The Journal of Systems and Software 127 (2017) 12–27

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Profiling and accelerating commodity NFV service chains with SCC

Georgios P. Katsikas*, Gerald Q. Maguire Jr., Dejan Kostić

KTH Royal Institute of Technology, Stockholm, Sweden

ARTICLE INFO

Article history:
Received 3 September 2016
Revised 12 December 2016
Accepted 16 January 2017
Available online 23 January 2017

Keywords:
NFV
Service chains
Profiler
Scheduling
I/O multiplexing

ABSTRACT

Recent approaches to network functions virtualization (NFV) have shown that commodity network stacks and drivers struggle to keep up with increasing hardware speed. Despite this, popular cloud networking services still rely on commodity operating systems (OSs) and device drivers. Taking into account the hardware underlying of commodity servers, we built an NFV profiler that tracks the movement of packets across the system's memory hierarchy by collecting key hardware and OS-level performance counters. Leveraging the profiler's data, our Service Chain Coordinator's (SCC) run-time accelerates user-space NFV service chains, based on commodity drivers. To do so, SCC combines multiplexing of system calls with scheduling strategies, taking time, priority, and processing load into account. By granting longer time quanta to chained network functions (NFs), combined with I/O multiplexing, SCC reduces unnecessary scheduling and I/O overheads, resulting in three-fold latency reduction due to cache and main memory utilization improvements. More importantly, SCC reduces the latency variance of NFV service chains by up to 40x compared to standard FastClick chains by making the average case for an NFV chain to perform as well as the best case. These improvements are possible because of our profiler's accuracy.

1. Introduction

A cost effective means for network operators to increase quality of service (QoS) is through placing network functions (NFs) in the network. These functions provide either basic forwarding and routing capacities, or in the case of middleboxes, enrich the dataplane functionality by offering increased security, policy enforcement, performance improvements, etc. to the overlay services (Carpenter and Brim, 2002). However, to deploy and manage traditional middleboxes requires costly capital and operational expenditures (Sherry et al., 2012). As a result, network operators and cloud providers have shifted their focus towards network functions virtualization (NFV) by migrating middlebox functionality from hardware to software[1] running in commodity, off-the-shelf servers (European Telecommunications Standards Institute, 2012).

Making NFV-style packet processing (i.e., software-based) perform as well as its hardware equivalent (i.e., the hardware implementation of the middlebox device) is hard, mainly because of poor I/O performance. Therefore, researchers tailor the operating systems' (OSs) network stacks and device drivers to achieve line-rate forwarding and maximize throughput (Rizzo, 2012; Kim et al., 2012; DPDK, 2016; Bonelli et al., 2012). That is made possible by (*i*) enabling zero copy data transfers from the network interface (commonly abbreviated as NIC) to user-space (bypassing the kernel), (*ii*) pre-allocating memory resources, and (*iii*) batching packet processing to amortize system call overheads over multiple packets.

Although the aforementioned efforts improve the I/O performance of individual NFs, they diverted researchers' interest from the source of the problem. Indeed, no prior work *analyzed in-depth* the system's state when commodity NFV applications are executing, nor have the exact root causes of the observed performance been *quantified or explained*. NFV service providers could benefit from tools that can thoroughly analyze "hot" parts of NFV software stacks and draw attention to those functions that heavily utilize system resources, hence offer the greatest potential for acceleration. Such tools can also offer run-time support to allow automated tuning of the running NFV.

In response, the first contribution of this paper is our NFV profiler that collects data from low-level performance counters from the underlying NFV infrastructure to track packets as they move

* Corresponding author.
E-mail addresses: katsikas@kth.se (G.P. Katsikas), maguire@kth.se (G.Q. Maguire Jr.), dmk@kth.se (D. Kostić).

[1] The NFV Industry Specification Group of the European Telecommunications Standards Institute (ETSI) has defined NFV-based network functions as "virtual network functions (VNFs)" (ETSI, 2013). In this article we refer to such functions by using the generic definition "network functions (NFs)", that does not necessarily require these functions to be virtualized.

http://dx.doi.org/10.1016/j.jss.2017.01.005

from the NICs to the processors (and vice versa) through the different levels of the system's memory hierarchy. Our profiler decomposes the observed per packet latency into components mapped to the involved hardware components (e.g., caches, main memory) and associates these components with their cause(s) (i.e., the responsible pieces of code that cause this latency).

Today, modern services require combinations of NFs, known as service chains, to satisfy their QoS requirements (Quinn and Nadeau, 2015). For instance, Amazon offers services that allow tenants to build their own virtual infrastructure by combining functions such as filtering, routing, slicing, and load balancing (Amazon, 2016). In such an environment, even state of the art frameworks such as ClickOS (Martins et al., 2014) and NetVM (Hwang et al., 2014) cannot achieve high-performance, as there is a substantial throughput degradation when interconnecting multiple NFs.[2] Recent efforts, such as E2 (Palkar et al., 2015) and OpenNetVM (Zhang et al., 2016), overcome this problem by eliminating hypervisor and paravirtualization overheads via lightweight NFs (e.g., placed in containers) interconnected with fast, custom software switches.

Unfortunately, these latest advancements have not yet been adopted by cloud providers and it is unlikely that this will happen soon, as cloud providers continue to rely on commodity OSs, I/O drivers, and switching fabrics. Although techniques such as single root I/O virtualization (SR-IOV) can bypass the hypervisor and pass packets from the NICs to the virtual machines (VMs) (Amazon, 2016), cloud applications still use costly system calls to interact with the NICs. These interactions are frequent and consume a large fraction of the execution time of an NFV instance.

In the context of chained services, according to our profiler, I/O is not the only problem, as the length of a service chain imposes serious scheduling overheads. Part of this problem has been recently addressed by Sivaraman et al. (2016) with their programmable packet scheduling techniques in switches and by Mittal et al. (2016) introduction of packet scheduling algorithms that roughly meet the requirements of a universal packet scheduler. These approaches can affect the order and timing of packet departures from a queue in a switch or NF, however we suggest a promising alternative direction that is inline with Amazon's attempts to integrate custom schedulers in their cloud services (Amazon, 2016).

In contrast to Sivaraman et al. (2016); Mittal et al. (2016), our research findings show that *a chain of NFs requires a global scheduler to make chain-level decisions*, rather than an internal scheduler that executes local switch policies. To address this gap, we designed and implemented the Service Chain Coordinator (SCC). SCC adjusts the frequency of I/O operations in tandem with adjusting the priority and time quanta allotted to each NF by the scheduler, to maximize the effective run-time of the service chain. In short, we make the following contributions:

1. We introduce an NFV profiler that collects and analyzes low-level performance counters in close collaboration with the hardware and OS. To the best of our knowledge, this is the first NFV profiler; a key tool for uncovering the underlying performance problems of NFV service chains.
2. By exploiting the output of the profiler, our run-time component automatically combines multiplexing of system calls and scheduling re-configurations to accelerate NFV service chains running on Linux OSs.

We implemented SCC on top of the FastClick NFV framework (Barbette et al., 2015). SCC's accelerations realize long chains of user-space NFV service chains, based on commodity device drivers, with 3x lower latency and 3x better cache, and main memory utilization compared to standard FastClick chains. SCC chains also achieve *multiple orders of magnitude lower latency variance* compared to FastClick; a crucial performance indicator for highly-interactive services.

In Section 2, we formulate the research problem and provide a quantitative summary of our contributions.

[2] Figs. 10 and 12 of the ClickOS and NetVM papers respectively.

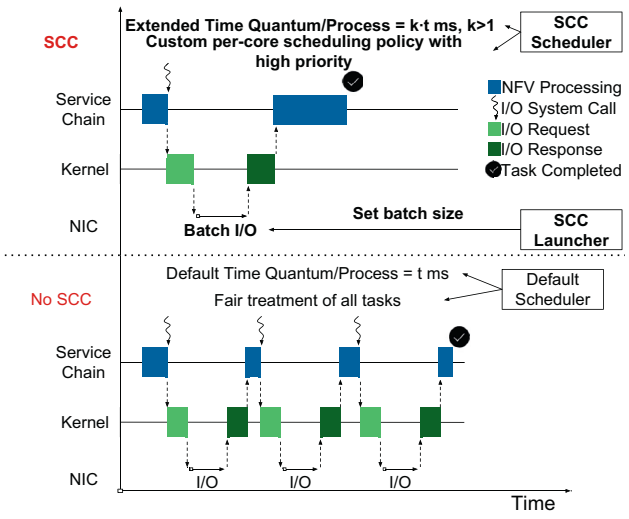


Fig. 1. The SCC run-time combines (*i*) tailored scheduling for NFV service chains via the SCC Scheduler with (*ii*) fewer (but longer) user to/from kernel-space interactions by multiplexing I/O-related system calls via the SCC Launcher. SCC achieves faster completion time, hence lower latency, than the "No-SCC" case.

2. Problem statement

First, we state our research question and the way to address this question.

Key Question: What are the reasons that cause user-space NFV service chains, using commodity OSs and network drivers, to exhibit low performance?

Methodology: In Section 3 we describe an NFV profiler that (*i*) utilizes low-level hardware and software performance counters to track packets as they move across the system's memory hierarchy, (*ii*) measures the per packet latency of the involved hardware components (e.g., caches and main memory), and (*iii*) associates this latency with the cause(s) (i.e., the responsible pieces of code).

We leverage the profiler's power to reveal problems in NFV service chains and quantify their effects (see Section 4). We accelerate NFV service chains by solving those problems identified by the profiler via an automated run-time called SCC (see Section 5).

We illustrate the problems and the solutions realized by SCC in Fig. 1. The bottom part of this figure, labeled as "No SCC", shows a typical way user-space NFV applications based on standard network drivers interact with the NICs via the OS's kernel. As we show in Section 4, this causes two major problems related to the key question stated above:

Problem 1. The service chain at the bottom part of Fig. 1 requires frequent, usually per packet, system calls that cause the service chain to yield the CPU to the OS in order that the latter can perform the necessary I/O operations.

Problem 2. The default Linux scheduler is inappropriate for NFV service chains because it grants short time quanta to the NFV processes and treats them as any other process in the system. As a result, the default Linux scheduler imposes excessive scheduling contention, the latency of which is greater than the actual run-time of a service chain.

Table 1
A summary of our contributions and findings, made in Sections 4 and 6. The evaluation concerns standalone and chained FastClick routers, in different contexts (i.e., user or kernel-space), using different network drivers (i.e., the standard Linux ixgbe and the DPDK drivers), with or without an underlying software switch. The chains are interconnected with either a kernel-based Open vSwitch (OVSK) or back-to-back (B2B).

Comparisons	Findings
Part of the kernel overhead for a single router using the ixgbe network driver compared to the same router using the DPDK network driver.	Locks (27% of the kernel router's time), 10x more context switches because interrupt-handling pre-emptions destroy cache coherency.
User to/from kernel-space time share with respect to the total time spent by a user-space router using the ixgbe network driver.	User-to-kernel for Tx (32.7%), kernel-to-user for Rx (40.5%) of the user-space router's time.
OVSK overhead, comparing a user-space router with and without OVSK, both using ixgbe.	14% overhead due to more function calls, lookup cost and additional trips to user-space.
I/O multiplexing benefits for a user-space router using the ixgbe driver.	3x lower latency and up to 4x lower jitter.
I/O multiplexing benefits for user-space service chains using the ixgbe driver.	Not implemented for chains interconnected with OVSK. 10-40% lower latency and 2x lower jitter for B2B-interconnected chains.
Scheduling benefits for user-space service chains using the ixgbe driver.	30-300% lower latency and up to 40x lower jitter for chains interconnected with OVSK. 10-25% lower latency and 2x lower jitter for B2B-interconnected chains.

