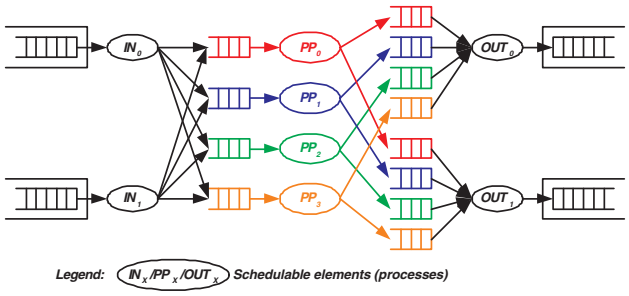


Fig. 3. *Traditional software routers' forwarding architecture*

three different processes (input, packet processing/forwarding, and output) are introducing extra cycles needed to forward a packet, while on the other hand for load-balancing purposes the three different processes handling a packet might be executed on different cores, and as a consequence the packets have to "switch cores" either via the shared last level cache (L3 in our architecture) or even worse, via the main memory; (3) in the case of flow-based resource management, as classification of the packets only happens after a packet has gone through the expensive input operations, and the input process has no control over which flow's packet to poll in next, it is practically impossible to guarantee a flow-based service, while the configuration is also exposed to DoS attacks; (4) resources need to be allocated very carefully to the schedulable processes, in order to avoid under-utilization by allocating resources in an unbalanced way to the processes, resulting in unnecessarily high packet loss. For example, allocating too many resources to the input processes results in a high number of packets which pile up and eventually are dropped from the queues down-stream of the input processes; (5) last but not least, contention over the input and output ports occurs when separate cores want to access the same port at the same time.

### C. Proposed Forwarding Architecture

A packet-processing workload involves moving lots of packets from input to output ports besides also undertaking different packet processing functions required. The question is how should we distribute this workload among the available cores to make optimum use of our multi-core architecture. With this aim, the three main objectives of our proposed architecture are basically: (1) to entirely bind all the operations needed to forward a packet to a single CPU core; (2) to achieve better parallelism in the execution; (3) and to create a configuration that allows us to more accurately measure the cost of forwarding and thus to more easily and accurately distribute the computational load among all the resources in the system. We discuss the first two objectives in more detail in this section, while the third one in Section III.

We propose two broad changes to face the previously described challenges and to overcome the drawbacks of traditional forwarding architectures. First, we extend the Click architecture to leverage the multiple hardware queues available on modern NICs. For this we developed a lock-free device driver for 10Gbps multi-queue NICs and extended the Click modular router with multi-queue support both on the input (PollDevice) and output (ToDevice) side. Our multi-queued Click extension allows us to bind our extended polling and sending elements to a particular HW queue (as opposed to a particular port). As shown in [16], the use of multiple Rx and Tx queues and how the forwarding paths are distributed

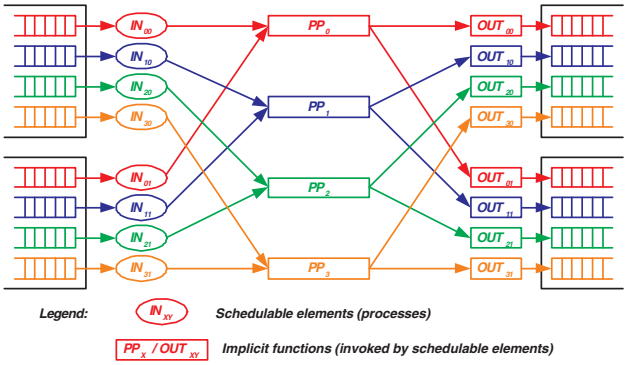


Fig. 4. *Proposed software router forwarding architecture*

across the cores is key to improve both the throughput as well as the latency of our software router, while it also helps to perform more accurate resource management, as it will be demonstrated in the remainder of the paper. The second change is that we extended the ToDevice element so that it can receive packets from an up-stream Click element without the need of an internal packet queue in the forwarding configuration and without the need of scheduling it separately, further simplifying the complexity of the forwarding architecture as well as the scheduling computations (i.e. we extended the ToDevice element with a *push* input and took care that the batching of packets still works in the extended version too).

