Partition-Aware Packet Steering Using XDP and eBPF for Improving Application-Level Parallelism

Pekka Enberg
University of Helsinki

Ashwin Rao
University of Helsinki

Sasu Tarkoma
University of Helsinki

ABSTRACT

A single CPU core is not fast enough to process packets arriving from the network on commodity NICs. Applications are therefore turning to application-level partitioning and NIC offload to exploit parallelism on multicore systems and relieve the CPU. Although NIC offload techniques are not new, programmable NICs have emerged as a way for custom packet processing offload. However, it is not clear what parts of the application should be offloaded to a programmable NIC for improving parallelism.

We propose an approach that combines application-level partitioning and packet steering with a programmable NIC. Applications partition data in DRAM between CPU cores, and steer requests to the correct core by parsing L7 packet headers on a programmable NIC. This approach improves request-level parallelism but keeps the partitioning scheme transparent to clients. We believe this approach can reduce latency and improve throughput because it utilizes multicore systems efficiently, and applications can improve partitioning scheme without impacting clients.

CCS CONCEPTS

• **Software and its engineering → Operating systems**; **Communications management**; *Multiprocessing / multiprogramming / multitasking*.

KEYWORDS

XDP, eBPF, Packet Steering, Parallelism, Partioning

ACM Reference Format:
Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 2019. Partition-Aware Packet Steering Using XDP and eBPF for Improving Application-Level Parallelism. In *1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (ENCP '19), December 9, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3359993.3366766

1 INTRODUCTION

A single CPU core is not fast enough to serve packets arriving at line rate. For example, the arrival rate of packets on a 40 Gbps NIC is faster than the rate at which a single CPU core can access its last-level cache (LLC), and this difference in operating speeds can prevent the CPU from keeping up with the network [15]. The performance gap is further expected to increase with 400 Gbps and beyond on the horizon. Fundamentally, the time budget to process a single packet is shrinking radically, forcing applications to embrace parallel processing and NIC offload capabilities.

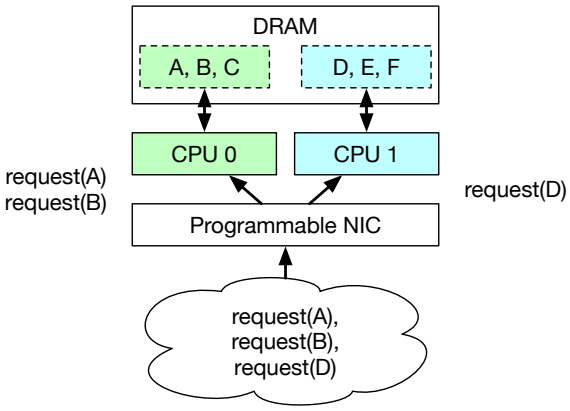


Figure 1: Partition-aware packet steering on a programmable NIC. *The application partitions its resources and the data in DRAM between CPU cores, and a programmable NIC steers requests to the target CPU core by inspecting protocol headers. This allows request processing to run independently on each CPU, while keeping partitioning transparent to client. For the example key-value store shown in this figure, the NIC parses keys from client requests.*

Application-level partitioning is one approach to parallelize request processing on multicore systems [23, 35, 37]. In the thread-per-core model, applications run only one OS thread per CPU core and also partition the data in DRAM between the cores [7, 35]. This enables the CPU cores to run independently by eliminating synchronization for data access and avoiding OS-level locking. However, steering requests to the CPU core that manages request data either requires clients to specify the partition [14, 23] or uses CPU cycles for the steering [6, 35].

Offloading the network processing to the NIC helps conserve CPU cycles [5, 27], and NICs are also starting to support the ability to run arbitrary programs and customize the offload. These *programmable NICs* come in different flavors, ASIC-based, FPGAs, special-purpose cores (e.g. NPU), or multicore system-on-chips (SoC) [8, 31, 36], but their objective is the same: provide programmable packet processing on the NIC before packets are forwarded down to the OS network stack. Programmable NICs have a significant performance advantage over host CPU for packet processing because they can be highly specialized and do not have to wait for packet data to DMA over the I/O bus to DRAM. However, the emergence of programmable NICs raises a question: *What should applications offload to a programmable NIC for improving parallelism?*

ENCP '19, December 9, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-7000-4/19/12…$15.00
https://doi.org/10.1145/3359993.3366766

