Is it a SmartNIC or a Key-Value Store? Both!

Giuseppe Siracusano
University of Rome, Tor Vergata

Roberto Bifulco
NEC Laboratories Europe

CCS CONCEPTS

• **Networks** → *Programming interfaces*; **In-network processing**;

KEYWORDS

SmartNic; eBPF; XDP; Key-Value Store.

ACM Reference format:
Giuseppe Siracusano and Roberto Bifulco. 2017. Is it a SmartNIC or a Key-Value Store? Both!. In *Proceedings of SIGCOMM Posters and Demos '17, Los Angeles, CA, USA, August 22–24, 2017,* 3 pages.
https://doi.org/10.1145/3123878.3123928

1 INTRODUCTION

In-memory Key-Value stores (KVSs) are important components of modern web services. Companies such as Facebook, Twitter, Amazon, etc. deploy such systems at scale, in their datacenters, to improve services' quality and scalability.

Given the critical role they play, the performance of KVSs has been the focus of several works in the last few years. On the one hand, KVSs have been optimized to take advantage of modern server's hardware [6]. This usually included exploiting techniques such as Kernel by-pass, to reduce the overhead of network processing in software, and a number of careful design decisions to take advantage of multi-core architectures and their cache memories [7]. On the other hand, purpose-built accelerators have been proposed in an effort to further scale throughput and reduce response time [3].

In this work, we combine the two approaches in a single implementation that runs both on general purpose servers and on SmartNICs. A SmartNIC couples a *hardware accelerator*, such as FPGAs or Network Processing Units (NPUs), together with a Network Iterface Card (NIC), in order to help scaling network workloads beyond 40Gbps. Unfortunately, they are generally tailored for network processing, making them a hard fit for different types of workloads.

Our contribution is the design of NICached, a general caching system for KVSs, which can be supported by upstream Linux Kernels and amenable to be offloaded to different types of SmartNICs. NICached can handle KVS requests that use a connection-less transport protocol (i.e., we do not currently support TCP), achieving a 6x improvement in terms of requests per second (RPS) over a production ready implementation of memcached, a popular KVS. Contrary to related work, NICached does not modify the Linux kernel, nor the KVS, and it can take advantage of SmartNICs' accelerators without requiring dedicated hardware KVS implementations.

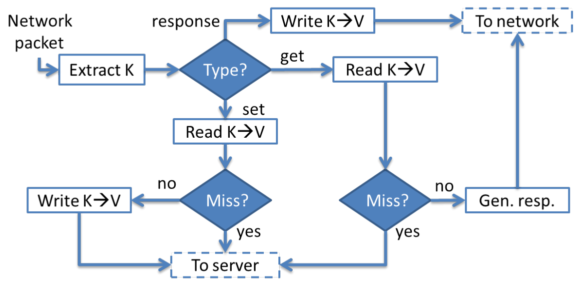


Figure 1: NICached operations

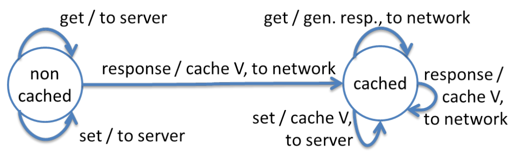

get / to server
get / gen. resp., to network
non cached
response / cache V, to network
cached
response / cache V, to network
set / to server
set / cache V, to server

Figure 2: NICached Finite State Machine. The labels on the edges show the transitions' [event] / [actions].

2 NICACHED

Working as a cache, the purpose of NICached is to store the most recently requested Key/Value (K/V) entries, in order to respond on behalf of the KVS for *get* requests. To offload as much processing as possible from the KVS, NICached sits at the earliest packet processing point of a modern server's network processing stack. That is, it is logically deployed at the NIC level. As such, NICached can intercept a network packet containing a *get* request before it is delivered to the server's network stack. Likewise, NICached can intercept response messages coming from the KVS in order to cache at the NIC level a K/V entry.

