Inference of virtual network functions' state via analysis of the CPU behavior

Charles Shelbourne
BP R&D
London, United Kingdom
charles.shelbourne@bp.com

Leonardo Linguaglossa
Telecom Paris
Paris, France
linguaglossa@telecom-paristech.fr

Tianzhu Zhang
Nokia Bell Labs
Paris-Saclay, France
tianzhu.zhang@nokia.com

Aldo Lipani
University College London
London, United Kingdom
aldo.lipani@ucl.ac.uk

***Abstract*—The ongoing process of *softwarization of IT networks* promises to reduce the operational and management costs of network infrastructures by replacing hardware middleboxes with equivalent pieces of code executed on general-purpose servers. Alongside the benefits from the operator's perspective, new strategies to provide the network's resources to users are arising. Following the principle of "everything as a service", multiple tenants can access the required resources – typically CPUs, NICs, or RAM – according to a Service-Level Agreement. However, tenants' applications usually require a complex and expensive measurement infrastructure to continuously monitor the network function's state. Although the application's specific behavior is unknown (and often opaque to the infrastructure owner), the software nature of (virtual) network functions (VNFs) may be the key to infer the behavior of the high-level functions by accessing low-level information, which is still under the control of the operating system and therefore of the infrastructure owner. As such, in the scenario of software VNFs executed on COTS servers, *the underlying CPU's behavior can be used as the sole predictor for the high-level VNF state* without explicit in-network measurements: in this paper, we develop a novel methodology to infer high-level characteristics such as throughput or packet loss using CPU data instead of network measurements. Our methodology consists of (i) experimentally analyzing the behavior of a CPU that executes a VNF under different loads, (ii) extracting a correlation between the CPU footprint and the high-level application state, and (iii) use this knowledge to detect the previously mentioned network metrics. Our code and datasets are publicly available.**

I. Background

The increasing demand for scalability in cloud environments and data centers has challenged the classical network approach to adopt custom ASICs to deploy network functions. Instead of static, expensive hardware middleboxes, operators have started to use a more flexible approach based on pure software, which advocates implementing network functions (e.g., forwarding, firewall, IDS, etc.) as pieces of software to be deployed and executed on commercial off-the-shelf (COTS) hardware [1]. While the main drawback of these flexible approaches is generally related to the performance limitations w.r.t. ASICs specifically manufactured to execute a single function, several advances in the state-of-the-art high-speed packet processing engines, aka *software routers* or *VNF routers*, have brought line-rate capabilities to COTS servers [20], [35]. As a result, many frameworks for software networking are nowadays capable of achieving multi-10 Gbps performance, and a smaller and smaller gap compared to hardware-based systems. Among the most common techniques to speed up the processing rate, we have batching (both I/O and compute), polling, and NUMA awareness to cite a few, and are all implemented by the most popular libraries for high-speed packet processing such as netmap [27] or DPDK [16]. Batching and polling, in particular, are known to be very effective in high-load scenarios: one (or more) CPU(s) are statically allocated to perform only the packet processing, and the available budget in clock cycles is 100% allocated to perform the network-related instructions, thus mitigating the interrupt pressure [5], [27].

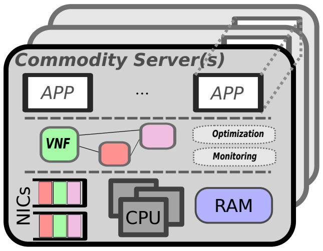


Fig. 1: A COTS server with the low-level resources, the deployed VNFs, and the high-level network applications.

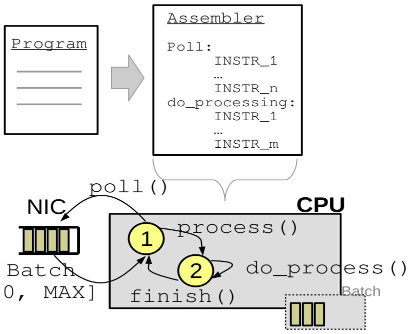
This current trend of "network softwarization", fueled by the growing popularity of Software Defined Networking (SDN) [24] or Network Function Virtualization (NFV) [1], can significantly reduce the maintenance cost of network services, as the life-cycle of software evolves much faster than that of hardware-based solutions [37]. Moreover, it opens new business models for network services. For instance, as represented in Fig. 1, a server owner may provide some resources such as CPUs, Network Interface Cards (NICs), or storage to the tenants following a Service-level agreement (SLA). Tenants can then utilize the allocated resources to deploy the VNFs that create the high-level application (which can scale both horizontally and vertically). It is important to underline that the server owner cannot usually access the deployed VNFs or the input traffic. In the context of high-speed networking applications, NICs typically operate in *kernel-bypass* mode [21], which removes their visibility from the operating system (and therefore from the server owner). Moreover, several existing tools provide further isolation between the tenants and the server owner, e.g., BlindBox [29] and Embark [18] protect user traffic from server owners through encryption, while ShieldBox [33], TrustedClick [8], and SGX-BOX [12] prevent the server owners from directly accessing the VNFs. SafeBricks [26], SafeLib [22], and LightBox [9] protect both traffic and VNFs from unauthorized entities. This implies that the classical operations of monitoring and optimization must be performed independently by the server owner and the tenants [28], which may require invasive techniques such as traffic mirroring or per-NIC active probing [23]. In all the above situations, the limited visibility about global resource usage may lead to an inefficient resource allocation. Moreover, a natural extension of the detection of *current resource requirement* is the *prediction of future resource usage* in line with the actual demand. While this is a classic problem, in modern cloud environments it is challenging to predict the evolution of the system in a short time scale (e.g., tens of minutes) that reflect a common usage pattern for the cloud behavior [10].