These I/O and scheduling problems cause NFV service chains deployed on commodity OSs and network drivers to exhibit high end-to-end latency and latency variance (see Section 4). To solve these problems, we employ SCC, presented in Section 5, (labeled as "SCC" in the top part of Fig. 1) as follows:

Solution to Problem 1: SCC reduces the number of times the path from user to kernel-space and the reverse are used by multiplexing multiple packets into one system call via the SCC Launcher component (see Section 5.1). Our implementation (Katsikas, 2016a) builds upon the popular FastClick NFV framework.

Solution to Problem 2: The SCC Scheduler component realizes a suitable scheduling plan to dramatically reduce the end-to-end latency and latency variance of NFV service chains. To do so, it implements custom single or multi-core scheduling policies for the entire service chain that grant longer time quanta and high priority to the involved processes (see Section 5.2).

We evaluate SCC in Section 6, but we provide a summary of our findings in Table 1.

In the first column of Table 1, we state the comparisons we made throughout this paper among (*i*) standalone NFs that use different network drivers in user or kernel-space and (*ii*) chained user-space NFs, interconnected either with a kernel-based Open vSwitch (OVSK) (Open vSwitch, 2016) software switch or back-to-back (B2B). The second column of Table 1 summarizes our observations, in quantitative terms, made in Sections 4 and 6. We use the term "ixgbe" to refer to the standard Linux network driver for our Intel NICs (see Section 3.1). Note that the first row of Table 1 does not show all of the kernel-space overhead as it was not fully quantified in Section 4.1.

We discuss related and future work in Section 7, and conclude this paper in Section 8.

3. Profiling NFV software stacks

This section introduces the research methodology used for profiling NFV service chains. First, our testbed is described in Section 3.1. The NFV profiler presented in Section 3.2 is used to address two questions:

Question 1: How does the data flow through the hardware of a commodity NFV server?
Question 2: Which elements of a service chain are responsible for the observed latency?

3.1. Testbed

Our testbed consists of two identical machines, each with a dual socket 16-core Intel®Xeon®CPU E5-2667 v3 clocked at 3.20 GHz (Intel, 2016e). The cache sizes are: 2x32 KB L1 (instruction and data caches), 256 KB L2, and 20 MB L3. Hyper-threading is disabled and the OS is the Ubuntu 14.04.3 distribution with Linux kernel v.3.13. The machines are directly interconnected using a dual-port 10 GbE Intel 82599 ES NIC; one machine uses MoonGen (Emmerich et al., 2015) to generate and sink traffic, while in the second machine we deploy our NFV service chains and tools. In the latter machine, we isolated an entire CPU socket to ensure that our measurements will not be affected by other competing processes, while all of the system's other functions use the CPUs in the other socket.

3.2. The SCC Profiler

An NFV profiler must interact with the underlying hardware and OS, to accurately collect and translate relevant events. Although there are tools (Perf, 2016; Levon, 2016; KCachegrind, 2016) for interacting with a Linux OS, one has to employ vendor-specific tools to acquire (some of) the hardware events. Additionally, these tools vary between different hardware architectures from the same vendor. Taking into account these facts, we designed the SCC Profiler; a tool that consists of four modules running atop Intel's Xeon architectures and Linux-based OS as illustrated in Fig. 2. In the remainder of this paper we will limit our discussion to the Linux OS and the Intel Xeon processor used in our testbed. In the following sections, we analyze how the SCC Profiler keeps track of data by establishing bindings with the relevant software and hardware counters of our testbed.

3.2.1. Software monitoring

We use (Perf, 2016) to access the performance counters of the Linux kernel. The SCC Profiler passes the process IDs (PIDs) of the NFV service chain to Perf asking for a variety of events (labeled as "Perf+Linux Kernel" in Fig. 2).[3] By querying the counters of the devices' socket buffers (skbuffs) and network I/O-related system calls, the SCC Profiler learns the number of packets sent/received by the devices and the number of system calls required for these I/O operations.

The Linux scheduler provides counters regarding the execution of each NF of the service chain. The SCC Profiler retrieves the number of CPU migrations and context switches as well as the active, waiting, and blocking times of each NF. As shown in Table 2, using our custom OS benchmarks (available at (Katsikas, 2016c)), we found that the context switching time between two processes scheduled using the default Completely Fair Scheduler (CFS)

[3] When the system's configuration indirectly involves CPU cores that are not used by the PIDs of the NFV processes, the SCC Profiler can be instructed to monitor these additional cores. For example, this might happen if an NF is pinned to a core, but the interrupts of the NICs used by this NF are served by another core.

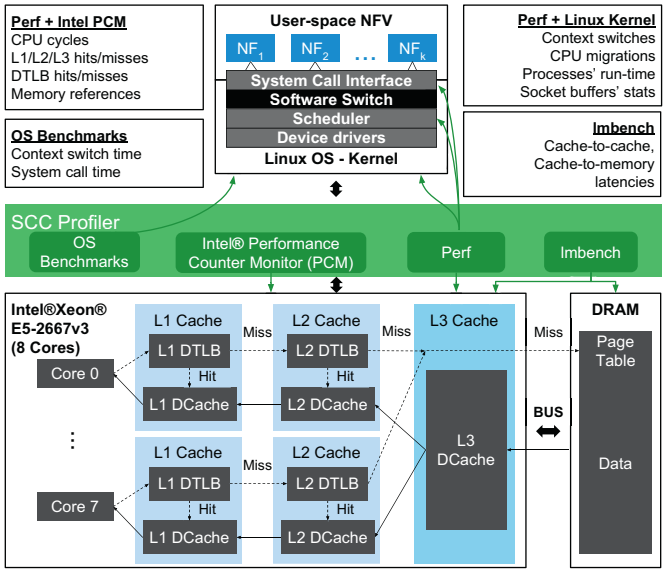


Fig. 2. The SCC Profiler. lmbench measures the latencies to access each part of the memory hierarchy. The SCC Profiler combines the latencies from lmbench with (*i*) the hardware counters obtained by Intel's Performance Counter Monitoring and Perf and (*ii*) the software counters obtained by Perf and our OS benchmarks (all the counters are listed at the top), to measure run-time NFV performance and generate a report of costly operations.

Table 2
Latencies (ns) for a system call and a context switch under different scheduling policies (with default priorities) of the Linux kernel.

OS-level latency source	Latency (ns)
Context switch - Default CFS	1000
Context switch - Batch CFS	1140
Context switch - Real time Round-Robin scheduler	940
Context switch - Real time FIFO scheduler	940
Network I/O System Call	41

(Pabla, 2009) (with the default priority) is roughly 1000 ns, while this time is 940 and 1140 ns when using the real-time First In, First Out (FIFO) & round-robin (RR) and batch scheduling policies respectively. Moreover, the Linux kernel requires 40 ns to execute a network I/O system call (i.e., a read or write to a socket) in our system.

Combining this information with the counters above, the SCC Profiler calculates the latency (per packet) due to the OS when providing basic I/O services (i.e., read and write system calls) to the NFV processes and to coordinate the execution (i.e., schedule) of the service chain. The next target is to capture how the underlying hardware executes the kernel's instructions and how efficiently these instructions pass the packets through the NFV pipeline.

3.2.2. Hardware monitoring

The bottom part of Fig. 2 depicts the elements of one of the CPU sockets and the memory system of the NFV host machine.[4] There are two types of arrows in this part of the figure. The dashed arrows show the flow of the virtual address translation procedure in our processor's memory management unit (MMU). This translation occurs when a program requires a memory access. In this case the CPU passes the virtual address, used by the program, to the MMU asking for a mapping (stored in the OS's page table in the main memory) of this address to physical memory.

Going to memory for translation information before every instruction fetch or explicit data load/store would be prohibitively slow, therefore modern processors employ specialized hardware caches, known as translation lookaside buffers (TLBs), that make a portion of the page table accessible at the speed of the processor, hence speeding up the address translation procedure for addresses with entries in the TLB. Our processor uses a hierarchy of TLBs at the first two (i.e., L1 and L2) cache levels[5], hence a TLB miss will only cause an access to the page table in main memory if neither of the two TLBs contains the mapping. Upon a data TLB (DTLB) hit, the physical page and offset are fetched and the data moves from the respective cache (or the main memory) to the processor following the solid lines in the bottom part of Fig. 2.

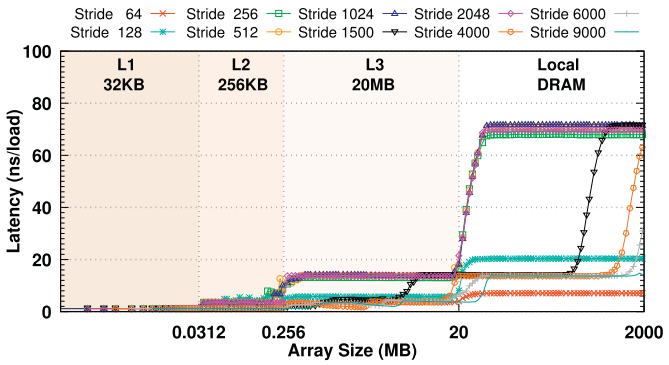


Fig. 3. Latencies to access a progressively increasing array size (1 KB-2 GB) on different parts of the memory hierarchy versus different stride sizes in bytes for an Intel®Xeon®CPU E5-2667 v3 clocked at 3.2 GHz.

Moreover, our processor takes advantage of Intel's Direct Data I/O (Intel, 2016a) (DDIO) technology, allowing the Ethernet controller to use a portion of the processor's last-level cache (LLC) as a primary source and destination of data rather than the main memory, thus achieving lower latency. This portion can be up to 10% of the LLC's capacity, which in our system[6] results in 2 MB (Intel, 2016a). Details about how DDIO works in our testbed are given in Section 6.2.1.2 in Katsikas (2016b).

To understand what delays an application will experience, it is crucial to measure the access costs to all the hardware components of Fig. 2. Specifically our goal is to quantify the latency when data moves across the memory hierarchy, in order to pinpoint the bottlenecks of NFV software stacks. This will allow us to accelerate the NFV implementation, to meet stricter latency requirements.

The SCC Profiler uses lmbench (McVoy and Staelin, 1996) to measure the latencies of all the components of the underlying system's memory hierarchy. Fig. 3 shows these latencies as measured in our testbed. lmbench initiates read and write transactions of progressively increasing array sizes (i.e., 1 KB-2 GB) that can eventually fill all of the caches and part of the main memory, to measure the latency of the following transactions:

Local: from a core to its local L1 and L2 caches,
On-chip: from a core to the shared L3 cache or the L1/L2 cache of another core in the same socket, and
Off-the-chip: from a core to main memory.

Note that the line size of our caches is 64 bytes. This is almost the size of the smallest Ethernet frame. A stride size equal to the cache line's size implies one hit per cache line, hence the latency to access the different parts of the same cache line is low. However, input data in reality might exhibit different access patterns in terms of size, hence we increased the stride sizes, to the size of the standard Ethernet maximum transfer unit (i.e., 1500 bytes), up to the size of a jumbo Ethernet frame (i.e., 9000 bytes) to measure

[4] The L1 instruction cache is omitted for readability reasons and because the miss rate of this cache was negligible in all of our experiments.

[5] The L1 cache also contains an instruction TLB.

[6] In our system, the L3 cache is the LLC, as shown in Fig. 2.

Table 3
Latency calculation formulas and notation for each source of latency in an NFV service chain. The latencies of Table 2 and Fig. 3 are used in the formulas.

Latency/packet		Formula	Notation
Data Access	**L1**	$L1Hits/Pkt \cdot L1DCacheHitLat$	L_{D1}
	L2	$L2Hits/Pkt \cdot L2DCacheHitLat$	L_{D2}
	L3	$L3Hits/Pkt \cdot L3DCacheHitLat$	L_{D3}
	DRAM	$L3Misses/Pkt \cdot MemHitLat$	L_{DM}
Address Transl.	**L1**	$L1DTLBHits/Pkt \cdot L1DTLBHitLat$	L_{T1}
	L2	$L2DTLBHits/Pkt \cdot L2DTLBHitLat$	L_{T2}
	DRAM	$L2DTLBMisses/Pkt \cdot MemHitLat$	L_{TM}
Context Switching		$\sum_{i=1}^{n} ConSw_i/Pkt_i \cdot ConSwLat_p$, where n is the number of processes and p the scheduling policy	L_{CS}
System Calls		$\sum_{i=1}^{n} SysCalls_i/Pkt_i \cdot SysCallLat$, where n is the number of processes	L_{SC}

its latency effect. This way we emulate the size of an skbuff that holds a frame equal to the stride size.

