A Black-Box Approach for Estimating Utilization of Polled IO Network Functions

Harshit Gupta
Georgia Institute of Technology

Abhigyan Sharma, Alex Zelezniak, Minsung Jang
AT&T Labs Research

Abstract

Cloud managment tasks such as performance diagnosis, workload placement, and power management depend critically on estimating the utilization of an application. But, it is challenging to measure actual utilization for polled IO network functions (NFs) without code instrumentation. We ask if CPU events (e.g., data cache misses) measured using hardware performance counters are good at estimating utilization for unmodified NFs. Our measurement shows a strong correlation between several CPU events and NF utilization for three types of network functions. Inspired by this finding, we explore the possibility of computing a universal estimation function that maps selected performance counters to CPU utilization estimates for a wide-range of NFs, traffic profiles and traffic loads.

1 Introduction

Network functions (NFs) such as routers, firewalls, and proxies use polled network IO in userspace for high performance [5, 7, 11]. Like any application, they require common cloud management tasks such as performance debugging [9], workload placement [8] and power management [10]. These tasks often depend on *NF utilization*: the percentage of the peak traffic supported by the NF that the traffic being processed by an NF represents, other factors such as hardware and traffic patterns remaining unchanged. But, due to polling, these NFs always report 100% CPU core utilization, even when they are not processing any packets! There is no server-level techniques to estimate utilization of black-box NFs. The only approach is to instrument NF code to report cycles spent in packet processing and those spent in empty polls [16].

We propose an approach that requires no modifications to NFs or their network drivers. Our idea is to use CPU events, e.g., branch misses, or L3 cache misses, collected using hardware performance counters [1]. CPU events have ubiquitous support in server CPUs and software utilities [3, 17]. Given a limited number of hardware counters in a CPU, the challenge is to choose which CPU events to monitor and how to map the events' counter values to NF utilization values. At the outset, there are two extreme possibilities for this line of research. It is possible that one finds a single CPU event (out of hundreds of events in a modern CPU [4]) whose counter values can be directly mapped to NF utilization for all NFs for all traffic profiles. The other extreme would be if no combinations of CPU events are found to accurately estimate the utilization of any NF for common traffic scenarios. The reality, as we show in this paper, lies somewhere in the middle.

We conduct an empirical study with three NFs from different layers of the stack: a layer-3 forwarder [6], a layer-4 load balancer [13], and a layer-7 network intrusion detector (Snort [14]). We subject these NFs to a range of workloads with varying packet sizes, connection lengths and traffic volumes. We collect data on counter values for all CPU events in each experiment. Use a subset of the collected data to train a variety of *estimator functions* or *estimators* that map counter values for a small number of CPU events to NF utilization. Using cross validation, we evaluate the accuracy of estimation functions.

We find a strong prediction accuracy in excess of **90**% between several *local* estimation functions that span an NF and a subset or even all workloads. Our results indicate that an NF's behavior can be accurately modeled using one of at most a few of estimation functions. We find that multiple estimators needed upon dramatic changes in the workload such as from short flows with 10 packets to long flows with 1000 packets. Our efforts to build a simple *universal* estimator for all NFs based on one or two CPU events yields a poor accuracy of **70**% at most. Our findings preclude the possibility of a simple universal estimator.

We conclude with a discussion on designing improved universal estimators based on hierarchical estimation functions, expanding the scope of our empirical study such as by including virtualized network functions, and addressing additional use cases such as workload management and performance diagnosis.

2 Vision and approach

2.1 Vision: a "`top`" for polled IO NFs

We provide background on DPDK – a popular framework for building polled IO NFs, discuss prior work on provide background on instrumentation-based techniques.

Anatomy of a DPDK application: DPDK [5] is a toolkit for developing network functions by bypassing the Linux kernel in order to attain efficiency. Traditionally, network applications rely on interrupts to process received packets, thus introducing additional latency and context switching overheads for packet processing [12]. But, DPDK applications actively poll the Rx descriptors of the NIC to determine whether there are packets to process. If the NIC had received packets before the poll, the Rx descriptors would hold the locations in memory where those packets are stored. Packets would have been transferred to a buffer in userspace through DMA. These packets are then delivered to the application in bursts, which processes them and then transmits them. Transmit path contains similar steps as the Rx path, involving an update of the Tx descriptors by the DPDK application, followed by the NIC DMA'ing those packets for transmission.