To solve these problems while, at the same time, reducing the amount of measurement on live traffic, we propose a novel approach that exploits the interactions among components at low-level (CPU, NICs, Memory) and high-level (VNFs, service chains, end-to-end applications). Within the context of software networking, a network application is a collection of VNFs, each consisting of pieces of code that are ultimately translated into a sequence of operations executed on low-level components to perform simple operations (instructions, branches, memory accesses). In this duality, finding a correlation between high-level and low-level behavior (i.e. inferring what is the low-level pattern associated with the high-level application) is the key to solve the aforementioned problem without altering the typical workflow of modern high-speed network applications. Although third-party VNFs might not be visible to the OS, the low-level information related to the CPUs usage (which are still under the control of the operating system, and therefore of the service provider) are easy to monitor and can thus be thought of as a *network fingerprint*, which is an indicator to infer the behavior of the high-level network application. In a nutshell, the CPU behavior will reflect *both* the actual VNF processing, as well as the input network conditions.

In this paper, we verify the validity of this approach by performing an extensive experimental campaign from which we gathered a large dataset openly available at [32]. We selected several high-speed open-source software routers where we deploy a simple L2 forwarding VNF, and we collect low-level CPU measurements by running the `perf` [2] command. Among our contributions, (i) we provide several insights on the evolution of the CPU patterns reflecting the different high-level scenarios; (ii) we support and empirically prove that such patterns are sufficient to act as predictors to infer the high-level network applications under different conditions; (iii) we discuss the potential applications of our findings.We present our intuition within the context of high-speed software networks in Sec. II. We then describe our experimental analysis in Sec. III. In Sec. IV we describe three case studies for our methodology to detect a state change in VNF, a packet loss regime, or for input traffic detection. We discuss our findings in Sec. V and Sec. VI concludes the manuscript.

II. Effect of VNF processing on CPU's behavior

To provide the ground for our inference model, we now briefly describe the scenario of a software router running one or more VNFs (Sec. II-A) and then we show the important low-level components of the CPU behavior and how the high-level application can have an impact on them (Sec. II-B).



```
Poll:
      INSTR_1
      …
      INSTR_n
do_processing:
      INSTR_1
      …
      INSTR_m
```

Fig. 2: The CPU is executing some instructions as a result of the translation from a high-level programming language.

A. VNF execution on a single-core CPU

The objective of software routers is to receive packets from the physical NICs, perform some processing, and steer the traffic towards either a physical NIC or another VNF. Via Receive-side scaling (RSS) [15] the input traffic can be partitioned into different flows, which are assigned to one (or more) CPU(s) for the processing required by the respective tenant. For simplicity, we will only focus on the single-CPU case, where the incoming traffic is managed by a single CPU core running the VNFs of a single-tenant (we describe how to generalize our approach in Sec. V).

Modern general-purpose CPUs in COTS servers (e.g., Intel Xeon, AMD Ryzen) are intrinsically complex systems with multiple pipelines, each processing instructions in a sequence of stages [14]. With pipelining it is possible to obtain instruction-level parallelism by allowing a single CPU core to issue *more than one instruction with a single clock cycle*: this is measured by the instructions per clock cycle (IPC) metric. When instructions are independent, and no interrupts are stopping the execution of the CPU, the IPC can be maximized. On the contrary, (i) when the instruction execution depends on some previous results, the pipelines must be stalled to wait for the said result to be finalized; (ii) when some interrupts are received due to some I/O, the CPU performs a context-switch and executes the corresponding interrupt handler, thus resetting the pipeline and repopulating it with the new instructions. Furthermore, CPUs are equipped with a hierarchy of small but fast cache memories (usually from L1 to L3), for both instructions and for data, which can significantly speed up the computation rate of the CPU by reducing the memory access latency. Finally, modern CPU try to proactively populate the internal pipelines thanks to extensive usage of *branch predictors*, which allows the CPU pipeline to advance even if the result of some True/False conditions is not known yet, by considering what would be the most likely outcome and verifying it after the result is ready.

Software routers extensively exploit such architecture to provide high-speed packet processing. In this context, the actual packet processing is implemented as a regular program, and as such it implies the translation of the program code into some low-level instructions executed by the CPU, as shown in Fig. 2. Two of the commonly adopted techniques to accelerate packet processing in software are *polling* and *batching*. Polling aims at reducing the interrupt pressure for incoming packets, as the CPU would otherwise be overwhelmed by the continuous context switches generated by incoming packets. On the other hand, batching is used to share both the I/O and the computational costs over a group of packets, as well as to populate the CPU instruction and data caches. In summary, both polling and batching can accelerate packet processing by providing an optimal utilization of the CPU pipelines and the low-level caches.

B. Inference model of the VNF behavior

To monitor the behavior of VNFs, classical approaches require obtaining measurements from three sources: NIC, CPU, and hypervisor. NIC-based measurements are used to quantify the bandwidth requirements (i.e., the number of processed packets per second) or the packet loss. Standard solutions such as Cisco's NetFlow [7] require custom ASICs, while software-based solutions may need to sample the incoming traffic. Hardware solutions are expensive and unscalable, while software solutions are slow and invasive, as they may alter the processing rate. Both CPU and hypervisor data are used by operators to correctly provision the server resources to the tenant. Since VNFs are usually deployed as virtual machines or containers, it is possible to detect the CPU utilization and, in the case of several inter-VM communications, optimize the deployment of multiple VNFs.

Although it is possible to adopt lightweight approaches to access the NIC counters [36], operators still need to access the underlying infrastructure to monitor the CPU and the VNF behaviors. Since the CPU utilization can significantly vary, depending on the instruction sequence to be executed (as shown in Sec. III), our approach relies solely on the CPU data to infer all the aforementioned metrics. We identify three main components affected by the changes in the CPU execution, namely: *computation*, *cache accesses* and *memory operations*.