Fig. 3 shows that L1 latencies are not affected by the stride size (with a constant access time 1.18 ns), while L2 latency exhibits low variance with respect to the stride size (between 1.25 and 5 ns). However, comparing the fastest (i.e., 64 bytes) and slowest (i.e., 1024 bytes) stride sizes we see that L3 cache and main memory latencies increase almost 10x (between 1.3 and 14.3 ns for the L3 cache and between 7 and 71.7 ns for main memory), although the smallest (i.e., 64 bytes) and largest (i.e., 9000 bytes) stride sizes exhibit a factor of 140.6x difference in the size. The reason is that the hardware executes prefetch requests, in parallel with the current data processing, to bring cache lines from the next higher level store into the current cache before it is actually needed. In addition, if there is no dependency between the data to be loaded, our CPU can issue multiple instructions to fetch independent chunks of data in parallel. These techniques hide part of the memory access latency, leading to decreased access latency as observed in Fig. 3.[7]

Finally, having quantified the latency to access the hardware components of Fig. 2, the SCC Profiler collects a set of run-time performance monitoring events from the underlying Intel processor. The complete list of available events is available at Intel (2016b) and a detailed explanation of how these counters are collected and combined is provided in the Appendix A.1 in Katsikas (2016b). Specifically, during the execution of an NFV service chain we acquire CPU core, L1, and DRAM events using Perf, while L2 and L3 events are fetched using Intel's PCM tool. To capture the data movements in the bottom part of Fig. 2, we monitor the number of hits and misses of load, store, and prefetch operations for all of the caches (including the DTLBs), and the number of accesses (load and stores) that occur in the DRAM. These events are labeled as "Perf+Intel PCM" at the top left box of Fig. 2.

3.2.3. Latency calculation

The software and hardware monitoring strategies of the previous sections provide enough data to the SCC Profiler for it to project the collected counters on a per packet scale and to calculate the total per packet latency incurred by our NFV server, with respect to the injected load.

Using the primitive latency values from Table 2 and Fig. 3, we compose a variety of latency factors (shown in Table 3). To calculate the total per packet latency, we sum the (*i*) data access and (*ii*) address translation latency factors of each memory level (namely the L1, L2, L3 caches, and the main memory), along with the context switching and system call latencies per packet of each component (i.e., process) of the service chain. The latter factor (i.e., system calls) does not sound as important as the other latency sources; however, in practice, an NFV service chain might be comprised of multiple NFs, usually each NF is deployed as a separate process (e.g., considering a chain as a set of VMs or containers), hence a read and write system call per packet per NF might add up a considerable latency.[8]

Note that according to Fig. 3 the latency of a hit depends on the workload. Under realistic scenarios the incoming traffic will exhibit variable frame sizes, hence all these latencies are possible. For this reason, we instructed the SCC Profiler to use the worst case latency hit for each memory level as per Fig. 3, because (*i*) these cases occur for several input data sizes (they are not corner cases), and (*ii*) their contribution is ten times greater than other input data sizes, hence only a few of these cases might contribute a large latency, the cause of which we do not want to ignore. The number of packets, processed by each NF and in total, are counted by the software monitoring process of the SCC Profiler (see Section 3.2.1). Consequently, using the notation from Table 3, the mathematical formula to compute the total latency per packet is as follows:

$$Latency/Pkt = L_{D1} + L_{D2} + L_{D3} + L_{DM} + L_{T1} + L_{T2} + L_{TM} + L_{CS} + L_{SC}$$

This formula captures the latencies from all the involved hardware components of our system and a portion of the latency added by the OS. In Section 4.2, we analyze a service chain and explain why there are other hidden costs, added by the OS, that are hard to accurately quantify on a per packet scale, although we manage to indirectly reveal their impact. Another important detail regarding the formula above is that when DDIO is utilized by the NICs, frames are exchanged directly with the LLC but this does not prevent a slow NFV system from interacting with main memory as explained in Section 3.2.2. In Section 4 we show that NFV service chains based on unmodified Linux network drivers destroy cache coherency and eventually end up using main memory, as they touch a larger number of memory locations than can stay in the LLC.

Finally we clarify that the SCC Profiler operates in counting mode which means that the counters are aggregated values collected during the execution of an experiment. The SCC Profiler resets all the relevant counters before the experiment and collects their values at the end of the experiment. Hence these values do not involve sampling techniques or other approximation methods. Moreover, since we believe these counters of hardware-based events have high accuracy, we will utilize the values from these counters as much as possible. For example, Perf does not query the memory controller to collect the main memory references, but rather uses a software-based event to estimate this number. Since all the L3 cache data misses in our system end up in main memory, so do the DTLB misses at the L2 cache, we *infer* the number of main memory references by adding up these

[7] Section 6.2.1.2 in Katsikas (2016b) further discusses about the latencies in Fig. 3.

[8] In this paper we do not consider synthesis of a single NF from a chain of NFs as in Katsikas et al. (2016).

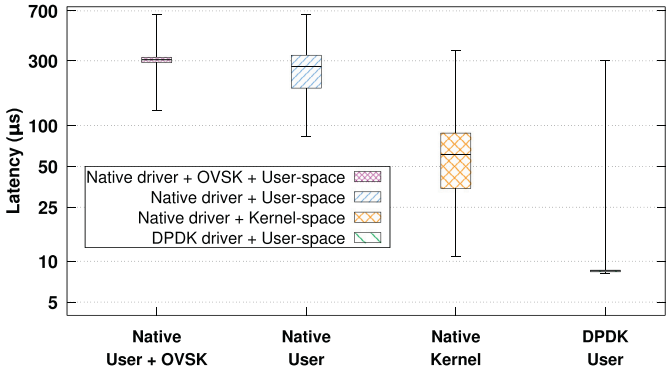


Fig. 4. End-to-end (per packet) latency (µs) plotted on a logarithmic scale for 64-byte frame sizes through four FastClick routers, each running in a different I/O context in a single core as stated in the legend. The input packet rate is 0.82 Mpps.

two hardware-based counters. However, we did not find a substantial difference between Perf's estimates and the values obtained from the hardware counters. As for the ability of the SCC Profiler to time the functions of the NFV stack, we exploit Intel's high-precision event timers (HPET) (Intel, 2016c) via Perf to acquire the entire list of functions together with their contribution to the total latency.

4. Uncovering NFV performance problems with the SCC Profiler

We examine the usability of the SCC Profiler by performing a measurement campaign for both standalone and chained NFs in Sections 4.1 and 4.2 respectively.

4.1. Standalone NFs

We implemented an NF using the fastest Click (Kohler et al., 2000) variant to date, called FastClick (Barbette et al., 2015). We focus on a basic NF, a router, using a slightly modified version of the router implemented by Kohler et al. in Kohler et al. (2000). Our router does not contain ARPQuerier elements, as we assume that the interconnections from this router to other nodes are static, hence we can directly encapsulate (using an EtherEncap element) the IP packet using a *predefined* gateway MAC address as a destination field in the Ethernet frame.

We measured the performance of this router (see Fig. 4) running in four different environments to establish a baseline, in terms of the resource requirements, of our NF. First, we deploy the router natively both in the Linux kernel as well as a user-space application, and tie its ports to the physical 10 GbE interfaces of our NFV server. Then, we measure the same router running as a user-space application in a Linux container. This container is attached to OVSK where it reads/writes frames from/to. Finally, although we target NFV applications that use the native Linux driver for I/O, we also deployed the same router using FastClick's DPDK (DPDK, 2016) I/O elements (using the DPDK network driver) to examine the highest achievable performance. Fig. 4 shows the latency of the router in these four different environments, using a single CPU core. We injected 5 million frames at an input rate of 0.82 million packets per second (Mpps) using a frame size of 64 bytes (without counting the trailing frame cyclic redundancy check (CRC) that we assume will be computed by the NIC itself). We chose a small frame size to impose more work on the CPU core, and hence better stress the different I/O mechanisms. The reason behind the selection of this packet rate is because at this packet rate we easily saturate a 10 Gbps link using a 1500-byte frame; hence, to maintain the same workload on the NF, regardless of the frame sizes we want to test, we use this same packet rate for all the different frame sizes in our experiments (see Section 6). Also, this packet rate conveniently matches the maximum rate that our slowest router (the user-space router, attached to OVSK, using the native network driver) can sustain without dropping packets.

From Fig. 4 we can distill several interesting findings:

Finding 1. The different network drivers clearly affect the performance of the router. A router with DPDK interfaces imposes almost 8x lower median latency than the kernel-space router with the native network driver, despite the fact that in the former case all of the NFs' code is running in user-space.

Finding 2. A user-space router imposes 4x greater median latency than its kernel-space counterpart when both use the same native network driver (i.e., ixgbe).

Finding 3. Attaching the user-space router with the native network driver to OVSK adds ∼10% more median latency compared to the same user-space router without OVSK, with the lower latency percentiles of these two routers exhibiting a larger difference.

The first challenge when exploiting these results from the SCC Profiler is to pinpoint the exact cause(s) of these differences. Running the same experiment under the supervision of the SCC Profiler leads to the results shown in Table 4.

The second column of this table depicts the total per packet latency calculated by the SCC Profiler, following the methodology described in Section 3.2.3. Comparing these numbers with the results of Fig. 4, we can safely state that the SCC Profiler reliably tracks the expended time, as the calculated latency falls within the range of the actual per packet latencies measured by the traffic sink. One interesting observation that arises from this comparison is that the latencies calculated by the SCC Profiler usually fall close to the lower percentiles of the actual latencies (except for the DPDK router where the median and low latency percentiles almost match), showing how "lucky" packets (i.e., those packets that experienced no additional delays) move across the different memories.

To associate the calculated latency with each of the memory levels (i.e., the caches and main memory), we also present the number of memory references per packet and the share (as a percentage) of the total latency spent in each memory level as the third and fourth columns of Table 4 respectively. Note that (*i*) DTLB statistics are not present, but the DTLB cost can be inferred by subtracting 100 from the sum of the percentages of the last column and (*ii*) we omitted the latencies imposed by the number of context switches and I/O-related system calls per packet (as per Table 3) to preserve the readability of the table. The former numbers are nearly zero because only one router is executed by this processor; hence almost no context switches occur. The latter numbers make a negligible contribution to the overall latency as this router executes at most 2 I/O-related system calls (i.e., receive and send) per packet.[9]

4.1.1. Root cause analysis & lessons learned

The results shown in Table 4 were interpreted in detail in Section 6.3.1.1 of the licentiate thesis by Katsikas (2016b). This analysis showed how the four different I/O approaches of the same software router interact with the underlying hardware (via the respective network driver) and which of these approaches exhibits performance bottlenecks.

To summarize the study made in Katsikas (2016b), we quantified the performance difference between a state of the art NFV router using the DPDK network driver and the kernel-based router

[9] In the case of the DPDK router, the memory is mapped to user-space, hence the router does not apply the standard receive/send system calls.

Table 4
The SCC Profiler's latency calculation while tracking the packets injected during the experiment shown in Fig. 4. The second column shows the calculated latency per packet in µs. Columns 3 and 4 show the number of memory references per packet as well as the share (%) of the total latency imposed by each memory level. DTLB impact can be inferred by subtracting 100 from the sum of the percentages of the last column.