Figure 4 shows our proposed forwarding architecture we created by using our extended Click elements. In this configuration we have $k$ input and $k$ output queues, where $k$ represents the number of CPU cores in the system (4 in the example in Figure 4), or in a flow-based scenario $k$ might represent the number of separate packet flows the router provides service for. When packets arrive on a network card, some of their Ethernet and/or IP header fields are looked up and a hash function decides into which queue they are placed. In the most widely-used case (i.e. Receive Side Scaling, RSS [8]), the packets are distributed in the available input queues based on a hash function, which function also ensures that packets belonging to the same flow end up in the same input queue, thus avoiding packet reordering within a flow. In addition, packets can also be placed in the queues based on specific values of some header fields, thus providing a flow-based separation of the packets and supporting distinguished (QoS) packet processing (e.g. by using VMDq the packets can be classified by their destination MAC address).

To forward a packet at the front of a given input queue the multi-queue PollDevice element (i.e. the *IN* element in Figure 4) needs to be scheduled at first. As a consequence, this input element polls the packet in from the queue and pushes it down for processing to the subsequent packet processing elements (represented by the $PP_x$ rectangle in Figure 4). After the packet has been processed by these elements, it gets finally pushed to our modified multi-queue ToDevice element, which moves the packet to the associated HW queue's Tx ring, from where the packet is DMAed directly to this hardware queue on the NIC without the interaction of the CPU, before finally transmitted out on the wire. In this configuration we need to only schedule every packet once (via the *IN* element) and after that the packets are forwarded in one step regardless of which output port they are routed to (the arrows among the *IN*, *PP*, and *OUT* elements represent simple function calls). This has a number of advantages that we discuss in parallel with the

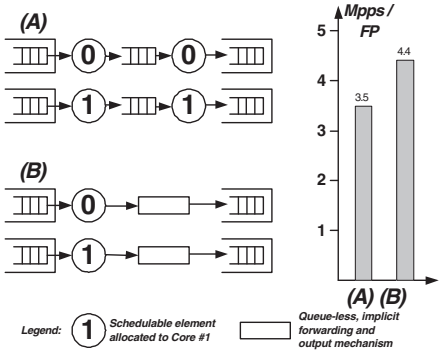


Fig. 5. *2-phase VS 1-phase forwarding mechanism*

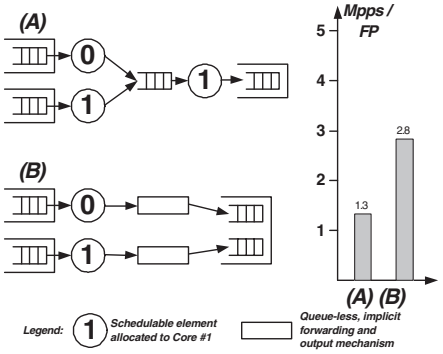


Fig. 6. *Advantage of multiple transmit (Tx) queues*

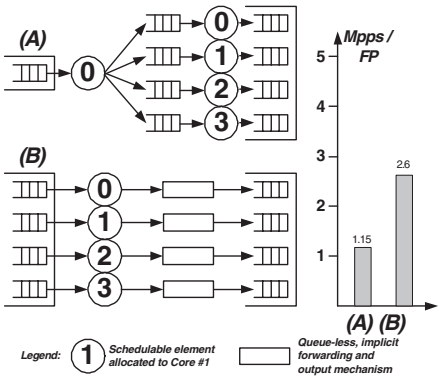


Fig. 7. *Advantage of multiple receive (Rx) queues*

evaluation of the architecture in Section II-D.

### *D. Evaluation*

In order to illustrate the importance and advantages of our forwarding architecture, we created some simple experiments shown in Figures 5, 6, and 7. In all of these experiments we performed only minimal forwarding, that is, traffic arriving at port $i$ is directly forwarded to a pre-determined port $j$ – there is no routing-table lookup nor any other form of packet processing. This minimal forwarding configuration is important, because it only uses the minimal subset of operations that *any* packet-processing application incurs, and because there is no additional processing, the difference in forwarding rates of the different experiments clearly represent the impact of the changes we have applied to the forwarding paths. (Note, that we have labeled every element in Figure 5, 6, and 7 with the identity of the core processing that element.)