Computation refers to how the VNF code is executed by the CPU. For instance, when the VNF must execute several independent instructions, the IPC may increase, and it can be quantified by counting the number of instructions issued per time unit. Similarly, when the code becomes more complex, the execution path can take different branches depending on the result of some `if` conditions, thus affecting the number of branches and branch mispredictions. Cache accesses reflect the code and data complexity of the high-level VNF. A simple forwarding function is likely to be executing a small set of instructions all the time, which increases the probability of both instructions and data in the L1-L3 caches. On the contrary, complex VNFs that require more instructions can incur a high number of cache misses. However, even in the case of complex VNFs, if the traffic is static (e.g., same source/destination for subsequent packets) the data caches may still experience several hits. In general, as not all the code may be placed within the internal caches, the VNF must eventually perform some memory accesses (though for high-speed applications this should be avoided as much as possible). Furthermore, if inter-VM communication occurs, the internal buses are used to move data across different processors of the server. In *nix OS, the system memory is organized in blocks called *pages* [6]. When a memory operation occurs, the bus utilization is higher: while caches are located on the same die of the CPU, RAM is usually an external piece of semiconductor that is connected via a bus. Whether memory access is required or not, a *Translation Lookaside Buffer* (TLB) is accessed to map the virtual addresses assigned to the program and the physical addresses of the main memory[1].

We now experimentally verify how such components are affected by the actual execution of VNFs, and we use the obtained results of the low-level behavior to infer the high-level state of the VNF.

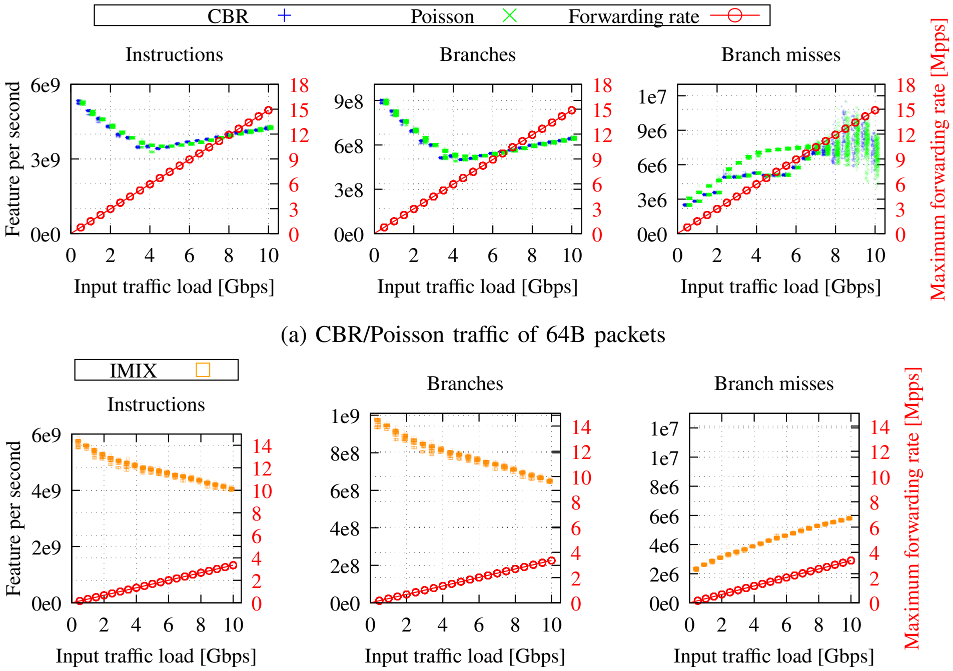
III. Experimental analysis of CPU features under different loads

For our experimental campaign, we select six state-of-the-art VNF routers: *vpp* [4], *ovs* [3], *fastclick* [5], *t4p4s* [17], *snabb* [25], and *bess* [13]. In each test, we deploy an instance of the VNF router in our hardware, which consists of a COTS server with two Intel Xeon E5-2690 v3 CPUs @ 2.60 GHz (each with 12 cores and 32k/256k/30720K L1-3 caches) and two Intel 82599ES dual-port 10 Gbps NICs. On one NIC, we run the MoonGen traffic generator [11] to inject packets towards another NIC with a combination of transmission rates, packet sizes, and traffic patterns. The transmission rate is between 0 Gbps (no link utilization) and 10 Gbps (full saturation of the link); the packet size can be 64B, 256B, or IMIX (i.e., a mixture of various packet sizes); the traffic pattern can be constant bit rate (CBR) or with Poisson inter-arrival time. On the other NIC, we execute a VNF router that is statically assigned to a CPU core isolated from the OS scheduler. With a sampling rate of 1s, we query the `perf` tool to collect the measured value of low-level CPU features (the full list of the features is provided at [32]). The router executes an L2 forwarding VNF. In this section, we focus on a single VNF router (for its ease of management and configurability we opted for *fastclick*) and we continuously generate traffic with different characteristics to observe the impact of traffic-related factors such as the input rate and traffic pattern.