	Per packet		
Routers	Latency (µs)	L1/L2/L3/DRAM References	L1/L2/L3/DRAM Latency (%)
User+ OVSK	259.14	3836/182/70/3557	1.71/0.25/0.37/97.06
User	216.97	3653/179/65/2964	1.99/0.29/0.41/96.60
Kernel	25.45	581/23/14/340	2.73/0.31/0.77/95.70
DPDK	8.01	3250/95/163/7	47.35/0.04/27.16/0.06

using the native Linux network driver. We found that locking mechanisms and interrupt handling in the kernel reduce the NFV performance; this is why FastClick adopted DPDK's NIC polling approach, as it requires at most LLC accesses, keeping the caches hot without involving time consuming locks. One could avoid the interrupt costs of the kernel-space router by using the PollDevice Click element. We did not use this element because it works only for a limited set of (old) network drivers.

As for the difference between the user-space and kernel-space routers, when both use the same native driver, we found that the memory allocation and copying between user and kernel-space are the main sources of latency. Despite the fact that user-space Click uses a smart polling mechanism to interact with the NICs, the per packet cost is still high as the polling is not very aggressive. These problems were discussed in earlier work (Rizzo, 2012), but without much evidence; thus one of our contributions is proving this evidence.

To accelerate the kernel-space router, one must employ polling of the NICs and avoid the kernel's locking mechanisms. To achieve better performance for a user-space router, while still using the native network driver, one has to minimize the interactions between the user-space application and the kernel, e.g., applying a system call to an entire batch of frames and to use pre-allocated pools of packet buffers to avoid the cost of dynamic memory allocations.

4.2. Chained NFs

Modern cloud services are comprised of multiple components, often chained together using an underlying switching fabric. Using the NF of the previous section as such a component, we create chains of 1–8 user-space routers, each running in a Linux container on top of a software switch. We chose OVSK (profiled in the previous section) as it is a popular and generic software switch.

We injected the same amount of traffic as in the standalone NF case (see Section 4.1) and pinned all of the routers to one isolated CPU core. We also scheduled OVSK in a different CPU core in the same socket. The boxplots of Fig. 5 show the latency of each chain versus the chain's length, as measured by the traffic sink. The points of Fig. 5, to the right of each boxplot, represent the latency of each chain as measured by the SCC Profiler. Next, we perform a root cause analysis to explain why the latency increases with the chain's length.

4.2.1. Root cause analysis

To visualize the latency of each chain, we fitted the median latencies measured by the traffic sink and the latencies calculated by the SCC Profiler, leading to the equations shown in the legend of Fig. 5. The fitting starts from the chain with 2 NFs. Based on these equations, each additional router in the chain adds 1769 µs of (median) latency, while the SCC Profiler is able to account for roughly 864 µs of this latency, falling between the 1st and 15th percentiles of each chain's latency. Looking at the number of memory

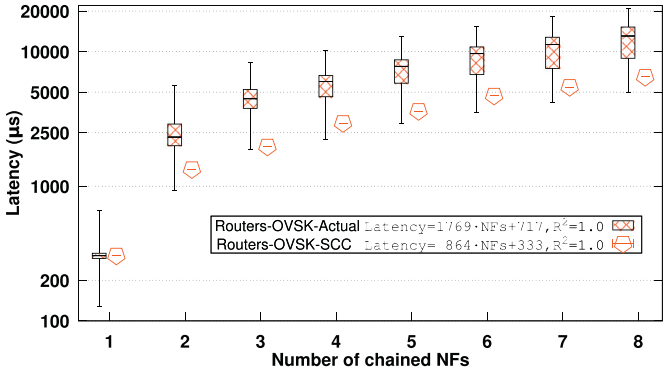


Fig. 5. End-to-end latency (µs), plotted on a logarithmic scale, (*i*) measured at the traffic sink (boxplots) and (*ii*) calculated by the SCC Profiler (points), versus the chain's length for user-space FastClick routers, running in containers on top of OVSK. The routers run in a single core and OVSK is scheduled in a different core in the same socket. The input rate is 0.82 Mpps with 64-byte frames. The linear fit to the median latencies, stated in the legend, begins from the chain with 2 NFs.

references per packet, performed by each chain, the equation that describes their dependence on the chain's length is as follows:

$$MainMemoryReferences/Pkt = 11647 \cdot NFs + 4689, R^2 = 1.0$$

According to Fig. 3, a hit to main memory takes 71.7 ns, hence multiplying this latency with the coefficient from the equation above, results in 835 µs of latency for each additional router; this is almost exactly the latency increase with the chain length, according to the calculation of the SCC Profiler (as shown in the legend of Fig. 5). This is not a surprising result; as in Section 4.1, we showed that even a single user-space router was unable to keep its data in its processor's local cache(s) because the I/O operations involve memory allocation in user-space and data is copied from/to the kernel, touching a lot of memory locations; hence the data cannot stay in the cache causing data exchanges back and forth between main memory and the cache. Clearly this problem only becomes more severe with a chain of these routers.

However, we have not yet clarified, why the latency calculations of the SCC Profiler are below the actual median latencies when we chain NFs. This can be explained by the OS-level counters that the SCC Profiler obtained from each NF. Specifically, by querying the Linux scheduler, the SCC Profiler acquires information about the time a task (*i*) is executing on a CPU, (*ii*) is not runnable, including I/O waiting time, and (*iii*) is runnable but not actually running due to scheduler contention. We derive two metrics from these counters. First, we divide the chain's waiting time by its actual run-time and define the metric "Wait/RunTime". Since the waiting time of each of our NFs is mostly affected by the I/O operations, the "Wait/RunTime" metric captures the impact of yielding the CPU to execute I/O with respect to the effective run-time of an NF. Secondly, we define the metric "SchedContention/RunTime" as a fraction of the time spent due to scheduler contention relative to the

Table 5
Effect of the service chain's length on the (*i*) waiting time and (*ii*) time spent due to scheduling contention with respect to the effective run-time of the service chain.

Chain Length	Wait/RunTime	SchedContention/RunTime
1	14.76	0
2	18.80	1.25
4	30.20	2.88
8	74.36	3.78

chain's run-time. This metric reflects the overhead, added by the OS, to execute the chain.

Table 5 depicts the values of these two metrics for four different chain lengths, as obtained from the experiment shown in Fig. 5. For one router (i.e., with chain length equal to 1), the amount of time spent waiting (mostly for I/O) is almost 15x higher than the time spent executing useful instructions on the CPU, while there is no scheduling overhead since the CPU executes, thus the OS schedules, only this router. Increasing the chain's length leads to increasing waiting and scheduling overheads. The processor spends 75x more time waiting than actually running a chain of 8 routers, while the time that this chain is runnable but does not execute on the processor due to contention in the scheduler is almost 4x higher than the chain's run-time. Our discussion so far has highlighted that both *I/O and scheduling overheads* appear in NFV service chains.

4.2.2. Lessons learned

The overheads shown in Table 5 are captured by the SCC Profiler, but not fully quantified. To clarify this issue, the formula that the SCC Profiler uses to compute the per packet latency (see Section 3.2.3) includes the entire I/O overhead, but *partially* captures the scheduler's overhead by computing the per packet latency imposed by context switching (only a part of the scheduling overhead). This is because it is hard to accurately project this latter overhead to a per packet latency dimension. Despite this, we believe that our latency calculation methodology provides enough accuracy to describe the per packet latency of NFV service chains. Moreover, *the results in Table 5 serve as a motivation for improving the performance of NFV service chains by addressing these I/O and scheduling overheads*. In Section 6.2, we quantify the total overhead of the default Linux scheduler by comparing the performance of NFV service chains scheduled by both the default and a more efficient task scheduler.

The I/O overhead is a well known problem in NFV and has led researchers to abandon commodity network drivers and adopt fast I/O solutions, such as DPDK or netmap. We take a different stance on this problem, by instead providing improved solutions when commodity network drivers are used, as these drivers are utilized by commercial cloud services, such as Amazon's EC2.

The source of the observed scheduling overheads is the default CFS of the Linux kernel. CFS maintains on average equal run-time to all processes, ensuring a fair partitioning of computing resources. This behavior is inappropriate for NFV, as it prevents NFs from running for longer periods of time, leading to more context switches and increased scheduling overhead. An appropriate NFV scheduler should grant enough time to an NF such that, one or more packets can be moved from its input to its output without being interrupted, as this interruption will cause a large loss in the cache coherency.

In the next section we address both I/O and scheduling problems by presenting the run-time part of SCC.

5. The Service Chain Coordinator (SCC)

In this section we utilize the knowledge mined by the SCC Profiler to increase the performance of both standalone and chained NFV applications. We designed SCC to integrate both the profiler and various acceleration techniques into the NFV framework illustrated in Fig. 6. We explain each module of this framework in the following sections.

5.1. The SCC Launcher

A system administrator specifies a service chain using a simple JSON format understood by SCC. This description is injected into two components of SCC: the Launcher and Scheduler. Specifically, the system administrator chooses those NFs that will comprise the service chain (e.g., a firewall followed by a router), the execution environment that will host each NF (i.e., a native or a container-based NF deployment), the I/O driver (to support a kernel or user-space Linux-based chain), the underlying switching fabric (e.g., OVSK, Linux bridges, etc.), the desired topology of the NFs, as well as the hardware components that will be used by the chain (i.e., by selecting the CPU affinity of each service component). These NFs can be selected from a pre-installed library. While in our prototype we used FastClick as a packet processing library, our design is not limited to this library.

Next, the configuration parameters for each NF are passed to the SCC Launcher via a JSON-based configuration file. The SCC Launcher parses this input, composes the chain, creates the necessary interfaces (as needed), and launches all of the components (i.e., switches and NFs). The components are pinned to the requested CPU core(s) according to the CPU affinity mask included in the service chain description.

Once the components are launched, the system administrator can choose, via the configuration file, whether the chain will operate in "profile" or "run-time" mode. In "profile" mode the SCC Launcher passes the service chain's PIDs and configuration to the Profiler. As the configuration specifies the CPU affinity of each NF, the SCC Profiler establishes monitoring connections with the appropriate hardware components. Then, the SCC Profiler operates as described in Section 3.2. In "run-time" mode the PIDs and some auxiliary data structures are passed to the SCC Scheduler. The reason for having these two modes is that the profiling itself occupies system resources, hence we believe that a system administrator would benefit from analyzing her NFV service chains offline, using the SCC Profiler, and then apply the knowledge from the profiling to deploy the accelerated chains online via the SCC run-time.

Before describing the Scheduler, we explain a key internal component of the SCC Launcher, the multiplexing of system calls.

5.1.1. Multiplexing of system calls

The first pillar of SCC's acceleration techniques is I/O optimization for user-space NFs that use native Linux network drivers. This technique is integrated into the SCC Launcher to accelerate interactions of the NFs with the host OS and hardware by multiplexing network I/O-related system calls to reduce the number of times the path from user to kernel-space and the reverse are used.

In Sections 4.1 and 4.2, we showcased that using per packet send/receive system calls to interact with the OS is costly for NFV tasks, especially when a chain of NFs is executed. The reason is that, instead of an NF utilizing its allocated CPU time for performing the actual packet processing, it yields the CPU to the OS in order that the OS can perform the necessary I/O operations each time a packet has to be received or emitted. Moreover, the time spent processing is a small fraction of the time spent for I/O, based on the experiments of Section 4.2.

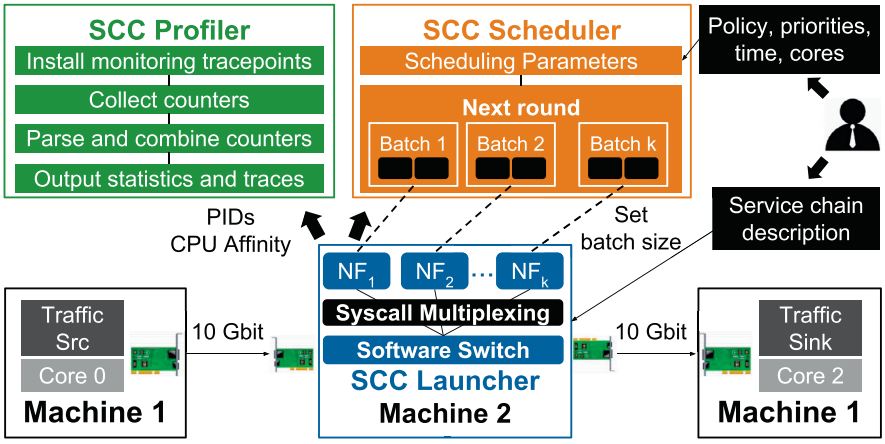


Fig. 6. The Service Chain Coordinator in the context of our testbed. A system administrator inputs a service chain description and configuration (top right). The SCC Launcher identifies the service components, applies the requested configuration and deploys the chain (bottom center). Using the PID and CPU affinity of each NF, the SCC Profiler (top left) can profile the deployed NFs. The SCC Scheduler (top center) ensures that service components comply with the scheduling configuration specified by the system administrator.