Instrumentation-based techniques: The poll mode driver results in 100% utilization reported for the cores it is running on. Hence, prior efforts have used application or driver instrumentation to assess actual work being done by a DPDK-based NF. Trifonov et al. [16] measure the number of empty polls to determine utilization and insert artificial delays between consecutive polls to reduce polling frequency. Niccolini et al. [10] use NIC driver instrumentation to estimate queue lengths of NIC packet buffers. Queue lengths are used as a proxy for utilization in performing CPU power management. However, these approaches require code or driver instrumentation and hence are inapplicable for general, unmodified NFs.

Goal: Our position is that getting a good utilization estimate for unmodified NFs can vastly simplify cloud management tasks such as performance diagnosis, workload management and power management for them. Towards this goal, we seek to create a utility like UNIX's `top` for polled IO NFs. Similar to `top`, this utility would list the polled IO NFs and their utilization on each of the cores they are running on.

2.2 Approach: NF utilization via CPU events

CPU events: Our intuition is that a polled IO NF that is processing packets is likely to generate a different type of activity on the CPU, e.g., data cache access due to arriving packets or branch prediction errors in the polling thread due to non-empty polls. A modern CPU defines hundreds of these events (e.g., 714 CPU events on an Intel Xeon E5-2680 v3 in our testbed), which can give a detailed picture of CPU activity. This motivates us to explore if there are some CPU events whose frequency is strongly correlated to the actual utilization of the NF.

Hardware performance counters: CPU events can be collected using CPU registers called *hardware performance counters*. Such counters have long been available in server CPUs such as Intel and AMD [2, 15]. Support for using them is available in various utilities such as Linux perf [17] and Intel EMON [3]. While there are hundreds of CPU events, a CPU only has a small number (e.g. 4 in Xeon E5-2680) of hardware counters. Since a register can be programmed to collect only one event at a time, only a small number of events can be counted concurrently.

Advantages of CPU events: If CPU events can accurately estimate NF utilization, it would be a black-box approach requiring no modification to NFs. It would be particularly useful to large networks such as ISP networks, which run dozens of NFs [**?**] and NICs from several vendors and would require considerable effort for instrumenting NFs to report utilization in a uniform manner. Further, hardware counters impose **negligible performance overhead. (how ?)** on NFs and hence can be used by operators on NFs running in production.

Open questions: While CPU events have been used for getting performance insights (e.g., L1/L2/L3 cache misses) for general applications [15], little is known if they are useful for estimating utilization for polled NFs. For example, (1) can we select a small number of CPU events whose values can be mapped to a NF utilization level? (2) If the mapping of CPU events and counter values to the utilization of an NF robust across variation in traffic profile for that NF? (3) Is there an universal estimation function that can map a fixed set of CPU events and their counter values to utilization for all NFs? (4) Finally, does this approach improve efficiency of cloud management tasks?

3 An empirical study of estimation functions

An *estimation function* (or an *estimator*) on a set of CPU events accepts as input the counter values for those events measured at a CPU core and outputs a percentage value from 0 to 100 that represents the utilization for the NF running on that core. In this section, we formally define the types of estimation functions we evaluate, describe the experimental setup to collect data for our evaluation and present statistical metrics on the accuracy of our estimation functions. We conclude with a discussion on feasibility of an universal estimator.

3.1 Local and universal estimators

We begin by defining some key assumptions under which we evaluate estimation functions. Since CPU events are dependent on the CPU architecture and its configuration, we assume that the CPU used for evaluation as well as its BIOS settings remain unchanged. In practice, the estimators can

be updated upon generational changes in hardware or major configuration changes. An NF can run on one more cores on a physical server. However, each core is assumed to perform identical packet processing so that an estimator for a single core can easily generalize to a multi-core NF. Henceforth, we assess the accuracy for single-core NFs. To assess the NF throughput corresponding to a 100% utilization, our evaluation assumes that the CPU is the bottleneck resource for the NF.