A. Computation: Instructions and Branches

We first analyze the computation components (i.e., number of instructions, branches, and branch mispredictions issued per time unit) and report our `perf` measurements in Fig. 3. The figure shows the impact of the input traffic rate on the computation features of the CPU for *fastclick*. For comparison, we plot on the right axis the maximum forwarding rate, which depends on the packet size: with 64B packets and 10 Gbps

[1] The TLB is also accessed in parallel with the caches, to reduce the latency of memory accesses. See more on Chap. 2 of [6]



(a) CBR/Poisson traffic of 64B packets

(b) CBR traffic of IMIX packets

Fig. 3: Computation features per time unit in *fastclick* as a function of the input rate

processing, the packet rate, measured in millions of packets per second (Mpps) is 14.88. In Fig. 3a, we observe that for both CBR and Poisson traffic, the instructions and the branches follow a similar non-linear pattern. When the rate is low, the measured value of such features is high, and it decreases with the input rate up to a knee point (at about 4.5 Gbps, or 6.7 Mpps) where the observed value starts increasing again. This behavior is related to which particular state the CPU is operating (cfr. Fig. 2). At a low rate, the processing is dominated by the polling and it has its maximum IPC, thus increasing the number of instructions per second. As the CPU performs the busy polling with no live traffic, the low-level pipelines are likely populated with the same subset of instructions, and the number of branch mispredictions is low: this reflects a high instruction rate (more than 5 billion per time unit) and a low branch miss rate (around 3 million per time unit). Given the CPU clock frequency of our system, we observe that in this state the CPU can execute more than two instructions with the same clock cycle, thus showing an IPC value greater than 2. When the rate increases, the system approaches an equilibrium between the polling and processing state, which leads to the IPC reduction due to the increased code complexity and the frequent switching between different states. After 4.5 Gbps, the CPU spends more time in the processing state than in the polling state, which allows the CPU to efficiently utilize the internal pipelines with a subsequent IPC increase. The number of branch-misses per second increases with the traffic rate up to a saturation point (around 7 Gbps/10.4 Mpps) where the average value is constant but the observed values become more scattered. Incidentally, beyond 8 Gbps we start observing packet drops. In Fig. 3b, we observe that for IMIX traffic the change of the features is monotonous: this is because the packet processing rate never exceeds 4 Mpps (as shown by the red line), which means the CPU never reaches the balance point between polling and processing.

B. Data and instruction caches

Caches' behavior reflects the locality of instructions and data during the VNF processing. In other words, similar processing and/or the utilization of the same data can positively affect the CPU caches such that (i) instructions can be found in the lowest level of caches, and (ii) the CPU can finalize the computation without memory access. In Fig. 4, we show the scatter plot of the aforementioned cache-related features as a function of the input rate for *fastclick*. A first observation is that the input rate significantly affects both the cache references and the L1 d-cache misses, with a minor impact on the L1 i-cache misses, independently from the generation pattern. The increase in the input rate corresponds to a higher packet-processing rate, and therefore the VNF router will have a greater probability to reference some cached data (incrementing the cache references) or to miss in the L1 i-d-caches. The generation pattern does affect the L1 i-cache behavior, with the IMIX being consistently smaller than the CBR/Poisson cases, which in turn reach a saturation point at about 8 Gbps where no increment is observed. A second observation refers to the fact that cache misses are not affected by the input rate. In a steady-state condition, the L2-L3 caches are likely to be filled with the data that the CPU would not find in the L1 caches, and since our VNF is a simple L2 forwarding function, we have a bound in the number of misses that the

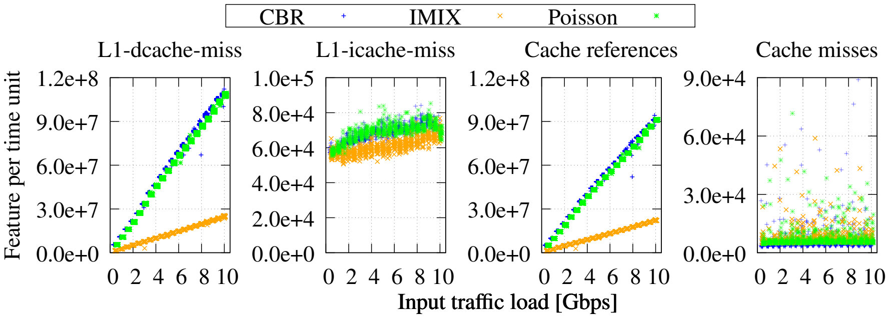


Fig. 4: Caches' features per time unit as a function of the input rate for CBR/Poisson traffic of 64B packets and CBR traffic of IMIX packets in *fastclick*.

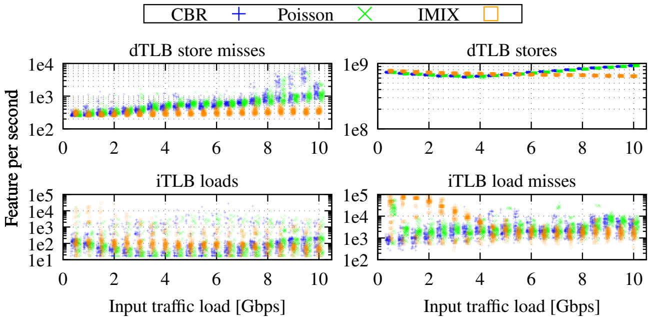


Fig. 5: *fastclick* address translation.

CPU will experience, being that limited to the L1 cache only. It is important to underline that this pattern may change with more complex VNFs, where even the data in the L2-L3 caches might be evicted by the caches' replacement policy.

C. Address translation: virtual memory

Whether the instructions or data to be retrieved are in the caches or not, the CPU will access in parallel the TLB to translate a virtual address (local to the program) to a physical address (shared among all processes of the system). During the translation, the TLB also verifies if the desired item is to be found in the L1-L3 cache, in the main memory, or a swap to disk has occurred. We remark that the latter scenario never happens for high-speed network applications by setup. For this component, we select the loads operations and the missed loads for the iTLB, as well as the stores and the "missed stores" for the dTLB. A store miss can occur when (i) the translated address cannot be stored on the first level of TLB, but can be stored on the second level, or (ii) the store fails on all levels and additional operations are needed[2].