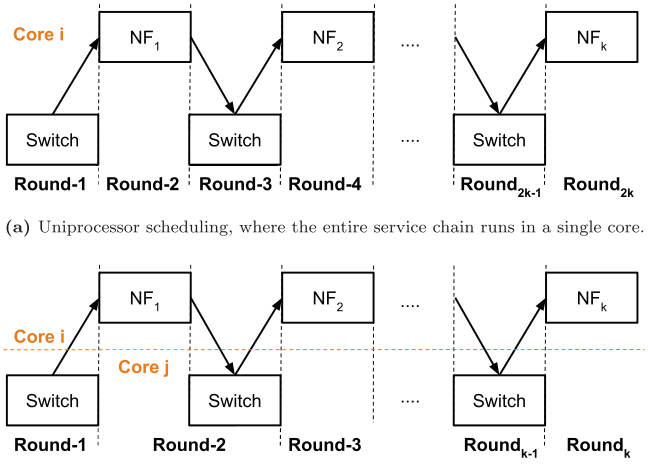
We solve this problem by multiplexing multiple packets into a single system call to allow batches of packets to enter/exit each NF using one receive/send transaction with the kernel. With careful engineering, this method increases the chances of each NF processing an entire batch of packets *uninterrupted*, the next time it gets the CPU. For this to happen, the SCC Launcher operates in three rounds sequentially. In the first round, each NF performs read operations in a batch style. Then, each NF performs its own processing during the second round, while in the last round packets are emitted out of the NFs. In the rest of this paper, we use the term batching interchangeably with multiplexing, both referring to groups of packets being sent/received via one system call.

The system administrator can tune the number of multiplexed system calls via the configuration file shown in Fig. 6. Based on this configuration, the SCC Launcher will *pre-allocate* the I/O vectors that will be used by the kernel to deliver and fetch packets to/from each NF. Ideally, SCC should be able to auto-tune i.e., to select the correct number of I/O vectors itself (based upon changes and feedback from measurements). However, for performance reasons, memory pre-allocation in SCC is static, hence auto-tuning the number of multiplexed system calls, using online feedback from the profiler, would require SCC to restart the NFs. We decided not to automate this process to maintain high performance and prevent service disruptions due to restarting the NFs.

Finally, a challenge when multiplexing I/O-related system calls is to ensure that traffic will not face unacceptable delays when the input rate is low. For example, imagine that one wants to multiplex 16 packets in one, e.g., receive, system call but the input packet rate is e.g., 1 pps. This means that a naive implementation of the multiplexing mechanism might cause the application to block until the entire batch of packets is received (after 16 seconds in this example). To avoid such a problem, the SCC Launcher operates in non-blocking mode, by reading or writing up to a certain number (e.g., 16 packets) of packets at once. If this number is not reached, the system call returns the available packets received/sent or zero if nothing was read/written. This choice allows us to exploit the merits of batching under high input packet rates, while still achieving low latency under low input packet rates.

5.2. The SCC Scheduler

In Section 4.2 we observed increasing scheduling overheads when executing chained NFs. We attributed these overheads to the fact that the default Linux scheduler does not grant large enough time quanta per NF, leading to more frequent scheduling decisions and increased number of context switches. In this section, we allow a system administrator to modify the scheduling procedure of NFV service chains by using our SCC Scheduler.



(a) Uniprocessor scheduling, where the entire service chain runs in a single core.

(b) Example of a multiprocessor scheduling where all the NFs run in one core and the software switch runs in a different core.

Fig. 7. Scheduling options for service chains (NFs and the underlying switch).

The SCC Scheduler boots once the service chain has been deployed by the SCC Launcher, the PIDs of the NFs are available, and the chain operates in "run-time" mode. The SCC Scheduler reads the scheduling parameters from the input configuration, registers the PIDs with the appropriate scheduler based on the requested policy, adjusts the priorities[10], and re-configures time-related scheduling parameters via system calls. We provide more details regarding the reconfiguration of the scheduler in Section 5.2.1.

Depending on the input configuration and the selected data-plane technology that interconnects the NFs, the SCC Scheduler can operate either in single or multi-core mode as illustrated in Fig. 7. If the system administrator wants to deploy both the NFs and the underlying switch in the same core, then the SCC Scheduler executes the service chain as shown in Fig. 7a. This uniprocessor task scheduling scheme invokes the software switch in the odd rounds (i.e., round 1, 3, etc.), and the NFs in the even rounds (i.e., round 2, 4, etc.). The system administrator might allocate a different core

[10] If no priorities are given, default kernel values are used (see Section 5.2.1).

Table 6
Scheduling parameters, useful for NFV tasks, in the Linux kernel version 3.13.

Scheduling Policy		Priority Range		Time Allocation
		Static	Dynamic	
CFS	Default	0	[-19, 19]	Dynamically selected based on (*i*) # of running tasks (*ii*) dynamic priority (see Eq. (1)
	Batch	0	[-19, 19]	
Real Time	RR	[1, 99]	-	Reconfigurable via: sched_rr_timeslice_ms
	FIFO	[1, 99]	-	Time-less scheduler

for the switch, to run the NFs in a dedicated core. In this case, the scheme depicted in Fig. 7b can be used. Modern OSs maintain task queues per core, providing mechanisms to hand off the work from one task scheduler to another, hence better exploiting the hardware capacities. Therefore, to realize the multi-core scheduling plan shown in Fig. 7b, two instances of the SCC Scheduler are required. One instance schedules the NFs and the other schedules the switch, while in each round the NFs and the switch are running in parallel.

If the system administrator wants to allocate more cores for the chained NFs, the latter scenario (and Fig. 7b accordingly) can be generalized in a per core basis manner. This involves running one instance of the SCC Scheduler per core and each instance of the scheduler will coordinate its own processes (i.e., NFs/switch).

A key task of an NFV scheduler is to guarantee that the scheduling plan (e.g., as per Fig. 7) requested by the system administrator will be executed meticulously. This is challenging because modern OSs employ task migration mechanisms to balance the load among all the available cores, when some of the cores are overloaded. Such a mechanism might cause continuous migrations of the chain's processes from one core to another, destroying the cache coherency.

To guarantee that our system realizes the scheduling according to the input CPU affinity and scheduling properties, SCC takes two measures. First, the SCC Scheduler reserves the set of cores requested by the system administrator (see the input of the SCC Scheduler in Fig. 6), by excluding those cores from the candidate list of cores that undertake other (non-NFV) processes in the system, hence ensuring dedicated resources for the NFV processing. Second, as explained in Section 5.1, the SCC Launcher explicitly pins the service chain's processes to the reserved cores based upon the same CPU affinity input. These measures ensure that the reserved CPU cores execute only NFV tasks and the NFV scheduling is solely orchestrated by the SCC Scheduler.

5.2.1. Tuning the Linux schedulers

We studied the task scheduler of the Linux kernel v3.13 to identify knobs that will allow a developer to reconfigure key parameters for NFV service chains, such as the scheduling policy, the priority range of a given scheduling policy, and the time quantum granted to a task by the scheduler. Table 6 summarizes the important properties of these Linux schedulers.

This version of the Linux scheduler maintains 140 queues, each corresponding to a different priority level. Priority levels between 1 and 99 (1 is the highest priority) are static and can be used by processes scheduled by the real-time scheduler. All of the remaining 40 priority levels (i.e., [100, 139]) correspond to a single static priority 0, which is lower than any real-time priority; however, these tasks are mapped to a dynamic priority range in [−19, 19] (with −19 being the maximum dynamic priority) as shown in Table 6.

CFS is the default Linux scheduler that schedules tasks with static priority 0. The core data structure that strikes the balance of all tasks' virtual run-times in CFS is a time-ordered tree, where each node corresponds to a task and is associated with the task's virtual run-time. A task with a low virtual run-time value is stored towards the left side of the tree and has the gravest need for the CPU. Conversely, tasks with a high virtual run-time value (or less need for the CPU) are stored towards the right side of the tree. Therefore, the leftmost node of this tree is the next task to execute on the CPU. CFS exposes system calls to modify a task's dynamic priority. CFS guarantees that the minimum time quantum granted to a task will be always greater or equal than *sched*_min_*granularity* and computes this value based on the following formula:

$$CFSTimeQuantum = \begin{cases} (140 - P) \cdot 20, & \text{if } P < 120 \\ (140 - P) \cdot 5, & \text{if } P \geq 120 \end{cases} \quad (1)$$

where $P \in [100, 139]$ is the task's dynamic priority mapped to a value in [-19, 19] (see Table 6).

Note that, the time that CFS will finally allot to a task also depends upon run-time state variables in the kernel. For example, preemption might be triggered if a more deserving task is available, hence a task's slice might not be entirely consumed. To maintain longer execution times, CFS offers another scheduling policy for CPU-bound processes, called batch CFS. This policy prevents other processes from preempting the CPU as would occur under the default CFS policy, hence the processes run for longer time slices. A process scheduled with the batch scheme "lives" in the same data structure as the processes scheduled by the default CFS scheme, uses the same priority ranges, and the next process to execute is still chosen by CFS. These properties of the batch scheduling scheme are beneficial for NFV tasks as shown in Section 6.2.

The Real-Time Scheduler provides two scheduling policies for interactive tasks: FIFO and RR. Tasks scheduled using either of these two policies will always be prioritized over any tasks scheduled by CFS. The scheduler maintains a list of runnable threads for each possible static priority value. In order to determine which thread runs next, the scheduler looks for the nonempty list with the highest static priority and then selects the task at the head of this list.

A process scheduled by the FIFO scheduling algorithm has no time slice, but instead runs until it blocks (e.g., for I/O), is preempted by a higher-priority real-time task, or voluntarily yields the processor. Two or more FIFO tasks with equal priority do not preempt each other and tasks of lower priority will not be scheduled until the process relinquishes the CPU.

In contrast, the RR real-time scheduling policy is a timeful extension of the FIFO scheme. Unlike FIFO, each task scheduled with the RR algorithm is allowed to run only for a certain maximum time quantum. Upon the expiration of this time quantum, the task will be put at the tail of the queue for its priority. As depicted in Table 6, this scheme exposes a way to adjust the duration of the value of the time quantum via the proc filesystem, hence RR is an alternative scheduling policy for SCC. In contrast, FIFO's timeless approach could be beneficial for executing single-process NFV

tasks, but provides limited control of the process execution time in multi-process NFV scenarios.

5.3. The entire SCC system

Separately employing the I/O and scheduling techniques above might not lead to the desired performance. For example, granting a short time quantum to a process that reads a large batch of packets might not fully reap the benefits of batching. In contrast, allocating a long time quantum for a process that applies per packet read/write operations cannot be fully exploited since, sooner or later, the process will yield the processor to perform I/O, thus "losing" the opportunity to exploit the long time quantum.

For these reasons, as shown in Fig. 6, SCC builds a run-time that effectively combines these accelerations. As explained above, the system administrator can tune the number of multiplexed system calls, scheduling policy, scheduling priority and the time quantum of each process to achieve fast packet processing. As illustrated in Fig. 1, correct selection of these parameters will allow a CPU to process an entire batch of packets per scheduling round, leading to fewer user to/from kernel-space paths being used, hence lower latency. We evaluate the effectiveness of SCC in Section 6.