In [16], we have demostrated how the input and output processing parts of the forwarding paths should be allocated to cores, and how hardware multi-queueing should be exploited for improved performance and parallelism. For completeness, we summarize and expand on these principles here. We showed that significant performance loss occurs when multiple CPU cores are involved in the forwarding of the packets (e.g. one core is performing the input processing, while the other the output processing), compared to when a single core is doing all the work to forward and process a packet. This performance drop is extremely significant ($\sim$67%) when the different cores are residing on different sockets and are not sharing any CPU caches, causing a memory access everytime the packet is moved from one core to the other. The performance drop is somewhat mitigated ($\sim$33%) when the cores are sharing the same CPU cache, as accessing the cache is much faster than going to the memory. Thus, to achieve high forwarding performance, it is critical to make sure packets stay on the same core all the time while inside the router.

With the experiments in Figure 5 we demonstrate the performance gain that comes from using our 1-phase forwarding mechanism (experiment *B*) as opposed to the traditional 2-phase mechanism (experiment *A*). There are two factors that contribute to the increased performance. On the one hand, we have to schedule every packet only once instead of twice, while on the other, packets do not have to be enqueued into and then dequeued from an internal software queue. Both of these factors also contribute to a lower latency, which unfortunately we are not able to measure accurately due to the lack of an accurate latency measuring device.[2]

The experiment in Figure 5 showed the performance improvement gained by enforcing packets to stay on the same core and by using the proposed 1-phase forwarding mechanism. However, this configuration seem unrealistic in the sense that every forwarding path has only one output port where its packets can leave the router. To overcome this, we have to make sure that every core has parallel access to every output port, which can simply be achieved by using multiple output queues on every port. More precisely, we have to allocate $k$ queues on every output port, where $k$ represents the number of cores in the system. The $k$ queues are needed to avoid contention among different cores accessing the same output port, and to make sure that whichever core has polled a packet in is able to push it out without the need of an internal queue and a second scheduling phase. In Figure 6 we show how important it is to have multiple output queues in order to avoid contention and to be able to keep the packets on the same core for all the time. As the bar-graph shows, the performance improvement is well above 100%.

As just illustrated, having multiple output queues helps to keep packets on the same core and to avoid output port contention, but the input ports are still unshared, which can lead to load-balancing problems. For example, some cores might be under-utilized while others might be overloaded due to unbalanced arrival traffic. In addition, if we have more cores

[2]With Click we were able to measure the average time a packet spends in the internal queue under high load, which is approx. 8600 cycles (3 $\mu$sec), but we could not measure the latency gain of reducing the scheduling iterations from two to one per packet. Measuring latency is subject of future work.

than ports the packet processing has to be pipelined to exploit all the available cores, like in Figure 7(*A*), which can result in core switching and a difficult resource allocation problem. To overcome these problems, we also need multiple input queues. By allocating $k$ queues on every input port we can make sure that every core has parallel access to every input port and thus it significantly simplifies the resource management. Figure 7 demostrates how important multiple input queues are for high-performance packet forwarding.

## III. Resource Management

In the previous section we showed the throughput and latency gains one can achieve with our proposed forwarding architecture. In this section we are going to discuss the other advantages the architecture has to offer from the perspective of resource management.

The objective of proper resource management in software routers is to provide flexible and efficient flow differentiation, resource allocation, and scheduling in order to ensure fair resource sharing, isolation, as well as high performance. In Section II-D, we demonstrated that we can achieve efficient flow differentiation by using hardware classification and multi-queueing available in current NICs. In this section we show how the current Click scheduler and resource allocator might be improved by providing a more predictable system behavior, a guaranteed service (i.e. a guaranteed packet rate and/or bandwidth), higher per-flow and overall throughput and better fairness amongst the forwarding paths competing for the same resources.

### A. Resource management in Click

In Click the base unit of scheduling are the *Task*s. A *Task* is considered to be a chain of elements, which practically compose a part of a forwarding path (FP), that are traversed by a packet within a single scheduling step and it starts with an explicitly schedulable element. These explicitly schedulable elements have a so called *run_task* function that is called upon scheduling, after which the element executes its packet processing function(s) and hands the packet over to the next element downstream in the router configuration. Thus, these other elements are scheduled implicitly, by calling either their *push* or *pull* functions (i.e. they do not have any *run_task* function). A packet is processed and handed over to the next element until it either ends up in a Click queue inside the configuration or in one of the output ports' (or output queues') Tx ring. In these cases the CPU is released and another element within the same thread, with a *run_task* function, at the front of a *Task*, is scheduled. These schedulable elements are responsible for either the input (e.g. PollDevice, FromDevice) or output processing (e.g. ToDevice) or for unqueueing the packets from an internal Click queue (e.g. Unqueue, RatedUnqueue, etc.). Note, that in our proposed forwarding architecture the PollDevice element (i.e. $IN_{XY}$ in Figure 4) is the only schedulable element.