As for the impact of the input rate on the TLB features, we observe in Fig. 5 that the input rate for the CBR/Poisson traffic has an impact on the steady-state value of the dTLB-stores and iTLB-loads that is similar to the knee plot previously observed for the compute-related features. IMIX traffic does not show this behavior. The dTLB store-misses grow linearly with the input rate (CBR/Poisson), but remain constant for the IMIX traffic, because of the increasing packet processing rate that may require new address translations for the new data, and it may not fit on the first-level TLB. A similar linear dependency is found on the iTLB load misses, although the IMIX traffic causes this feature to decrease up to 6 Gbps, to stay constant until 10 Gbps.

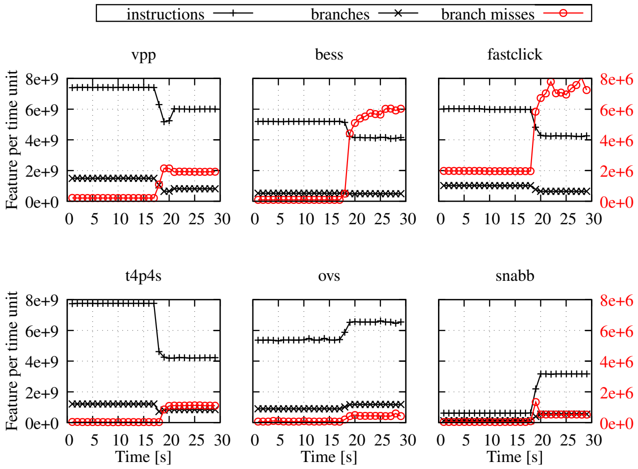


Fig. 6: Number of instructions, branches and branch misses as a function of time in the on-off experiment for 10 Gbps CBR traffic of 64B packets.

IV. Applications

We now show three applications of this methodology to infer the VNF's behavior via the CPU fingerprint. First, with the *on-off experiment*, we show how to detect a state change in the VNF. Second, the *packet-loss experiment* will show that it is possible to detect packet loss without accessing the NIC counters. Finally, with the *traffic injection experiment*, we show how it is possible to accurately recognize the injected traffic typology as well as the traffic rate.

[2] This is called a *page walk*, but a detailed discussion is out of the scope of this manuscript.

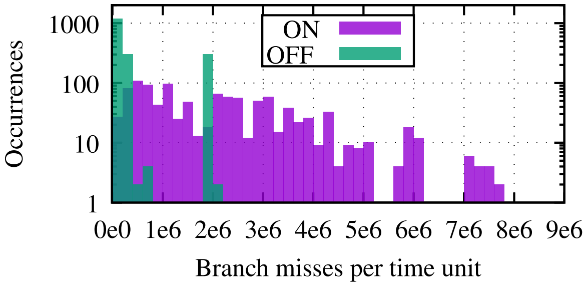


Fig. 7: Aggregated histogram of branch misses in the on-off experiment for all the VNF routers and all traffic rates.

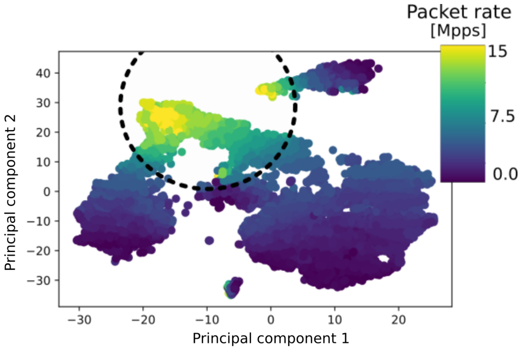


Fig. 8: Packet loss detection via Principal Component Analysis for fastclick.

A. Detecting a VNF state change

Since a VNF state change mostly impacts the packet processing, we engineer this experiment to focus on the CPU computation features only. This experiment consists of two stages: in the first half, the VNF router does not receive traffic (and therefore only a busy-polling is executing); in the second half, we generate some traffic with MoonGen and we observe the variation in the CPU features due to the change in the state of the VNF. We repeat the experiment for all the chosen VNF routers. Figure 6 shows the values collected every second by `perf` of the number of instructions, branches, and branch-misses per time unit. In the first half, we start the VNF router in polling mode: as no traffic is present, the measurements refer to the polling state of the CPU. In the second half, at about 17 seconds, MoonGen starts transmitting packets to the VNF router at the maximum rate.

It is easy to discern a low-traffic vs. high-traffic duality, despite the behaviors of individual VNF routers may differ. Most VNF routers present a decrease in the number of instructions issued per time unit. This is justified by the fact that the polling state (cfr. Sec. II) utilizes a very small set of instructions, resulting in optimal utilization of the internal pipelines and caches (although such instructions are not used to perform any actual packet processing). When MoonGen sends traffic, the CPU consistently switches between polling and processing: every time the CPU switches from one state to the other, new instructions must be added to the pipelines, thus reducing the IPC efficiency. A similar explanation holds for the number of branches and branch-misses: although the code executed during the polling state may have a high number of ”if conditions”, the outcome is almost always correctly predicted, and the number of branch-misses approaches zero. When some traffic is received, the VNF router executes the processing code that presents a higher value of unpredictability and the number of missed branches will increase accordingly. This further confirms the analysis of Sec. III-A.

Both the *snabb* and the *ovs* experiments show an increase in all the computation features when the traffic appears, which is due to the CPU not being used when no traffic is present (as for the *snabb* experiment, where the number of instructions per second is significantly less than that of the other VNF routers) or to the datapath not being accessed when no traffic is present, or just to the fact that the polling state has a simpler code than the VNF processing. However, a predictor component must only take into account a variation in the observed value, rather than the numerical absolute values. As such, it can be possible to detect a state change for all the behaviors of our VNF routers.