NICached implements the simple algorithm represented by the flow chart of Fig. 1. When a network packet containing a KVS message enters the system, the contained K is extracted, then, depending on the message's type, different actions are taken. If the packet is a *get*, a lookup in the cache (e.g., a hash table) retrieves the corresponding K/V entry to immediately generate a response message in reply, without delivering the packet to the server. If the lookup could not retrieve any entry, i.e., in presence of a cache miss, the message is instead delivered to the server so that the KVS can process it. In such a case, the KVS will eventually provide a response containing the K/V entry. NICached reads this response to update the cache with such entry, before delivering the message to the network. If the message is a *set*, NICached still checks the local cache. In case of a miss, it simply delivers the message to the server. In fact, a miss shows that the corresponding K was not recently requested. On the contrary, if there is a K/V entry, then the cached value should be updated with the newly provided one. In effect, the cached V is outdated by the new *set* request, which contains a new value, say V'. For correctness of the system, V must be replaced with V' during the processing of the *set*, otherwise any following *get* for such value would be answered by NICached with the outdated V.

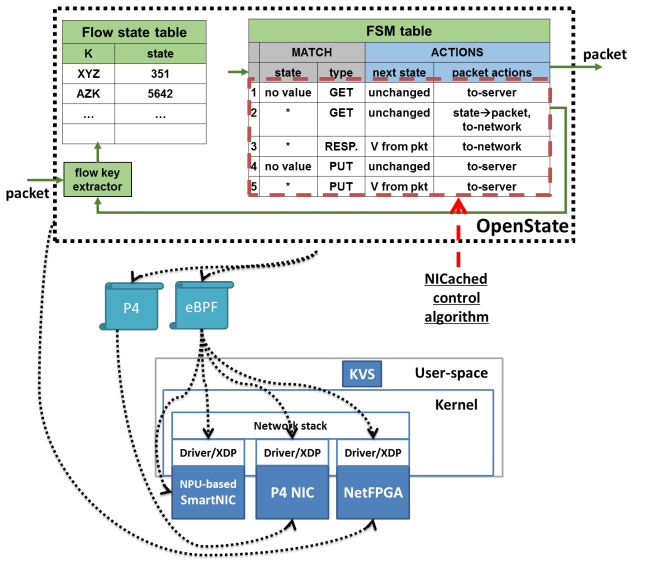
Implementation The implementation of NICached builds on a simple observation: the algorithm of Fig. 1 can be easily expressed using a Finite State Machine (FSM), as shown in Fig. 2. Since recent proposals in the area of programmable network data planes suggest an FSM abstraction as programming model [2, 8], we explored the possibility to implement NICached using a similar approach. In particular, we describe the OpenState [2] data plane using eBPF [10](extended Berkeley Packet Filter). eBPF is a Linux's technology that allows a programmer to specify limited but safe network processing directives that can be injected directly in the kernel. Also, recently the eXpress Data Path (XDP) *hook* has been introduced [11]. XDP is an interface supported by NIC vendors that allows a programmer to inject eBPF programs at lower points in the network stack, e.g., directly in the NIC's driver. Therefore, we use XDP as execution point for the OpenState-eBPF program.

One important implementation detail is the realization of the cache for K/V entries. Briefly, in OpenState a hash-table stores the state associated with a particular network flow. A programmer specifies a definition for a network flow by setting a *lookup-key extractor*. The extractor combines packet header's fields in a hash used as key for the lookup in the hash-table. We configure the extractor to use the K contained in the KVS messages to lookup in the hash-table. Consequently, we store Vs in the hash-table as if they were flow states.