The CPU scheduler currently implemented in Click is based on the Proportional-Share (PS) Stride scheduling algorithm [17]. The implemented scheduler holds the following, mainly positive, characteristics that also apply to most PS schedulers in general:

- Each task reserves a given amount a resources (represented in *Ticket*s in the Stride scheduling algorithm as well as in Click) and it is guaranteed to receive at least this amount of resources when it is not idle. In general PS schedulers, resources reserved for a *Task* ususally mean a given number of CPU cycles-per-second. In current Click the reserved resources mean the number of cycles it takes to execute a *Task*. Hence, because in Click the *Task*s are non-preemptive they can hold the CPU, and thus put load on the other resources (e.g. memory and I/O), as long as they need, regardless of how long it takes to execute them. As a consequence, *Task*s in Click with the same amount of tickets are scheduled equally often, but because of their different resource needs they consume different amounts of the different resources. This way of scheduling results in an unfair resource usage, for which we propose a solution below.
- An idle *Task* cannot "save tickets" to use it when it becomes active.
- For work-conserving purposes, tickets unused by idle *Task*s are distributed among the active *Task*s within the same thread. These active *Task*s are not charged for these extra resources.
- To do a context-switch between *Task*s is an expensive operation, therefore it is recommended to avoid scheduling an idle *Task* if possible. To this aim, Click implements an *adjust_ticket* mechanism, that dynamically changes the actual ticket value of a *Task* between 1 and its allocated ticket value (1024 by default) in correlation with the number of packets processed recently by the *Task* in question.

As far as resource allocation is concerned, Click currently provides two mechanisms for that. On the one hand, it lets the user to statically define which *Task* should be run within which kernel thread. This method is useful for experimentation, but it is very limited when it comes to dynamic resource management in real forwarding scenarios. On the other hand, Click also provides a method where the recent CPU usage of every *Task* is measured, and based on this measurement the *Task*s (if needed) are redistibuted among the available threads for load-balancing purposes.

Although this latter method provides more sophisticated resource management than the static method, it still lacks a number of significant features, such as, it does not take into account the cache hierarchy, it only measures the recent cycle usage of the *Task*s also including in the measurements the cycles used when the *Task* was practically idle and consumed only a few cycles, while it does not maintain any statistics about the real cost of forwarding a packet by any given *Task*. In addition, it does not use any other metric, besides CPU cycles, to describe a *Task*.

### B. Scheduling based on different resources

In previous work [2], [3], [5], we have shown that in multi-core architectures the bottleneck of packet processing can as easily be at the memory or I/O as at the CPUs and hence resource allocation needs to be able to handle the case where any of these is the scarce resource and not the CPU. Therefore, we find it necessary to not only measure the CPU cycles used by the *Task*s, but also other metrics describing the utilization of resources, like memory and I/O bandwidth, cache miss rates at any level of the cache hierarchy, instructions-per-second, etc. and then use these metrics for resource allocation and scheduling calculations. To this end, we use performance monitoring counters to determine the utilization of the resources. These hardware counters are commonplace on modern processors, they are very low overhead counters and were originally introduced for profiling and performance analysis and are therefore suitable for our purposes, that is, to

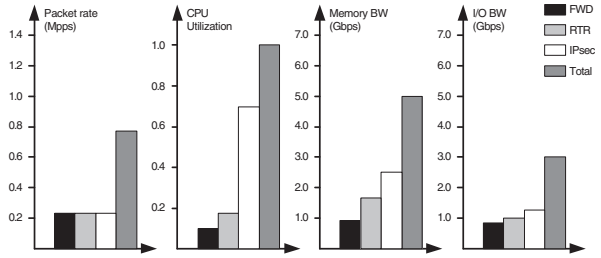


Fig. 8. *Resource usage with the current Click scheduler*

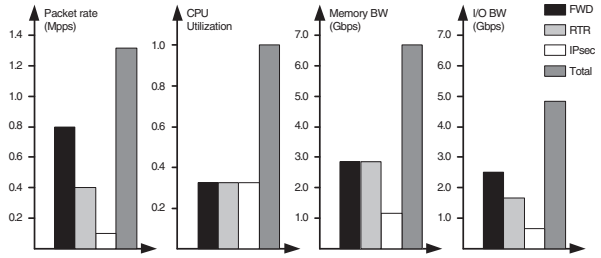


Fig. 9. *Resource usage with fair CPU scheduling*

measure the cost of forwarding a packet on multiple resources. In our hardware architecure, every Hyper-Thread (there are two Hyper-Threads per core when Symmetric Multi-Threading is enabled) possesses 3 fixed and 4 customizable counters, but there are over 500 events these counters can measure, using time-division multiplexing.