We consider a set of NFs (N). An NF $n \in N$ is evaluated for a set of test profiles T_n. The test profile includes traffic characteristics such as packet sizes and flow lengths as well as the NF's internal configuration such as routing table sizes for a router or the size of memory cache for a proxy. For each traffic profile, the NF is subject to any load (L) in the range 0-100, where a load of 0 means no traffic and a load of 100 means a traffic volume equal to the maximum throughput of the NF for that test profile. Let C_ALL represent the set of all CPU events that can be counted.

Local estimators: A local estimator outputs utilization for an NF and a subset of traffic profiles of that NF. Let $n \in N$ and $T'_n \subseteq T_n$. For events $C \in C\_ALL$, we define a local estimator function $L_{nT_nC}()$ that takes as inputs the counter values of CPU events C and outputs the percentage utilization. The simplest local estimators output utilization for a single test profile for an NF while the most general local estimators output utilization for all test profiles for an NF.

Universal estimators: A universal estimator outputs utilization for all NFs $n \in N$ and their respective test profiles T_n. For events $C \in C\_ALL$, we define an universal estimator function $U_C()$ that takes as inputs the counter values of CPU events C and outputs the percentage utilization for any NF.

Evaluating estimator performance: To evaluate a local or a global estimator's performance, we define a test suite TS consisting of multiple test cases denoted by the tuple (n,t,u_0). A test case for an NF $n \in N$ for a traffic profile $t \in T_n$ sends traffic at a rate so as to generate a utilization u_0 for the NF on a core. After running a test case in an experiment, we record the NF utilization u reported by the estimator for that experiment. The estimator performance is evaluated solely based on tuples (u_0,u) from a test suite.

3.2 Experimental methodology

3.2.1 Network functions

We have evaluated three NFs with varying amounts of computation and state.

A layer 3 forwarder (**L3FWD** for short) matches the destination IP of incoming packet to the entries stored in its forwarding table, and updates the layer 2 header before sending the packet to the next-hop router. The forwarder is a stateless NF because it does not maintain per-flow state. Our experiments are conducted with the L3FWD application included in the DPDK **18.05** release.

A layer 4 load balancer (**L4LB** for short) multiplexes connections to a pool of servers. In order to cope with a changing pool of servers, it creates a flow table entry upon the arrival of the first packet of a new connection. The flow table entry stores a connection identifier as the key – typically a 5-tuple of source IP, destination IP, source port, destination port and protocol – and the chosen server as the value. We evaluate a homegrown load balancer implemented as a DPDK application using nearly 3K lines of code. It performs common flow table management tasks such as flow table resizing and deletion of flow table entries.

An intrusion detection system (IDS for short) identifies traffic matching a set of pre-configured rules. The rules can specify layer-3 and layer-4 header fields, application-layer rules, e.g., URL patterns, and strings in the packet data. It analyzes a connection to reconstruct its bytestream, which help examine an application across its packets. Thus, an IDS not only maintains per-flow state, it also performs non-trivial computation on each flow. We evaluate Snort (release **2.9**) as an IDS and use a DPDK-patch that enables Snort's data acquisition module to send and receive packets using network interfaces bound to a DPDK driver.

3.2.2 Test profiles

Our test profiles for NFs depend on their respective performance metrics. Our test profiles are generated using an open-source DPDK traffic generator T-Rex. L3FWD, L4LB and IDS respectively use T-Rex's stateless, stateful and advanced stateful modes. Stateful mode can emulate continuous arrival and departure of millions of connections, where as the advanced stateful mode implements a full-featured TCP stack with

A standard metric for L3FWD's performance is its throughput in packets per second. We stress the computation done by L3FWD by evaluating traffic profiles with minimal-sized packets (64B) as well as larger packet sizes (128B, 256B, 512B). L3FWD is configured to apply **longest-prefix match** algorithm on routing table is configured with 10K entries.