Software micro-benchmarks We tested NICached on a testbed composed by two machines with Intel Xeon E5-1630 CPUs (4 cores @3.70GHz), connected back-to-back with two Mellanox ConnectX-3 (40Gbps) Ethernet cards. The NICached eBPF program uses the *XDP_TX* action to transmit packets. Using the XDP benchmark tool [4], we measured the baseline performance of our NIC, when using XDP_TX to forward packets. In our tests we achieved 6.9 million packets per second (Mpps) with a single CPU's core.

Having established our baseline, we performed a preliminary evaluation of our implementation comparing it with a production-ready memcached deployment on the same server. In particular, we compared the throughput in terms of RPS, using minimum sized K/V entries. That is, we used a cache composed of 3.2 million of entries, with key and value of 8 bytes. We chose to use small entries because they represent the critical workload for KVS [1, 6], and they ensure that the available network bandwidth is not the bottleneck. Notice that in such setting, handling a memcached request corresponds to the forwarding of a single network packet.

Under these conditions, and using the CPU's four cores, memcached is able to handle 0.9MRPS. In contrast, our NICached prototype can process 5.4MRPS using just a single core. The 6x improvement factor is mainly given by our ability to avoid most of the operating system's overheads.



Flow state table	
K	state
XYZ	351
AZK	5642
...	...

FSM table				
	MATCH		ACTIONS	
	state	type	next state	packet actions
1	no value	GET	unchanged	to-server
2	*	GET	unchanged	state→packet, to-network
3	*	RESP.	V from pkt	to-network
4	no value	PUT	unchanged	to-server
5	*	PUT	V from pkt	to-server

Figure 3: NICached deployments options: OpenState "native" on FPGA; eBPF, for Linux or NPU-based SmartNIC; P4 for P4-based NICs.

SmartNICs While the testing of our implementation on actual SmartNICs is part of our future work, we are confident that NICached can be already deployed on SmartNICs.

We motivate our statement as follows. First, we used OpenState to describe NICached operations, and OpenState has implementations available for the NetFPGA, a SmartNIC that uses an FPGA as hardware accelerator. Second, OpenState can be also described with P4, making it potentially suitable for implementation on top of other P4 targets [9]. Finally, we described NICached and OpenState using eBPF. Several SmartNICs based on NPUs already (or plan to) support eBPF programs [5]. These options are summarized in Fig. 3

In conclusion, NICached provides an example of transparent acceleration for KVSs, using a single implementation that can run both in the Linux kernel and, potentially, on a heterogeneous set of SmartNICs. While the actual potential of this approach has yet to be demonstrated, with several issues to be tackled (e.g., cache size and consistency), we believe this is a promising area of research that could help in exploiting the heterogenous hardware resources of future servers.

3 ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671648 ("VirtuWind"). This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Berk Atikoglu et al. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Eval. Rev.*
[2] Giuseppe Bianchi et al. 2014. OpenState: programming platform- independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review* (2014).

[3] Michaela Blott et al. 2013. Achieving 10Gbps Line-rate Key-value Stores with FPGAs.. In *HotCloud*.

[4] Jesper Dangaard Brouer. 2017. XDP benchmark. https://github.com/netoptimizer/prototype-kernel/. (2017).

[5] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Offload to Hardware: cls_bpf and XDP. *NetDev 1.2* (2016).

[6] Sheng Li et al. 2015. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM SIGARCH Computer Architecture News*. ACM.

[7] Hyeontaek Lim et al. 2014. MICA: A holistic approach to fast in-memory key-value storage. *management* 15, 32 (2014), 36.

[8] Masoud Moshref et al. 2014. Flow-level state transition as a new switch primitive for SDN. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM.

[9] OpenstateP4. 2015. http://github.com/OpenState-SDN/openstate.p4. (2015).

[10] Alexei Starovoitov. 2014. eBPF. https://www.kernel.org/doc/Documentation/networking/filter.txt. (2014).

[11] Alexei Starovoitov and Tom Herbert. 2016. eXpress Data Path. https://github.com/iovisor/bpf-docs/blob/master/Express$_D$ata$_P$ath.pdf. (2016).