One additional great advantage of our proposed forwarding architecture is, that it makes it possible to measure the resource utilization of packet forwarding very accurately and cost effectively. This comes from the fact that every packet is forwarded from the input to the output port in one "go" and on a single core, that is, we have to read and store the values of the counters only once per packet and only on a single CPU. To get the correct resource utilizations we recommend to reset the counters just before a *Task* gets scheduled and read them just after the *Task* finishes running.

To demonstrate the resource usage of different scheduling principles (i.e. default Click, fair CPU, and fair memory scheduling) [3], we at first measured the resource utilization of different workloads. In the second step, we calculated the resource usage of the demonstrated scheduling principles based on the previously measured resource utilizations. The workloads we use for evaluations in the remainder of this section are the followings:

**(1) Minimal forwarding (FWD):** This is the same application that we used in Section II-D, that is, traffic arriving at port *i* is directly forwarded to a pre-determined port *j* – there is no routing-table lookup nor any other form of packet processing.

**(2) IP routing (RTR):** We implement full IP routing including checksum calculations, updating headers and performing a longest-prefix-match lookup of the destination address in an IP routing table. For this latter, we use Click's implementation of the popular D-lookup algorithm [18] and, in keeping with recent reports, a routing table size of 256K entries. For synthetic input traffic, we generate packets with random destination addresses so as to stress cache locality for IP lookup operations.

**(3) IPsec packet encryption (IPsec):** Every packet is encrypted using AES-128 encryption, as is typical in VPNs.

**(4) CRC calculation (CRC):** 32 bit CRC is calculated over the whole packet and appended to the end of it.

Our selection represents commonly-deployed packet-processing applications that are fairly diverse in their computational needs. For example, minimal forwarding stresses memory and I/O; IP routing additionally references large data structures; encryption is CPU-intensive; while CRC is memory-intensive.

[3]Scheduling based on other metrics might be used as well, but due to lack of space we only focus on these now.

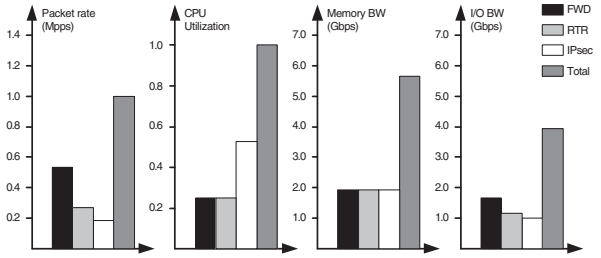


Fig. 10. *Resource usage with fair memory scheduling*

Figure 8 shows the achieved packet rates and resource utilization with 64B packets of three different forwarding paths/workloads (FWD, RTR, and IPsec) co-scheduled on the same CPU by the default Click scheduler. As this default scheduler does not take into account any cost parameters the FPs are considered to be equal and are scheduled equally often, thus resulting in the same throughput. However, as every FP uses different amounts of resources their individual resource consumption are significantly unbalanced and unfair. To overcome this fairness issue, Figures 9 and 10 show the packet rates and resource consumptions in the case of a fair CPU and a fair memory scheduler, respectively. As the graphs show, the packet rate of the cheaper workloads (i.e. FWD and RTR) increases while for IPsec it decreases accordingly to their CPU and memory needs.

As one might notice, all the resources except the one the scheduling is based on are unbalanced. Although, it is possible to devise a scheduling algorithm that equally balances the load over multiple resources, we find it unnecessary. The main reason for this is, that in a real-time scenario there will be always one resource at a given time that is mainly responsible for the contention. Although, this resource might change over time due to the change of the arriving traffic to the FPs, it is simpler and more efficient to equip the scheduler with the ability to schedule the *Task*s based on different resources, but base the scheduling on only one at a time, instead of all of them, including the ones below their saturation point. That is, in an overload scenario if we schedule and fair-share the saturated resource, we ensure that the *saturated* resource is used equally, which might result in unequal usage of the other resources, but as they are still below their saturation point, this unequal resource usage will not cause any performance drop for any of the FPs.