6. Performance evaluation

This section evaluates the acceleration techniques of SCC. We used the standalone and chained NFV services, profiled and analyzed in Sections 4.1 and 4.2 respectively, to assess the benefits of SCC. Using the experimental setup described in Section 3.1, we deploy these NFV service chains on top of SCC (see Section 5) and evaluate: (*i*) The effect of SCC's I/O multiplexing on the performance of individual user-space NFs (see Section 6.1). (*ii*) The impact of different scheduling strategies on the performance of chained user-space NFs (see Section 6.2). (*iii*) The benefits of SCC when both I/O multiplexing and scheduling are applied (see Section 6.2.2).

6.1. Impact of SCC's I/O multiplexing

The goal of this section is to evaluate our first acceleration technique for user-space NFV chains: multiplexing multiple packets into one system call. We use the user-space FastClick router (based on the native network driver) from Section 4.1 and deploy it on SCC. Then, we assess the impact of I/O multiplexing (as a function of the batch size) on the router's performance, by conducting a sensitivity analysis using an exponentially increasing batch size based on the formula: $batch\_size = 2^i\big|_{i=0}^{8}$. When the batch size equals 1, no batching is used, i.e., simply the standard FastClick I/O. We take this as the base of what we want to accelerate.

Fig. 8 depicts the latency of a single router as a function of the batch size with four different frame sizes (i.e., 64, 128, 256, and 1500 bytes). We highlighted two areas in this figure: the leftmost area, with a light red background, where batching is disabled, whereas the remaining area, with a light green background, shows the router's performance for different batch sizes. We input frames with different frame sizes at the same rate (i.e., 0.82 Mpps, which is the line-rate for the 1500 byte frame size) used in all of our experiments. As we see in Fig. 8, the load imposed on the router is the same for all the frame sizes, since the router exhibits similar latencies, independent of the frame size. For this reason, we used the SCC Profiler to analyze the memory utilization of the router during one of these experiments, with the smallest frame size (i.e., 64 bytes), as shown in Table 7. This frame size was also utilized to profile the same router (see Section 4.1) without batching, hence it offers a clear comparison reference for our I/O acceleration.

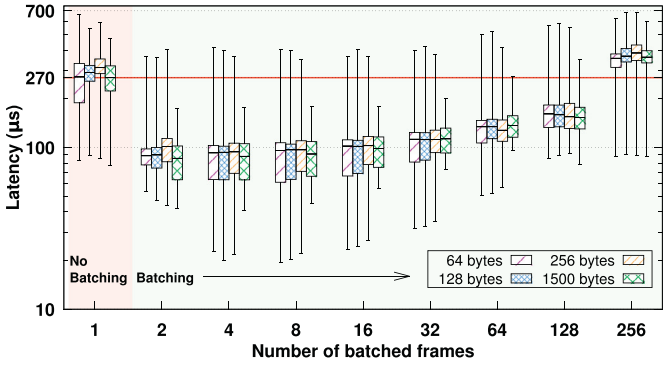


Fig. 8. End-to-end latency (µs), plotted on a logarithmic scale, versus the number of frames multiplexed/batched into one system call for a user-space FastClick router using the native Linux network driver. The router runs in a single core and the input packet rate is 0.82 Mpps with 64, 128, 256, and 1500 byte frames. The corresponding bit rates are 0.57, 0.99, 1.84, and 10 Gbps.

Table 7
The SCC Profiler's per packet latency calculation and memory utilization report while tracking the *64-byte* packets injected during the experiment shown in Fig. 8. The latency in the second column is calculated using the collected performance counters introduced in Section 3.2.3 and falls within the actual latency percentiles shown in Fig. 8.

	Per packet	
Batch size	Latency (µs)	L1/L2/L3/DRAM References
1	216.97	3653/179/65/2964
2	126.94	2048/138/49/1731
4	117.47	1872/125/46/1603
8	101.05	1601/115/42/1378
16	95.23	1503/119/41/1298
32	84.20	1315/110/38/1148
64	85.88	1331/113/40/1170
128	94.31	1471/145/46/1284
256	155.0	2564/252/81/2109

Looking at the results of Fig. 8, we see that our batching acceleration clearly outperforms the non-batching case, achieving 2-3x lower median latency for several batch sizes. Specifically, the best batch sizes, with respect to the end-to-end latency, are between 2 and 32 batched system calls, as the median latency for almost all frame sizes is in the range of 80–115 µs, whereas the non-batching cases achieve median latencies in the range of 270–310 µs. The lowest value of the latter group of medians is also visualized with the red horizontal dashed line shown in Fig. 8. This difference in latency is reflected in a decrease in the number of references to memory with batch sizes greater than 1, as shown in Table 7. Batching decreases the number of main memory references by 2-3x. As main memory accesses are the main component of the latency, we can see the effect of batching is very beneficial in reducing latency (up to some point). More notably, the worst case latency (here the 99th percentile) for some batch sizes *is comparable or even lower (i.e., for 1500-byte frames) than the median latency of the router without batching.*

Another benefit of batching is the reduction of the latency variance (also known as jitter). Without batching, the router exhibits a latency variance of 400–600 µs for the different frame sizes. When using a batch size of e.g., 2 system calls, this variance is roughly 300 µs (i.e., 33–200% lower) for the 64, 128, and 256 byte frames and only ~130–140 µs (i.e., 2.5-4x lower) for 1500 byte frames. We believe that jitter-sensitive NFV applications will find this batching beneficial.

Batch sizes between 4 and 32 have some outliers at very low latency, comparable to the levels of a DPDK router (see Fig. 4). Moreover all the latency percentiles for these batch sizes are shifted by

roughly 3x compared to the non-batching case. Further increasing the batch size (i.e., batch sizes of 64 and 128 frames) achieves yet lower median latency than the non-batching case. However, using a batch size of 256 frames increases the latency and latency variance as both metrics are greater than the non-batching case. This is not surprising, since aggressive batching has a well-studied effect on latency and latency variance (Kim et al., 2012).

From a resource utilization perspective, Table 7 shows the router's per packet latency and different types of memory accesses, as computed by the SCC Profiler, for the different batch sizes when the frame size is 64 bytes. As stated earlier, a clear impact of system calls' multiplexing is the reduction in main memory accesses. In the non-batching case, almost 3000 main memory references per 64-byte frame occur, resulting in a latency of 216.97 µs, as calculated by the SCC Profiler. Exponentially increasing the batch size from 2 to 256, leads the OS to transferring more data per batch (i.e., an entire batch of frames is transferred with one system call) and this transfer occurs less frequently (because we apply one system call every "batch size" number of frames). Consequently, batching exploits the spatial locality of virtual memory addresses; this means that multiple virtual addresses tend to fall into the same physical page, hence there are fewer references to main memory. This effect is shown in the 3rd column of Table 7. For batch sizes between 2 and 64, the router makes 2–2.8x fewer main memory references than the router without batching, hence the latencies shown in Fig. 8 are greatly affected by this phenomenon.

This analysis leads to three conclusions: (*i*) To benefit from I/O multiplexing, moderate batch sizes between 2 and 32 system calls decrease the end-to-end latency and latency variance for small, medium, and large frames; hence this degree of multiplexing appears attractive. (*ii*) When the goal of an NFV provider is to fit more service chains into a given hardware capacity, choosing bigger batch sizes (i.e., between 8 and 64 system calls) leads to more than 2x better cache utilization (especially by reducing the number of main memory accesses). (*iii*) For jitter sensitive applications, batching 2 system calls gives the best results both from the latency and jitter perspectives.

6.2. Impact of SCC's scheduling

In this section we evaluate the effects of different scheduling strategies on the performance of NFV service chains. In Section 4.2.1, we observed increasing scheduling overheads with the length of the chain and attributed these overheads to the inability of CFS to grant large enough time quanta per NF. Here, we deploy NFV chains using SCC and utilize the SCC Scheduler to, ideally, eliminate this overhead.

Considering the analysis of the different schedulers in Section 5.2, we see that the batch CFS and the real-time RR schedulers offer interesting properties that could be beneficial for NFV service chains. Here, we chose to evaluate one of them against CFS, due to space limitations. We selected batch CFS, which implicitly allows us to modify the time quantum of each NF in a chain by setting its "niceness". Based on Eq. 1, without modifying the "nice" value, a process can run for up to 100 ms; we have increased this value to study its effect on the performance of the NFV chains.

In Section 6.2.1 we evaluate a multi-core scheduling scenario without using our I/O multiplexing acceleration, while in Section 6.2.2 we combine scheduling with I/O multiplexing in a single-core scenario.

6.2.1. Multi-core scheduling without I/O multiplexing

Fig. 9 shows the latency of 1–8 user-space FastClick routers, chained together, on top of an OVSK instance. The chains are scheduled using the multi-processor scheduling option of SCC shown in Fig. 7b, where one CPU core executes the OVSK, while another CPU core in the same socket executes the NFs. Specifically, the top set of chains in the legend is scheduled by the default CFS (with the default time quantum up to 100 ms), while the other two sets of chains are scheduled by the batch CFS with two different time quanta configurations. The former configuration's time quantum is roughly 4x greater than the default (i.e., 420 ms), while the latter is 8x greater than the default (i.e., 800 ms). To achieve this configuration we set the "nice" value to −1 and −19 respectively (or 119 and 100 based on Eq. (1). Although CFS does not guarantee to exhaust its assigned slice, it acts as an upper bound.

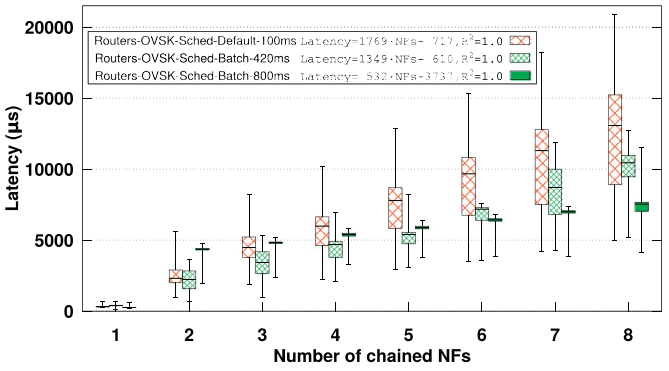


Fig. 9. End-to-end latency (µs) versus the chain's length for a series of FastClick routers, running in containers on top of OVSK. The chains are scheduled either with the default or the batch CFS policies, the latter with different time quanta allocations. The routers run in a single core, OVSK runs in a different core in the same socket, and the input rate is 0.82 Mpps with 64 byte frames. The linear fit to the median latencies, stated in the legend, begins from the chains with 2 NFs.

In this experiment, SCC uses only scheduling acceleration, i.e., operates without I/O multiplexing. This is because while we implemented the I/O multiplexing technique in FastClick, OVSK still relies on its standard I/O mechanism, without using batching. Enabling I/O multiplexing only in the NFs is counter-productive, since packets have to be batched and un-batched multiple times while moving from OVSK to an NF along the chain, hence we did not apply multiplexing in this experiment.

Based on Fig. 9, we can see that the SCC Scheduler realizes the chains with a considerably lower latency compared to the default scheduler. Starting from the chains with 2 NFs, we fit a linear equation to the median latencies for each scheduling scheme to determine the cost of additional NFs in each chain. This cost is 1769 µs per additional NF, when we use the default Linux scheduler (as was previously reported in Section 4.2). Using the batch scheduler, the latency of the same chains is 1349 or 532 µs (30-300% lower) depending upon the size of the time quantum. In the case of the batch CFS with a time quantum of 420 ms, the latency is *always* lower than the default CFS and the scheduling benefits continue to increase with the chain's length. In contrast, using the maximum time quantum (i.e., 800 ms) appears to be an overkill for short chains, as the latency is actually higher than the default CFS. However, if an NFV provider wants to deploy chains with more than 5 NFs, this larger time quantum achieves substantially lower latency, below that of the other two cases.