Despite individual VNF routers may behave differently, we have observed that our collected data for all the routers and different traffic rates is sufficient to effectively estimate the current VNF state or a state change with very high accuracy and little calibration. Although the design of an effective estimator is out of the scope of the current manuscript, we have performed some preliminary tests as proof of concept for our approach. As the branch misses consistently shows a very high correlation with the VNF state, we plot in Fig. 7 the observed values for the branch misses in the ON and OFF scenarios, in the form of a log-scale histogram, for all VNF routers and different traffic rates in the range [1, 10] Gbps. We observe that, independently of the VNF router, when no VNF is running the branch misses are mostly clustered around the minimum, while a running VNF causes a higher number of branch misses. Our naive estimator leverages the data distribution to guess whether an observation belongs to the ”ON” or ”OFF” state. The estimator’s output is ”OFF” when the observation is found within a standard deviation distance from the minimum, and ”ON” otherwise. Even in these simplistic settings, the accuracy is 0.85. We plan to develop more complex estimators that can take into account different computation features as future work.

B. Packet-loss detection

We now show how to detect a lossy regime without accessing the NIC counters. To do so, we collected all the available data for *fastclick*, and performed a principal component analysis (PCA) across all the features, to simplify the data visualization and find the presence of some clusters of data.

Figure 8 shows the PCA data where data points are additionally colored depending on the input packet rate (i.e., the number of packets per second injected to our VNF routers). As expected, we observe that the two principal components variable carry enough information to separate data points

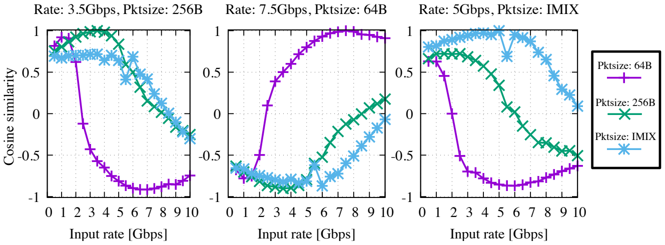


Fig. 9: Cosine similarity of different traffic categories for fastclick.

according to the incoming traffic rate. Our CPU is capable of performing about 12 Mpps (8 Gbps of 64-byte packets) without incurring packet drops. We underline that such values are clustered in a circular area centered at $(-8, 30)$. Therefore, it is possible to use a simple PCA classifier that periodically gathers the CPU data, finds the PCA coordinates in the PCA space, and detects whether the measurement is taken within the lossy area or not. We underline that some approaches exist which can detect the packet loss regime of VNF routers [19], but they work offline and cannot be easily deployed in the same server as the VNF. We also point out that it is only required to perform the full PCA analysis once (for data collection and the PCA space representation). After that, the PCA transformation is known, and therefore a simple coordinate change of the collected CPU observation and the PCA space is sufficient to check whether such observation falls within the lossy region or not.

C. Traffic classification

This application consists of identifying the traffic characteristics injected into a VNF router (including the input rate, the packet size, or the packet distribution) without relying on traffic mirroring, sampling, or invasive measurement software. For this case study, we will use all the collected features.

In the following, we consider each CPU-measurement as a vector of features V consisting of different components V_i, where each component is a CPU feature (such as the number of instructions, branches, or cache misses). The dimension n of the vector is the number of features that we obtain from `perf`. We then use *fastclick* with a combination of different input rates in the range [0, 10] Gbps, different packet sizes in the set {64-byte, 256-byte, IMIX} and different generation patterns of {poisson, CBR}. Each element of the cartesian product of these sets represents a scenario, for example ”5 Gbps traffic of 64-byte packets with poisson inter-arrival”. For each scenario we collected several CPU-measurement vectors $\{V^1, V^2, ..., V^m\}$ where m is the number of measurements. We define for each scenario a *representative vector* V^R as a vector whose components are the average value of all the observed measurements for that scenario: $V^R = \{V_1^R, V_2^R, ..., V_n^R\}$, where $V_i^R = \frac{1}{m} \cdot \sum_{j=1}^{m} V_i^j$.

When a new measurement V^* is obtained, we use the cosine similarity [30] to quantify the difference of the new vector w.r.t. each representative vector V^R. The cosine similarity approaches 1 when the two vectors are very similar, while it is equal to 0 when the two vectors are orthogonal (i.e., very different). The idea is to map CPU behavior patterns into network-related patterns using a vectorial distance metric. The details of the cosine similarity are briefly discussed in [30].Fig. 9 shows the cosine similarity values for three scenarios (for simplicity, we only show Poisson traffic). For each plot, the x-axis represents a potential matching input rate, while the colored curves represent the potential packet size distribution. Intuitively, the coordinates of the curve that reaches the maximum value should be taken as the inferred scenario. Starting from the left plot, the point reaching the maximum belongs to the ”256B” curve at 3.5 Gbps, perfectly matching the expected scenario.

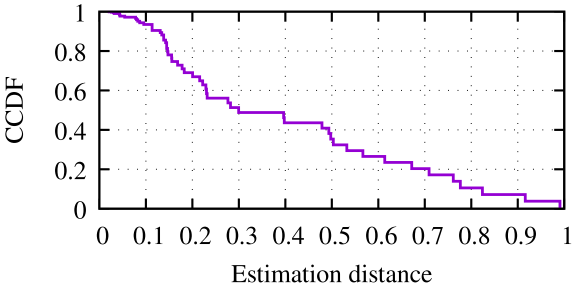


Fig. 10: Complementary CDF of the distance between the inferred experiment and the second best guess.