### C. Granularity and Frequency

In Figures 9 and 10 we showed the effect of scheduling the *Task*s based either on their CPU cycle or memory bandwidth consumption. The question "how this might be implemented?" comes naturally. One part is, what we have already described previously, to measure the resource consumption of every *Task*,

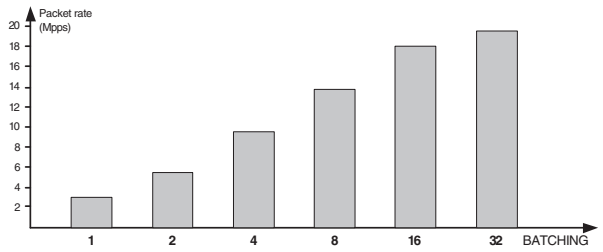


Fig. 11. *The advantage of batch processing in software*

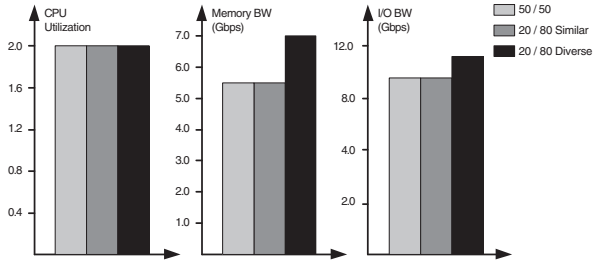


Fig. 12. *Resource utilizations when complying and when not complying with Principle #1*

while the other is to set when the packets should be scheduled. Scheduling a packet happens via the *Task* that is allocated to the hardware input queue the packet resides in. For this latter part we have two tools available, one is the *Ticket* value of the *Task*s while the other is the *BURST* value of the schedulable elements standing at the front of the *Task*s (i.e. PollDevice, or $IN_{XY}$ in Figure 4). In the current implementation both of these parameters are static (i.e. *Ticket*=1024, *BURST*=8), but with some simple extensions it is possible to set them dynamically by the scheduler. The *Ticket* value is the parameter that directly represents how often a *Task* should be scheduled relative to the others. The higher this value is the more often the *Task* is going to be scheduled (in a linear proportion). The *BURST* value is the parameter for batching. That is, when a schedulable element is scheduled it processes as many packets as there are available upto the value of the *BURST* parameter. The advantage of batching is well-know, it reduces the overhead of context-switching, and thus improves the performance.

To demonstrate how significant batching is in Click we ran experiments with minimal forwarding workload and 64B packets with different batching values. Figure 11 shows the performance gain as the function of the increasing batching value. Besides demostrating the performance gain, this set of experiments also helps us to determine the exact cost of context-switches. Using linear-regression we found that it takes approx. 6100 cycles to perform a context switch in Click on our machines, which includes recalculating the virtual time when the currently finished *Task* should be run again, putting the *Task* in the right place of the working queue and fetch the next *Task* from this working queue and run it. The cost of the context-switch is important to know for accurate resource management, as this value has to be included in the cost of forwarding a packet.

However, it is important to note that with batch processing of the packets, we decrease the responsiveness of the system and make the scheduling granularity coarser, which might not be favourable after a certain point. That is, the administrator of the system needs to determine a maximum value, preferably in terms of CPU cycles, for how long a *Task* can keep the resources when it is scheduled, in order to avoid too coarse switching of *Tasks* resulting in poor responsiveness. Let $R$ represent this maximum value. To this end, we recommend to determine the highest burst rate for every *Task* and use that burst rate every time the *Task* gets scheduled. The maximum allowed burst value of every *Task* can be determined according to the following equation:

$$BURST_i = \lfloor \frac{R - C_{CS}}{C_{C_i}} \rfloor;$$

where $C_{CS}$ indicates the cost of a context-switch (i.e. 6100 cycles in our case), and $C_{C_i}$ the CPU cost of $Task_i$.

### D. Co-scheduling principles

Today's general purpose operating systems adapt timeslice-based multiprocessor scheduling to be aware of resource contention and take steps to mitigate it. This means that timeslices need to start at the same time on each core, thus such schedulers require synchronization across all the cores.