SCC greatly reduces latency variance. In Fig. 9, the edges of the boxplots (i.e., 25th to 75th percentiles) for the batch CFS cases fall close to each other, hence these chains deliver the majority of packets with low variance. Especially when using batch CFS with the maximum time quantum, the 25th to 75th percentiles almost match. In contrast, when using the default scheduler, the chains exhibit a huge latency variance, that is orders of magnitude greater than the batch CFS cases for long chains. For example, the variance between the 25th and the 75th latency percentiles, for 7 chained routers, when using the default scheduler is 6327 µs. The same percentiles for the same chain, when scheduled by the batch CFS,

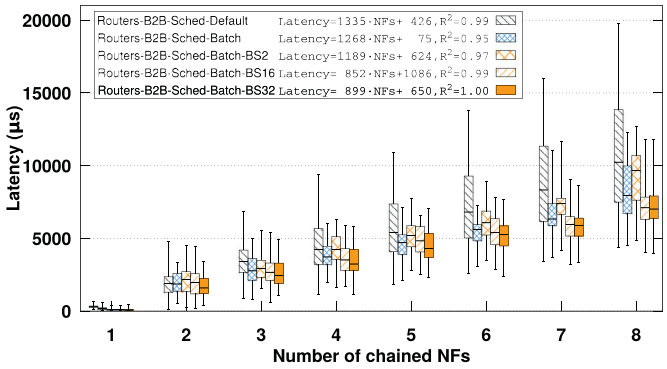


Fig. 10. End-to-end latency (µs) versus the chain's length for FastClick routers, running in B2B chained containers. The top set of chains in the legend are scheduled by the default CFS. The other four chains are scheduled by the batch CFS; the first of them does not use I/O multiplexing, while the remaining use I/O multiplexing with batch sizes 2, 16, and 32 (from top to bottom in the legend). Note that the maximum time quantum is granted to the NFs by the batch CFS in this experiment. The routers run in a single core and the input rate is 0.82 Mpps with 64 byte frames. The linear fit to the median latencies, stated in the legend, begins from the chains with 2 NFs.

differ by 3180 µs when using the 420 ms time quantum, and only 156 µs (40x less than the default CFS) using the maximum time quantum.

6.2.2. Single-core scheduling combined with I/O multiplexing

Next, we combine the benefits of the above scheduling scheme with our I/O acceleration (see Section 5.1.1) to exploit the full capacity of SCC. Hence, we deployed the same chains of routers on SCC, interconnected B2B, without an underlying software switch. This configuration avoids the slow I/O of OVSK and can operate using a batched I/O mode. The chains are scheduled using the uniprocessor scheduling option of SCC shown in Fig. 7a, where one CPU core coordinates the execution of all the routers.

Fig. 10 shows different variants of these chains. The top set of chains in the legend are scheduled by the default CFS, while the other four are scheduled by the batch CFS. In this experiment we use the maximum time quantum for the batch CFS, as we found that it does not exhibit the negative impact observed in Section 6.2.1 for short chains. This is because, when combined with SCC's I/O multiplexing, this scheduling allows the NFs to better exploit this large time slice. To quantify the effects of both the batch CFS and the I/O multiplexing, we tested four different cases. From top to bottom in the legend, the second set of chains use the batch CFS without I/O multiplexing, while the last three sets of chains use I/O multiplexing with batch sizes 2, 16, and 32 respectively.

Looking first at the latencies of the chains scheduled by the default and batch CFSs' (the latter without I/O multiplexing), we notice a similar trend to that shown in Fig. 9. This means that batch CFS is beneficial regardless of the interconnect of the chains. The benefits of the batch CFS can be quantified by looking at the equations fitted to these two cases. Although the per NF latency cost (i.e., the slope in the equations) of the batch CFS is only ∼10% lower (1268 µs versus 1335 µs), one should pay attention to the intercepts of these functions (75 versus 426 µs for a zero length chain). These values indicate the basic processing cost of each chain, which in the case of the batch CFS is almost 7x lower than the default CFS. The batch CFS also reduces the latency variance by up to 2x.

In Section 6.1, we showed the benefits of multiplexing multiple packets into a single system call for a standalone NF. Now, we quantify the benefit of this technique when combined with the increased time quanta per NF allocated by the batch CFS in a scenario with chained NFs. The remaining three sets of boxplots in Fig. 10 show the latencies for these three batch sizes. Our first observation is that with a batch size of 2 system calls, the best results are achieved in the standalone case (see Section 6.1), but this does not have similar performance in a chained scenario. We attribute this fact to the mismatch between the execution time granted by the batch CFS and the frequency of system calls made by the NFs in this case. In other words, batching only two packets at a time for a long chain of processes requires frequent system calls, which leads to yielding the CPU long before a process' time quantum expires.

We observe that when using larger batches, the NFs seem to better exploit their time slices. A close look at the fitted equations proves this fact. A batch size of 2 system calls incurs 1189 µs of latency per additional NF, but with a much greater basic processing cost than the batch CFS without I/O multiplexing, rendering the I/O multiplexing technique as a worse option. However, batch sizes of 16 and 32 packets reduce the per NF latency by up to 50%, while also greatly reducing latency variance. For a chain of 8 routers with the batch CFS using a batch size of 32 system calls the latency reduction is 4x greater than the default CFS. Indeed, comparing the variance between the 25th and the 75th latency percentiles in Fig. 10, half of this reduction is due to the scheduler, while the other half stems from I/O multiplexing.

Finally, we correlate the above latency measurements with the SCC Profiler's data, gathered during the execution of both the OVSK and B2B interconnected chains. Table 8 shows the values of the "SchedContention/RunTime" metric, defined in Section 4.2.1. This metric captures the time spent due to scheduler contention relative to the chain's run-time. For a single router (i.e., chain length equal to 1), there is no corresponding cost because this router is the only process to be scheduled. However, under the default CFS policy, for a chain of 4–8 NFs the time that the chain is runnable but does not execute due to contention in the scheduler is 3-4x greater than the actual run-time of the chain for both OVSK and B2B cases. The batch CFS scheduler employed by SCC reduces this overhead by ∼50%. However, for a chain of 2 routers we observe an increased scheduling overhead compared to the default CFS, especially for the OVSK case. That is confirmed by the increased latency of this particular chain, as depicted in Fig. 9. As explained above, the presence of OVSK does not allow an NF to fully exploit its longer execution time, because of OVSK's ineffective I/O. This is not the case for the B2B chain of 2 routers, since our I/O multiplexing better exploits the time available from the CPU.

The metric "Wait/RunTime", defined in Section 4.2.1, is mostly affected by the I/O mechanism of the chain. We measured a ∼40–60% reduction of this overhead when we used SCC's I/O multiplexing, compared to a chain without this acceleration.

Table 8
Effect of the batch CFS scheduler on the time spent due to scheduling contention with respect to the effective run-time of the service chain for four chain lengths. The last case of B2B chained NFs, labeled as "Batch+MUX", also uses I/O multiplexing of 32 system calls.

	SchedContention/RunTime				
Chain	OVSK Chains		B2B Chains		
Length	Default	Batch	Default	Batch	Batch+MUX
1	0	0	0	0	0
2	1.25	4.34	1.39	2.83	2.82
4	2.88	2.73	3.11	2.92	2.91
8	3.78	2.58	4.17	2.82	2.98

7. Related & future work

Here we discuss the most relevant works with respect to system profiling, scheduling, and network I/O in the general systems'

area and the NFV area in particular. We also briefly sketch our plans for future work.

System benchmarks: The main source of data about a system's performance can be collected by tools that reveal the hardware's performance. For example, lmbench (McVoy and Staelin, 1996) and Intel's memory latency checker (MLC) (Intel, 2013) quantify the latency when transferring data of variable size across different hardware components (i.e., registers, caches, main memory). SCC uses lmbench as a system benchmark. Apart from memory latencies, SCC requires kernel-level benchmarks to measure scheduling and system calls overhead. Although there are relevant available tools such as (Sigoure, 2010), we developed our own benchmarks to acquire these metrics. These benchmarks are available at Katsikas (2016c).

Code Profilers: Modern code profilers, such as OProfile (Levon, 2016) and Perf (Perf, 2016), access low-level performance counters at run-time, providing statistics about applications or the entire OS. Such tools can draw a developer's attention to those functions that exhibit high utilization of system resources, hence these functions offer the greatest potential for acceleration. A great deal of effort is required to understand how the applications under test use the system's resources. Moreover, this knowledge is needed, to instruct these code profilers which particular subset of relevant events, out of a large pool of potential events, are actually relevant. *We performed a study to find crucial NFV performance counters and incorporated Perf in SCC, as we found that it can access these counters*. These counters are illustrated in Fig. 2 and quantified in Table 3.

Data Profilers: Various cache profiling tools have been proposed, such as CProf (Lebeck and Wood, 1994), callgrind's KCachegrind tool (KCachegrind, 2016) (based on valgrind), and Intel's PMU (Intel, 2016d). These tools can track applications' cache utilization allowing a developer to build a map of the system's caches and how they are used. Moreover, likwid (likwid, 2016) provides a broader performance monitoring suite. One can either wrap an entire application to measure its performance with respect to key hardware counters, or enclose a particular piece of code within an application between likwid start and stop functions.

DProf (Pesterev et al., 2010) helps programmers understand cache miss costs by associating these misses with the data types instead of the code. DProf provides clear insights into which objects of an application's data structures incur expensive cache loads; however, DProf mostly focuses on the LLCs and in particular how data moves in and out of LLCs. This focus results from the tool being designed to optimize cross-CPU data exchanges. *The SCC Profiler extends these earlier profilers by keeping track of the entire memory hierarchy (including TLBs and main memory), hence our profiler quantifies both data movements and address translations from a processor all the way to the main memory and correlates this data with OS-level counters*.

Scheduling: Sivaraman et al. (2016) envisioned future switches with programmable boards that allow network administrators to deploy custom packet scheduling schemes. They introduced a Push-In First-Out abstraction model that controls the order and departure of packets, capturing the needs of several packet scheduling schemes. Mittal et al. (2016) explored the possibility of designing a universal packet scheduler that can match the results of any scheduling algorithm, concluding that the Least Slack Time First algorithm is the one that best approximates the universal scheduler.

Scheduling a multitude of processes that comprise a service chain is not addressed by prior scheduling works since these solutions operate at the packet and not at the task level. In contrast, we believe that one should study the performance of task scheduling in NFV to identify ways to achieve better resource utilization. In Section 5.2.1, we studied all the available schedulers of our OS and provided useful insights on their properties with respect to NFV. In Section 6.2, we evaluated the benefits of the batch CFS scheduler and compared it to the default Linux scheduler. As a future work, we would like to evaluate the remaining two real-time policies. Also, an attractive research direction could be to enforce the correct execution order of the chain into a scheduler, in an attempt to completely eliminate the "SchedContention/RunTime" overhead.

Network I/O: Netmap (Rizzo, 2012), DPDK (DPDK, 2016), PFQ (Bonelli et al., 2012), and PF_RING (Deri, 2011) are network I/O mechanisms that boost NFV performance by providing direct access to the ring buffers of a NIC, using custom network drivers. The time required for these tools to be widely adopted in the market motivated a solution that could be immediately adopted by cloud providers. The Linux kernel has substantially evolved over the past decade and today provides sufficient tools to speed up NFV applications running on top of *unmodified* network drivers.

We found that the vectorized I/O technique (Bhattacharya et al., 2003) introduced in version 2.5 of the Linux kernel permits reading/writing frames from/to multiple buffers using a single transaction. This technique is supported by the standard ixgbe driver and can be exploited by activating the scatter/gather feature of the NIC. Our implementation is released as an open source project (Katsikas, 2016a) on top of FastClick. Earlier efforts have successfully applied similar techniques (Rizzo, 2012; Kim et al., 2012; DPDK, 2016) to amortize the system calls' overhead.