It is worth noticing that another high-similarity is the ”64B” at 1.5 Gbps. This means that the two scenarios are very similar, but since this false-positive value is still smaller than the maximum, it is correctly discarded. Analogously, the central and the right plots show two different scenarios that can be correctly classified with cosine similarity. We define the *estimation distance* as the difference between the maximum value and the second-best guess. As with the previous example, a low estimation distance means that two observations are very similar, while a large estimation distance means that two observations are very different. We report in Fig. 10 the complementary cumulative distribution function of the estimation distance obtained with *fastclick* and all of our experiments. This plot shows that more than 90% of the estimation distance is greater than 0.1, and 20% of the estimation distance is between 0.7 and 1 (with 1 being completely orthogonal). We highlight that the minimum value of the estimation distance is 0.02, but even at this extreme the cosine similarity correctly classified both the traffic rate and the traffic pattern.

V. Discussion

The experimental analysis that we presented supports the opportunity *to indirectly infer the high-level state of a VNF executing some network function by observing only the low-level CPU features* for any traffic type. This is achieved by interpreting the CPU features as a network fingerprint associated with the high-level VNF application, without interfering with additional measurements. A similar approach has been proposed in the data mining community to detect misbehavior in some cryptocurrency mining [31], and there is a large potential for adoption in the network community. We now discuss the main limitations of our approach, and the required effort to extend our research for other applications.

A. Overhead and calibration of the `perf` data collection

Using CPU-related features for deriving a network fingerprint still has a cost, even if this cost is being shifted to the CPU instead of the measurement infrastructure. As shown in [34], the overhead of the `perf` counters significantly varies with the adopted OS kernel and the technique used for collecting the data. In particular, *self-monitoring*, i.e. when the code to be monitored is instrumented to collect the measurements at specific execution points, is the most expensive, showing an overhead in the order of thousands of clock cycles. However, our objective is to avoid the usage of invasive techniques to monitor the CPU behavior, and therefore we do not modify any VNFs to collect our data. We adopt a statistical sampling approach, which relies on the OS to periodically collect the aggregated counters within the specified time interval. We performed our experiments with time intervals of 100ms, 1s, and 5s, and we finally opted for the granularity of 1s. With this approach, no code modification is required: it is sufficient for `perf` to link to the process ID of the VNF to be monitored. Furthermore, the overhead is only occurring once per time interval, thus having a minimal impact of about a thousand clock cycles per time unit, which is negligible considering that the budget of available clock cycles for a multi-GHz CPU is a few billion per second. Finally, depending on the VNF to be monitored, a finer or a coarser approach can be adopted.

B. Multiple VNFs and multiple cores

Whereas the quantitative measures may vary, the qualitative analysis can be generalized with further exploration on more complex scenarios involving multiple VNFs assigned to the same core, or the same VNF distributed to multiple cores. In the former scenario, there may be contention between different VNFs on the same core may lead to difficulty in inferring the actual application. This will result notably in computation features (such as instructions or branches) to be shared among the different VNFs. We note that the `perf` tool is capable of monitoring the instructions executed by different processes (with different process IDs) even if they are executed on the same CPU. Therefore, we plan to measure the effect of multiple VNFs pinned to the same core, as it would likely result in a shared allocation of the available clock cycles to the different VNFs (depending on the OS's scheduler). Albeit our methodology is shown for a single-core scenario, it can also be generalizable to the multicore case. When several VNFs are running on different cores the computation can be assumed as independent, but cores deployed to the same NUMA node will share the last level caches. In this case, the server's operating system can still use the `perf` tools to obtain the measurements from different CPUs. A piece of software can then be used at OS-level to periodically collect data from all running VNFs and perform the proposed analysis on a per-core basis.

C. More complex VNFs

We focused on a simple L2 forwarding VNF as a starting point to illustrate our methodology. More complex VNFs will require additional analysis, especially related to caches and memory patterns. L2 forwarding is one of the simplest VNF, with limited impact on memory-related features. Most of the processing is spent on computation, while no state is necessary. When the VNFs have to access the RAM, this will also have an impact on both the caches and the TLB features. We plan to extend our experimental campaign with complex VNFs that either require to access a large data structure (as for an IPv4 forwarding module that has to perform a longest prefix match lookup on a hash table) or present stateful characteristics (as for a NAT module).

D. Other applications of our methodology

In addition to reducing the number of invasive network measurements, another use-case for our methodology is the resource optimization for high-speed VNFs of unknown requirements. For instance, it is possible to leverage the great amount of available data with lightweight machine learning algorithms that could be deployed within the datapath to quickly react to changes in the application state such as packet loss or misbehavior. We highlight that some tools exist [28] to monitor and plan the resource allocation of VNFs, but they need to be integrated within the tenants' sequence of VNFs, and therefore are opaque to the server owner. With our approach, the server owner would have the possibility to analyze the requirements for a plethora of use-cases and plan the resources accordingly [38]. Moreover, a simple algorithm that periodically analyzes the network fingerprint can, at runtime, adjust the resource allocation accordingly, without any change on the tenants' or the operator's side, or quickly associate special patterns to misbehavior. Additional data about the CPU utilization (not utilized for this manuscript) can be leveraged to estimate the CPU requirements for VNFs and optimize resource usage. Finally, since a global software network application can be considered as the union of multiple VNFs deployed on a COTS server, we plan to investigate how *to also infer the global network behavior* based solely on CPU measurements. As the network is not necessarily changing so abruptly as we engineered for our experiments, it is possible to adopt our approach to predict network changes in the short term, and therefore extending our inference model to a predictive one.