As described above, the Click *Task*s cannot be preempted, eliminating the possibility of timeslice-based scheduling. This has the advantage of better resource utilization compared to timeslice-based scheduling. Namely, when in the latter case a process terminates earlier than when its timeslice would have finished, the given CPU is going to be idle for the remaining time of the timeslice. In Click's scheduler, the resources are handed back as soon as a *Task* finishes processing the packets. However, because of the lack of synchronized timeslices across all the cores, it is virtually impossible to have a global scheduler that can accurately co-schedule exact *Task*s running on separate cores, which might lead to unpredictable system behaviour as there is no control over what tasks, with what characteristics are going to run at the same time.

To overcome this unpredictability, we propose to co-schedule *Task*s with the same or similar characteristics on the same core (*Principle #1*), thus ensuring that the given core is always using approximately the same amount of shared resources (i.e. memory, cache, I/O) regardless which *Task* is running at a given time.

To demostrate the importance of this co-scheduling principle we ran three experiments, each with 4 forwarding paths, 2 IPsec and 2 CRC, distributed on 2 cores, each *Task* (FP) being allocated 50% of the CPU resources, emulating a scenario where the CPU is the bottleneck. Figure 12 illustrates the total memory and I/O utilization for the three experiments (as well as the CPU utilization, but because both cores were running at 100% all the time, this has less relevance). The first bars (called "50/50") are the baseline numbers, in which experiment the two IPsec FPs were sharing a core, and the two CRC FPs were sharing the other. The arriving packet rate for every FP was identical and consisted of only 1024B packets. In the second experiment (called "20/80 Similar") the FP-to-core allocation was the same, but on both cores one of the FPs received only 25% of the other FPs packet rate, resulting in a 20 / 80 resource usage due to the proportional-share scheduling mechanism. As the results show, the total memory and I/O utilization has not changed, due to the fact that the *Task*s sharing the same core have the same resource usage characteristics. In the third experiment, we allocated one IPsec and one CRC FP to every core and generated 20% of the traffic on both cores to the IPsec FP, while 80% to the CRC. As the results show, because the CRC workload has different characteristics (higher memory and less CPU needs)

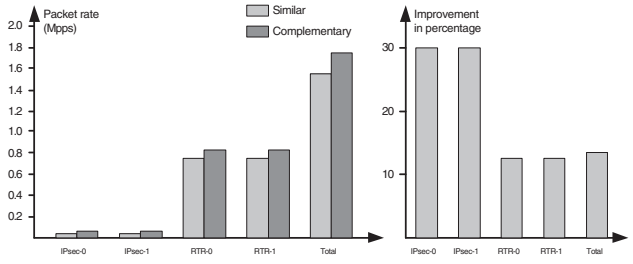


Fig. 13. *Performance improvement when complying with Principle #2*

it used more memory and because of the higher packet rate more I/O bandwidth as well. This phenomenon can be critical in the case when the CPU is the resource saturating at first, but the memory is near its saturation point as well, and when the arriving traffic pattern changes the bottlenecked resource might fluctuate between CPU and memory [4].

Recent Intel processors support the Hyper-Threading (HT) Technology [19] (a.k.a Simultaneous Multi-Threading (SMT)), delivering thread level parallelism for more efficient use of the resources and higher processing throughput. Our server architecture supports two threads per core, thus in total 16 threads. The key behind the improved performance of Hyper-Threading is the ability to fetch instructions from multiple threads in a cycle, thus instructions from more than one thread can be executed in any given pipeline stage at a time. However, to really see improvement in the performance with SMT switched on, we found that carefull resource allocation is needed in the case of a parallelized software router architecture. Our finding is that *Task*s with more diverse (or complementary) characteristics executed on adjacent threads [5] result in more significant performance improvement than *Task*s with similar characteristics (*Principle #2*). Figure 13 shows the results of our experiments on this. In these experiments we had 4 FPs, 2 IPsec and 2 RTR, and 2 cores each with 2 threads, thus 4 threads in total. In the first case (called "Similar") we allocated the identical FPs to the adjacent threads, while in the second case (called "Complementary") one thread on every core was executing an IPsec FP, while the other a RTR FP. The left graph in Figure 13 shows the packet rate for every FP, while the right graph shows the improvement in the packet rate of the *Complementary* experiment over the *Similar* one in percentage. As we can see for the IPsec FPs the improvement is nearly 30%, for the RTR it is nearly 12%, while the total packet rate improvement is slightly above 12%. This improvement comes from the fact that the adjacent threads were executing workloads with different characteristics (i.e. a CPU-intensive IPsec, and a less CPU-intensive, but more memory-intesive RTR), and while their instructions were stressing separate resources the parallelism provided by Hyper-Threading managed to more efficiently utilize these resources (i.e. while the RTR workload was waiting for data to be fetched from the memory the IPsec workload managed to carry on with execution in parallel).