We propose an approach for improving parallelism that combines application-level partitioning and packet steering with a programmable NIC (§3). As shown in Figure 1, the application uses a thread-per-core approach in which data in the DRAM is partitioned between threads that are pinned to CPU cores. In one of our previous works, we highlighted that packet steering is a per-request overhead in the thread-per-core model, and this overhead could be address with the help of a programmable NIC [7]. A programmable NIC runs a program that parses application-specific protocol headers, including L7 headers, to steer the packet to the thread responsible for serving it. For example, for a key-value (KV) store such as Memcached, a program running on the NIC parses the Memcached protocol headers to determine a request key and forwards the packet to a CPU core that manages that keys (§3.3). We propose to implement our approach using Linux's Express Data Path (XDP) interface, which is available on a Linux. XDP combines programmable packet processor with kernel-bypass [10], and supports offload to a programmable NIC using eBPF [18]. We present an overview of the XDP interface and eBPF in §2.
Our contributions are as follows.

- We propose a NIC-CPU co-design using eBPF and XDP on Linux, where the NIC performs packet steering and CPU executes application logic (§3). As an example application, we describe the design and implementation of a key-value store using eBPF and XDP. We believe this approach provides a practical solution for accelerating network-intensive applications.
- We discuss the limitations and future research directions of our proposed NIC-CPU co-design approach in §4. Specifically, we analyze a) how applications beyond key-value stores can take advantage of this approach, b) how NIC-based packet steering can improve application-level partitioning, and c) what are the limitations of eBPF and XDP for NIC offload.

Previous approaches to application-level partitioning either require the clients to be aware of the partition scheme, or require expensive inter-thread communication. MICA [23] and HERD [14] expose application partitioning scheme to clients, which makes it challenging to improve partitioning without impacting the clients. Minos implements a size-aware partitioning scheme that is transparent to clients, but it requires inter-thread communication over a software queue [6]. Similarly, the Seastar framework steers requests in user space [35] but requires expensive CPU-intensive polling to avoid thread wakeups [19]. Our proposed approach keeps the partitioning scheme transparent similar to software steering but maintains low request steering overhead, similar to hardware steering. It complements Floem [30], a data-flow programming language for NIC-CPU co-design, which can be used for steering packets in sharded applications. Furthermore, our approach to offloading only packet steering to a programmable NIC is easier to adopt for general purpose applications than previous approaches that offload whole applications to a programmable NIC using a special-purpose programming language such as OpenCL [20].

2 BACKGROUND

We believe that *combining application-level partitioning with packet steering can improve request-level parallelism.* To motivate our approach, we begin by discussing why parallel processing and NIC offload are critical. We then highlight that kernel-bypass networking is a key enabler for this approach. Finally, we give an overview of the Express Data Path (XDP) networking interface [10] and the extended Berkeley Packet Filter (eBPF) virtual machine [18], which make our proposed application-level partitioning with a packet steering approach practical on Linux.

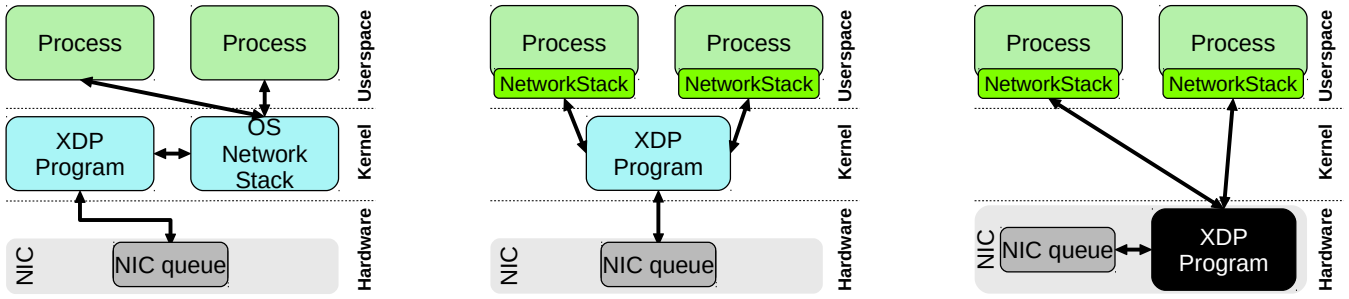
System	Partitioning	Steering	Hardware
Seastar [35]	CPU core	HW/SW	RSS
MICA [23]	CPU core	HW	Flow Director
HERD [14]	OS process	HW	RDMA
Minos [6]	Size-aware	HW/SW	RSS and Flow Director
KV-Direct [20]	None	HW	FPGA
NetCache [13]	Server	HW	ASIC