L4LB should support a high rate of packets and a high rate of connections. Our evaluation test profiles with short connections with 10 packets as well as longer connections up to 1000 packets per connection. All packets in a flow are minimum-sized to test the packet throughput. L4LB is configured to use a 10 M flow table which is sufficient to store all flow entries.

Test profile for IDS.

3.2.3 Testbed configuration

Server. NIC port.

CPU configuration. bios settings.

Software: cpu programming. length of each measurement.

3.3 Results

3.3.1 Local estimators

Conclusion: several highly correlated counters.

3.3.2 A simple universal estimator

Conclusion: lower accuracy with simple model, need more sophisticated models.

4 Evaluations

4.1 Cross-profile utilization estimator

4.2 Stateless NF - L3 forwarding

4.2.1 Generating training data set

The generation of training data is broken down into a set of phases, each one described below

1. Determine peak-load : For each traffic profile, we determine the peak load that the NF can sustain [1]. For each traffic profile, we increase the traffic rate (packets per second) until the packet drop rate becomes higher than 1%. We denote this limiting traffic rate as the peak load that the NF can serve for that specific traffic profile.

2. Record event counts for varying loads : For each profile, we subject the NF to $20\%, 40\%, \cdots$ of the peak traffic rate and record the counts of each event exposed by the CPU architecture over a period of 1 second [2]. The data on event counts for various utilizations of the NF with different traffic profiles is then examined to find out-of-core indicators of NF utilization.

For this experiment we use the following traffic profiles to test against L3-forwarder network function. The choice of packets profiles is such that upon increasing the traffic rate, the CPU processing becomes a bottleneck faster than network congestion.

- Pkt size = 64
- Pkt size = 90
- Pkt size = 150
- Pkt size = 278

The NF is assigned a single core for processing packets that operates at a frequency of 1.2GHz. We choose the lowest frequency, as it would maximize the stress on CPU packet processing.

[1] Note that we assume that the bottleneck lies in the packet processing done on the CPU rather than the network.

[2] Since the number of hardware performance counters is limited, counting all the events takes significantly longer time

4.3 Picking the right counters

Upon collecting system event counts, we need to pick a subset of events that have the following characteristics

1. V1 : For a given utilization (% of peak load) event counts are consistent (low coefficient of variation) across traffic profiles.

2. V2 : For a given traffic profile, event counts are highly correlated (absolute correlation coefficient more than 0.8) with traffic rate

Upon filtering the entire monitored event count data, we arrive at the following events that satisfy both the aforementioned characteristics.

- $UOPS\_EXECUTED.CORE_CYCLES\_GE\_2$
- $UOPS\_EXECUTED\_PORT.PORT\_6\_CORE$
- $IDQ.MITE\_ALL\_UOPS$
- $UOPS\_EXECUTED.CYCLES\_GE\_2\_UOPS\_EXEC$
- $UOPS\_EXECUTED\_PORT.PORT\_6$
- $BR\_INST\_EXEC.NONTAKEN\_CONDITIONAL$

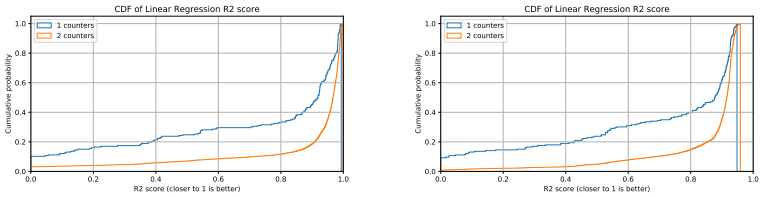
5 Related work

6 Discussion

Network cloud management and automation requires a common interface to report utilization of NFs. The central question before the workshop audience is whether our effort to develop a CPU event-based universal estimator worthwhile or is it easier to define a common API and have NF vendors implement and maintain that API for their respective NFs.

Improved universal estimators: A key factor is whether a CPU event-based estimator can achieve an accuracy comparable to NF instrumentation. We have found that CPU events are not a "silver bullet" to the complex problem of estimating utilization of polled NFs. Simple universal estimators based on one or two CPU functions do not achieve low error rates for them to be useful. However, simple local estimators do estimate utilization within a percentage point. A logical next step is to explore universal estimators based on few tens of CPU events, that can be still be collected in few milliseconds. Such an estimator seems practical for scenarios that do not need sub-millisecond response time.