## IV. Conclusion

The performance of modern multi-core commodity architectures clearly indicates its viability for high performance packet forwarding. However, as we demonstrated in this paper, a software router platform has to be designed in a way that provides effective parallelism and proper resource management for workloads with diverse resource characteristics.

Our proposed forwarding architecture enables full parallelization of the resources, while it also improves performance by simplifying the basic structure of the forwarding paths, made possible by hardware multi-queues. Although this architecture is not equipped with the ability to deal with overloaded output ports at the moment, the problem can easily be solved by using an auxiliary queue in the output element that is only used when packets cannot be moved to the output port. These extensions are the subject of our future work together with accurately measuring the latency gain of our architecture.

Our results in Section III clearly indicate that we need to pay attention to the utilization of multiple resources, and not only the CPU, when allocating resources to different workloads. For predictable system behaviour we identified *Principle #1*, recommending to co-schedule workloads with similar characteristics within the same thread, while for improved performance co-schedule workloads with different characteristics on adjacent hyper-threads (*Principle #2*).

Our proposed architecture is capable of performing flow-based packet processing, while it can also be used as a forwarding plane of virtualized software routers, such as introduced in [5].

[4]The saturation point of the resources can be determined by benchmark experiments.

[5]We call the two threads running in parallel on the same core "adjacent threads". Some literature call them "physical" and "logical" threads.

## References


[1] "Vyatta," http://www.vyatta.com/products/.

[2] K. Argyraki, S. A. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedveschi, and S. Ratnasamy, "Can Software Routers Scale?" in *ACM Presto Workshop*, Aug. 2008.

[3] A. Greenhalgh, M. Handley, L. Mathy, N. Egi, M. Hoerdt, and F. Huici, "Fairness Issues in Software Virtual Routers," in *ACM Presto Workshop*, Aug. 2008.

[4] Q. Ye and M. H. MacGregor, "Click on a Cluster: A Viable Approach to Scale Software-based Routers," in *IEEE ICC'07*, 2007.

[5] N. Egi, A. Greenhalgh, M. Handley, L. Mathy, M. Hoerdt, and F. Huici, "Towards High Performance Virtual Routers on Commodity Hardware," in *ACM CoNext*, Dec. 2008.

[6] "Intel 10 Gigabit XF SR Server Adapters," http://www.intel.com/products/server/adapters/10-gbe-xfsr-adapters/10-gbe-xfsr-adapters-overview.htm.

[7] "Next Generation Intel Microarchitecture (Nehalem)," http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf, 2008.

[8] "Receive side scaling on Intel Network Adapters," http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm.

[9] "Virtual Machine Device Queues," http://www.intel.com/technology/platform-technology/virtualization/VMDq_whitepaper.pdf, 2007.

[10] E. Kohler, R. Morris *et al.*, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[11] B. Chen and R. Morris, "Flexible Control of Parallelism in a Multiprocesor PC Router," in *USENIX Technical Conference*, 2001.

[12] R. Bolla and R. Bruschi, "Ip forwarding performance analysis in the presence of control plane functionalities in a pc-based open router," *Springer Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements*, pp. 143–158, 2006.

[13] "VTune Performance Analyzer," http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm.

[14] "OProfile - A System Profiler for Linux," http://oprofile.sourceforge.net/news/.

[15] Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott Karlin, "Scheduling Computations on a Software-Based Router," in *Proceedings of ACM SIGMETRICS 2001*, Cambridge, MA, USA, 2001.

[16] M. Dobrescu and N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proc. of 22nd Symposium on Operating Systems Principles*. Big Sky, MT: ACM, Oct. 2009.

[17] C. A. Waldspurger and W. E. Weihl., "Stride scheduling: deterministic proportional-share resource management." MIT Laboratory for Computer Science, Tech. Rep. MIT/CS/TM-528, June 1995.

[18] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," in *IEEE Infocom*, 1998.

[19] "Intel Hyper-Threading Technology," http://www.intel.com/technology/platform-technology/hyper-threading/.