VI. Conclusion and Future work

We propose a novel methodology, based on a "network fingerprint", to infer the behavior of high-level VNF applications by using the low-level CPU features related to the VNF processing instead of using common network measurement tools. In this work, we analyzed a set of potential CPU features linked to the VNF execution and showed how they can be affected by different high-level scenarios. We performed an extensive evaluation campaign to validate the foundations of our approach: variations on the CPU measurements are indeed tightly coupled to the high-level state, and this knowledge could be used as the sole hypothesis to infer the network function's state. We have shown that this technique can be effectively utilized to infer current state conditions without additional monitoring tools. As future work, we plan to extend our inference to analyze both the cases of different VNFs sharing the same core (which can affect the CPU-related measurements) and a VNF distributed across multiple cores. We also plan to analyze the network fingerprint associated with more complex VNFs performing stateful processing (e.g., firewalls, load balancers). Finally, we will tackle the adoption of our technique also as a predictive model, where CPU measurements can give network operators insights about the short-term evolution of their infrastructure and could use such knowledge from planning actions ahead of time within the constraints of high-speed modern cloud environments. Our code and dataset are open-source, and we strongly encourage researchers to use them and perform additional evaluations.

References

[1] ETSI GS NFV-IFA 002 - Network Functions Virtualisation (NFV); Acceleration Technologies; VNF Interfaces Specification, 2017.

[2] Performance Counters for Linux, PCL. https://perf.wiki.kernel.org/index.php/Main_Page (version 4.15.18), 2019.

[3] Open vSwitch. https://www.openvswitch.org/, 2020.

[4] VPP - fd.io. https://wiki.fd.io/view/VPP, 2020.

[5] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE ANCS*, 2015.

[6] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. O'Reilly Media, Inc., 2005.

[7] Benoit Claise, Ganesh Sadasivan, Vamsidhar Valluri, and Martin Djernaes. RFC3954: Cisco Systems NetFlow Services Export Version 9. http://www.ietf.org/rfc/rfc3954.txt, 2004.

[8] Michael Coughlin, Eric Keller, and Eric Wustrow. Trusted click: Overcoming security issues of NFV in the cloud. In *SDN-NFV Security*, 2017.

[9] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. Lightbox: Full-stack protected stateful middlebox at lightning speed. In *ACM CCS*, 2019.

[10] Martin Duggan, Karl Mason, Jim Duggan, Enda Howley, and Enda Barrett. Predicting host cpu utilization in cloud computing using recurrent neural networks. In *International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 67–72. IEEE, 2017.

[11] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *ACM IMC*, 2015.

[12] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module. In *ACM APNet*, 2017.

[13] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware (BESS). 2015.

[14] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[15] Tom Herbert and Willem de Bruijn. Scaling in the Linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2011.

[16] Intel. Data Plane Development Kit. http://dpdk.org.

[17] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *ACM SIGCOMM*, 2016.

[18] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *USENIX NSDI*, 2016.

[19] Stanislav Lange, Leonardo Linguaglossa, Stefan Geissler, Dario Rossi, and Thomas Zinner. Discrete-time modeling of nfv accelerators that exploit batched processing. In *IEEE INFOCOM*, 2019.

[20] Leonardo Linguaglossa, Stanislav Lange, Salvatore Pontarelli, Gabor Rétvári, Dario Rossi, Thomas Zinner, Roberto Bifulco, Michael Jarschel, and Giuseppe Bianchi. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE*, 2019.

[21] Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, and Pierre Pfister. High-speed data plane and network functions virtualization by vectorizing packet processing. *Computer Networks*, 2019.

[22] Enio Marku, Gergely Biczók, and Colin Boyd. Towards protected vnfs for multi-operator service delivery. In *IEEE NetSoft*, 2019.

[23] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *IEEE NFV-SDN*, 2016.

[24] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. *ONF White Paper - http://_pages.cs.wisc.edu/∼agember/cs640/s14/docs/SDNWhitePaper.pdf*, 2012.

[25] Michele Paolino, Nikolay Nikolaev, Jeremy Fanguede, and Daniel Raho. SnabbSwitch: user space virtual switch benchmark and performance optimization for NFV. In *IEEE NFV-SDN*, 2015.

[26] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *USENIX NSDI*, 2018.

[27] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium*, 2012.

[28] Raphael Vicente Rosa, Christian Esteve Rothenberg, and Robert Szabo. VBaaS: VNF benchmark-as-a-service. In *IEEE EWSDN*, 2015.

[29] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[30] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Engineering Bulletin*, 2001.

[31] Rashid Tahir, Muhammad Huzaifa, Anupam Das, Mohammad Ahmad, Carl Gunter, Fareed Zaffar, Matthew Caesar, and Nikita Borisov. Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises. In *Research in Attacks, Intrusions, and Defenses*, 2017.

[32] This url has been obscured due to double blind revision. Dataset repository. https://anonymous.4open.science/r/475690b7-9433-4023-a79c-18cf2573c890/, 2021.

[33] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *ACM SOSR*, 2018.

[34] Vincent M Weaver. Self-monitoring overhead of the linux perf event performance counter interface. In *IEEE ISPASS*, 2015.

[35] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, Luigi Iannone, and James Roberts. Comparing the performance of state-of-the-art software switches for nfv. In *ACM CoNEXT*, pages 68–81, 2019.

[36] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, and Dario Rossi. Flowatcher-dpdk: Lightweight line-rate flow-level monitoring in software. *IEEE TNSM*, 2019.

[37] Tianzhu Zhang, Han Qiu, Leonardo Linguaglossa, Walter Cerroni, and Paolo Giaccone. Nfv platforms: Taxonomy, design choices and future challenges. *IEEE TNSM*, 2020.

[38] Thomas Zinner, Stefan Geissler, Stanislav Lange, Steffen Gebert, Michael Seufert, and Phuoc Tran-Gia. A discrete-time model for optimizing the processing time of virtualized network functions. *Computer Networks*, 2017.