Virtualized and/or cross-core processing: We are exploring two more kinds of NFs in ongoing work: virtualized NFs and NFs that perform cross-core processing. Virtualized polled IO NFs using network interfaces that support SR-IOV (a form of virtual PCI device) can perform close to an NF that has direct physical access to network interfaces. We are



(a) L3 router (b) L4 load-balancer

Figure 1: Distribution of estimator score (coefficient of determination) for tested NFs.

studying whether estimators need to be rebuilt if an NF is run in a VM on a SR-IOV interface. Many multi-core NFs dedicate one or more threads to poll hardware queues for Rx and multiplex them among worker threads via software queues. Worker threads poll these software queues to read packets. Unlike NFs we have evaluated, these NFs may need different estimator functions for Rx threads and worker threads. Determining whether Rx or worker thread is the bottleneck is a challenge for such NFs. We would welcome feedback on other kinds of NFs we can include in our study.

Use cases: We see a number of applications of CPU event-based estimators. A top-like utility for NFs can be made available for system administrators. With community help, the utility can be pre-packaged with a set of estimator functions for common CPU types, so that a tool can be used out of the box. For cloud orchestration tasks, the utility would implement a northbound API. Cloud orchestration software (e.g., OpenStack) can query that API to optimize resource allocation (number of cores), workload placement and load distribution. Power management using CPU frequency scaling and/or sleep states seems an urgent need for polled IO NFs that are always running at 100% CPU utilization. A key question for frequency scaling is whether estimators trained at a given CPU frequency work at other frequencies using simple interpolation, e.g., 1000 events at 1GHz is equivalent to to 2000 events at 2GHz. Exploring the trade-off between power management and performance (latency, jitter and packet loss) is an interesting area of exploration.

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, May 1997.

[2] AMD Corporation. Instruction-Based Sampling : A new performance analysis technique for amd family 10h processors. http://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf. Accessed: 2018-07-20.

[3] Intel Corporation. EMON: User's guide. https://software.intel.com/en-us/download/emon-users-guide. Accessed: 2018-07-20.

[4] Intel Corporation. Intel Processor Events Reference. https://download.01.org/perfmon/index/. Accessed: 2019-03-04.

[5] DPDK. DPDK data plane development kit. https://www.dpdk.org. Accessed: 2018-06-06.

[6] DPDK. DPDK L3 Forwarding Sample Application. https://doc.dpdk.org/guides/sample_app_ug/l3_forward.html. Accessed: 2019-03-04.

[7] S Gallenmüller. Comparison of memory mapping techniques for high-speed packet processing. *Technical University of Munich*, 2014.

[8] Gueyoung Jung, Matti A Hiltunen, Kaustubh R Joshi, Richard D Schlichting, and Calton Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 62–73. IEEE, 2010.

[9] Kai Li, Ming Shen, and Chuanpeng Zhong. I/o system performance debugging using model-driven anomaly characterization. *FAST*, 2005.

[10] Luca Niccolini, Gianluca Iannaccone, Sylvia Ratnasamy, Jaideep Chandrashekar, and Luigi Rizzo. Building a power-proportional software router. In *Presented*

as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12), pages 89–100, 2012.

[11] NTOP. PF_RING. https://github.com/ntop/PF_RING. Accessed: 2019-03-04.

[12] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In *OSDI*, pages 203–216, 2016.

[13] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *ACM SIGCOMM Computer Communication Review*, pages 207–218. ACM, 2013.

[14] Snort. Snort - network intrusion detection and prevention system. https://www.snort.org. Accessed: 2019-03-04.

[15] Patrick Fay Thomas Willhalm, Roman Dementiev. Intel®Performance Counter Monitor : A better way to measure cpu utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor. Accessed: 2018-07-20.

[16] Hristo Georgiev Trifonov. Traffic-aware adaptive polling mechanism for high performance packet processing. *University of Limerick Institutional Repository*, 2017.

[17] Linux Perf Wiki. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2018-07-20.