As a future work, we aim to further improve the I/O performance of SCC by integrating the asynchronous, zero-copy I/O proposed by Drepper (Drepper, 2006) into FastClick. Based on our measurements reported in Section 6.1, this is expected to reduce the latency imposed by a user-space FastClick router by ∼10x, as currently the best median latency of our fully-featured user-space router is 80 µs and a DPDK router achieves a median latency of 8 µs (as per Section 4.1).

8. Conclusion

We have demonstrated that by *automatically* mining the hardware and OS-level performance counters of NFV service chains using the SCC Profiler, cloud providers can *quickly and reliably* identify performance bottlenecks. To prove this, we built the SCC runtime, a platform that employs acceleration techniques to address the performance issues pinpointed by the SCC Profiler.

Our systematic performance evaluation of standalone and chained NFV services showed that SCC achieves (*i*) 3x lower end-to-end latency and (*ii*) 2x (up to 40x for certain percentiles) lower latency variance by reducing cache misses and main memory accesses and by lowering scheduling overheads. All these are possible using I/O multiplexing and modified scheduling techniques for user-space NFV service chains based on native network drivers. *In summary, using SCC, we managed to make the average case for a user-space NFV service chain to perform as well as the best case*.

Acknowledgments

The research leading to these results has been co-funded by the European Union (EU) in the context of (i) the European Research Council under EU's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110 and (ii) the BEhavioural BAsed forwarding (BEBA) project with grant agreement number 644122.

References

Amazon, 2016a. Compute and Networking Services for Amazon Web Services. Accessed: October 17, 2016, URL http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro/gsg-aws-compute-network.html.

Amazon, 2016b. EC2 Container Service (ECS). Accessed: October 17, 2016, URL https://aws.amazon.com/ecs/.

Amazon, 2016c. Enhanced Networking with SR-IOV. Accessed: October 17, 2016, URL http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.

Barbette, T., Soldani, C., Mathy, L., 2015. Fast userspace packet processing. In: Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems. IEEE Computer Society, Washington, DC, USA, pp. 5–16. URL: http://dl.acm.org/citation.cfm?id=2772722.2772727.

Bhattacharya, S., Pratt, S., Pulavarty, B., Morgan, J., 2003. Asynchronous I/O support in Linux 2.5. In: Proceedings of the Linux Symposium, pp. 351–366. URL: http://www.linuxinsight.com/files/ols2003/pulavarty-reprint.pdf.

Bonelli, N., Di Pietro, A., Giordano, S., Procissi, G., 2012. On multi—gigabit packet capturing with multi—core commodity hardware. In: Proceedings of the 13th International Conference on Passive and Active Measurement. Springer-Verlag, Berlin, Heidelberg, pp. 64–73. doi:10.1007/978-3-642-28537-0_7.

Carpenter, B.E., Brim, S.W., 2002. Middleboxes: taxonomy and issues. Internet Request for Comments (RFC) 3234 (Informational). URL: http://www.ietf.org/rfc/rfc3234.txt.

Deri, L., 2011. Direct NIC Access with PF_RING. Accessed: October 17, 2016, URL http://www.ntop.org/pf_ring/.

DPDK, 2016. Data Plane Development Kit (DPDK). Accessed: October 17, 2016, URL: http://dpdk.org.

Drepper, U., 2006. The need for asynchronous, zero-copy network I/O. In: Proceedings of the Linux Symposium, 1, pp. 247–260. URL: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-247-260.pdf.

Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., Carle, G., 2015. MoonGen: A scriptable high-speed packet generator. In: Proceedings of the 2015 ACM Conference on Internet Measurement Conference. ACM, New York, NY, USA, pp. 275–287. doi:10.1145/2815675.2815692.

European Telecommunications Standards Institute, 2012. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. URL: https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

Hwang, J., Ramakrishnan, K.K., Wood, T., 2014. NetVM: high performance and flexible networking using virtualization on commodity platforms. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, Berkeley, CA, USA, pp. 445–458. URL: http://dl.acm.org/citation.cfm?id=2616448.2616490.

Intel 2016a. Data direct I/O technology. Accessed: October 17, 2016, URL http://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf.

Intel, 2016b. Hardware events for Intel Haswell processors. Accessed: October 17, 2016, URL: https://software.intel.com/en-us/node/589935.

Intel, 2016c. High Precision Event Timers (HPET). Accessed: October 17, 2016, URL http://www.intel.se/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf.

Intel, 2016d. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. Accessed: October 17, 2016, URL https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

Intel, 2016e. Xeon E5 2667 v3 processor. Accessed: October 17, 2016, URL: http://ark.intel.com/products/83361/Intel-Xeon-Processor-E5-2667-v3-20M-Cache-3_20-GHz.

Intel, 2013. Memory Latency Checker (MLC) v3.1a. Accessed: July 5, 2016, URL: https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

Katsikas, G. P., 2016a. FastClick user-space I/O multiplexing using standard Linux network drivers. Accessed: October 17, 2016, URL: https://github.com/gkatsikas/fastclick/tree/mmap.

Katsikas, G.P., 2016b. Realizing high performance NFV service chains. KTH Royal Institute of Technology, School of Information and Communication Technology. Kista, Sweden Licentiate thesis. TRITA-ICT 2016:35, URL: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1044355&dswid=-1520.

Katsikas, G. P., 2016c. System benchmarks. Accessed: October 17, 2016, URL: https://github.com/gkatsikas/system-bench.

Katsikas, G.P., Enguehard, M., Kuźniar, M., Maguire Jr., G.Q., Kostić, D., 2016. SNF: synthesizing high performance NFV service chains. PeerJ Computer Science 2:e98 doi:10.7717/peerj-cs.98.

KCachegrind, 2016. KCachegrind profiler. Accessed: October 17, 2016.

Kim, J., Huh, S., Jang, K., Park, K., Moon, S., 2012. The power of batching in the click modular router. In: Proceedings of the Asia-Pacific Workshop on Systems. ACM, New York, NY, USA, pp. 14:1–14:6. doi:10.1145/2349896.2349910.

Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F., 2000. The click modular router. ACM Trans. Comput. Syst. 18 (3), 263–297. doi:10.1145/354871.354874.

Lebeck, A.R., Wood, D.A., 1994. Cache profiling and the SPEC benchmarks: a case study. Computer Volume 27 (10), 15–26. doi:10.1109/2.318580.

Levon, J., 2016. OProfile: A System Profiler for Linux. Accessed: October 17, 2016, URL: http://oprofile.sourceforge.net/news/.

likwid, 2016. Performance monitoring and benchmarking suite. Accessed: October 17, 2016, URL: https://github.com/RRZE-HPC/likwid.

Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., Huici, F., 2014. ClickOS and the art of network function virtualization. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, Berkeley, CA, USA, pp. 459–473. URL: http://dl.acm.org/citation.cfm?id=2616448.2616491.

McVoy, L., Staelin, C., 1996. lmbench: portable tools for performance analysis. In: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA. 23–23, URL: http://dl.acm.org/citation.cfm?id=1268299.1268322.

Mittal, R., Agarwal, R., Ratnasamy, S., Shenker, S., 2016. Universal packet scheduling. In: Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation. USENIX Association, Berkeley, CA, USA, pp. 501–521. URL: http://dl.acm.org/citation.cfm?id=2930611.2930644.

Network Functions Virtualization (NFV), European Telecommunications Standards Institute (ETSI), Industry Specification Group (ISG), 2013. Network Functions Virtualization (NFV); Terminology for Main Concepts in NFV, pp. 1–10. URL: http://faculty.cs.gwu.edu/~timwood/papers/16-HotMiddlebox-onvm.pdf.

Open vSwitch,. An Open Virtual Switch. Accessed: October 17, 2016, URL http://openvswitch.org.

Pabla, C.S., 2009. Completely fair scheduler. Linux J 2009 (Volume 2009, Issue 184). URL: http://dl.acm.org/citation.cfm?id=1594371.1594375.

Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., Shenker, S., 2015. E2: a framework for NFV applications. In: Proceedings of the 25th Symposium on Operating Systems Principles. ACM, New York, NY, USA, pp. 121–136. doi:10.1145/2815400.2815423.

Perf,. Linux profiling with performance counters. Accessed: October 17, 2016, URL https://perf.wiki.kernel.org/index.php/Main_Page.

Pesterev, A., Zeldovich, N., Morris, R.T., 2010. Locating cache performance bottlenecks using data profiling. In: Proceedings of the 5th European Conference on Computer Systems. ACM, New York, NY, USA, pp. 335–348. doi:10.1145/1755913.1755947.

Quinn, P., Nadeau, T., 2015. Problem statement for service function chaining. Internet Request for Comments (RFC) 7498 (Informational). URL: http://www.ietf.org/rfc/rfc7498.txt

Rizzo, L., 2012. Netmap: a novel framework for fast packet I/O. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference. USENIX Association, Berkeley, CA, USA. 9–9, URL: http://dl.acm.org/citation.cfm?id=2342821.2342830.

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., Sekar, V., 2012. Making middleboxes someone else's problem: network processing as a cloud service. In: Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. ACM, New York, NY, USA, pp. 13–24. doi:10.1145/2342356.2342359.

Sigoure, B., 2010. Custom tools for measuring the cost of context switching. posted at 11/14/2010 08:53:00 PM, http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.

Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., McKeown, N., 2016. Programmable packet scheduling at line rate. In: Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference. ACM, New York, NY, USA, pp. 44–57. doi:10.1145/2934872.2934899.

Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., Ramakrishnan, K., Wood, T., 2016. OpenNetVM: a platform for high performance network service chains. In: Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization. ACM. URL: http://faculty.cs.gwu.edu/~timwood/papers/16-HotMiddlebox-onvm.pdf.

Georgios P. Katsikas received his B.Sc. and M.Sc. diplomas from the Department of Informatics and Telecommunications at the National and Kapodistrian University of Athens (NKUA) in 2010 and 2012 respectively. Since July 2014, he has been pursuing his doctoral studies at the KTH Royal Institute of Technology under the supervision of Prof. Dejan Kostic and Prof. Gerald Q. Maguire Jr. His main research interests include Computer Networks and Operating Systems, with a focus on Network Functions Virtualization (NFV) and Software-Defined Networking (SDN).

Gerald Q. Maguire Jr. FIEEE, completed his Ph. D. (1983) and M. S. (1981) in Computer Science at the University of Utah and a B.A., magna cum laude, in Physics at Indiana University of Pennsylvania (1975). Since July 1994 he has been Professor of Computer Communication at KTH Royal Institute of Technology. Prior this he was on the faculty of the Computer Science Department of Columbia University in the City of New York (in various positions starting in 1983). He has been a Gastprofessor at Graz University of Technology (spring 1994), Acting Professor (July 1993 to June 1994) and Guest Professor (July 1992 to December 1992) at KTH, and an Invited Lecturer in Local Area Networks and Image Processing at Leiden State University (Spring 1986). During 1991 and 1992 he was Program Director for Experimental Systems, U.S. National Science Foundation, Directorate for Computer and Information Science and Engineering, Division of Microelectronic Information Processing systems. For the academic year 1983–1984 he had a Fulbright-Hayes Fellowship at Karolinska Institutet.

Dejan Kostic obtained his Ph.D. in Computer Science at the Duke University. He spent the last two years of his studies and a brief stay as a postdoctoral scholar at the University of California, San Diego. He received his Master of Science degree in Computer Science from the University of Texas at Dallas, and his Bachelor of Science degree in Computer Engineering and Information Technology from the University of Belgrade (ETF), Serbia. From 2006 until 2012 he worked as a tenure-track Assistant Professor at the School of Computer and Communications Sciences at EPFL (Ecole Polytechnique Federale de Lausanne), Switzerland. In 2010, he received a European Research Council (ERC) Starting Investigator Award. From 2012 until June 2014, he worked at the IMDEA Networks Institute (Madrid, Spain) as a Research Associate Professor with tenure. He is a Professor of Internetworking at KTH since April 2014.