Table 1: Partitioning and packet steering implementations. *The data is partitioned either per-CPU core, or per OS process, or is based on the size of requested items, or per server. The steering of requests is done either using a combination of hardware software co-design, or solely in the hardware.*

Parallel processing. Applications must embrace parallel processing because single-threaded CPU core speeds have stagnated [9, 34], but NIC speeds are getting faster [15]. To perform parallel request processing, applications use OS threads, but they have overheads from synchronization and context switching. OS system calls can block an OS thread, which is why applications need to create more threads than CPU cores. However, having a large number of threads incurs high overheads because of context switching costs and memory footprint. To address these overheads, applications are increasingly leveraging application-level partitioning [14, 23, 35].

CPU and NIC offload co-design. State-of-art systems, summarized in Table 1, use partitioning, packet steering, and NIC offload for high performance, but their approaches differ from each other. KV-Direct [20] and NetCache [13] offload the whole application to hardware for high performance. However, systems that want to use the CPU for application logic have to use simple partitioning schemes for hardware steering. For example, MICA [23] and HERD [14] partition by CPU core and by OS process, and use hardware steering provided by commodity multi-queue NIC or RDMA. Seastar [35] and Minos [6] use a combination of hardware and software steering; they either support commodity POSIX APIs or provide more advanced partitioning schemes. *There is a gap in systems that want to combine CPU use with NIC offload while enabling advanced application-level partitioning, which our proposed approach aims to fill.*

Kernel-bypass networking. Traditional in-kernel network stacks are designed for flexibility, but are a bottleneck for network-intensive applications for two reasons: (1) they perform too much work per packet, and (2) their system call interface is too expensive [12, 16, 33, 38]. Traditional network stacks require memory allocation and locking per packet, which is too heavy-weight for packet processing time budgets of current NICs. Applications receive and transmit data using the POSIX sockets API, which has high overheads from system call costs and copying. Kernel-bypass networking has emerged as a solution to eliminate these overheads [12, 24, 33]. With kernel-bypass networking, the OS is eliminated from data



(a) XDP via POSIX sockets (b) XDP via kernel-bypass interface (c) XDP via NIC offload

Figure 2: XDP and eBPF configurations. *Applications can use XDP and eBPF via (a) POSIX sockets without bypassing the kernel, (b) the* `AF_XDP` *kernel-bypass interface, or (c) hardware offload with a programmable NIC.*

plane, and the NIC leverages DMAs to write packets to a memory buffer that applications consume directly.

XDP and eBPF. The XDP interface enables the implementation of high-performance networking applications on Linux by combining programmable packet processing and kernel-bypass [10]. Applications implement custom packet processing in a programming language such as C, which compiles to the eBPF virtual machine instruction set. These eBPF programs run before the OS forwards the packets to the in-kernel network stack. The OS provides an in-kernel virtual machine for eBPF programs, but they can also be offloaded to a programmable NIC [18]. As shown in Figure 2, XDP supports multiple configurations: POSIX sockets API, `AF_XDP` kernel-bypass, and NIC offload; the `AF_XDP` socket type in XDP allows applications to by-pass the OS network stack entirely if needed. As eBPF is programming language agnostic, applications can reuse the same partitioning code in the XDP program and the application. For example, applications can use the same application code implemented in C for request steering or run existing packet processors implemented in the P4 [3] programming language on XDP [38]. The availability of XDP and eBPF in a commodity OS makes application-level partitioning with packet steering a practical approach for applications.

3 APPLICATION-LEVEL PARTITIONING AND PACKET STEERING

Applications must embrace parallelism to take advantage of multi-core CPUs. In the thread-per-core approach to improve parallelism, applications restrict the number of application threads to the number of CPU cores, and partition application data in DRAM and the resources between the CPU cores to eliminate thread synchronization and OS-level locks. However, current solutions are unable to steer packets to the CPU core that can independently serve the request without exposing the partitioning schema to its clients. We now show how application-level packet steering with a programmable NIC can solve this problem.

3.1 Partitioning in the thread-per-core model

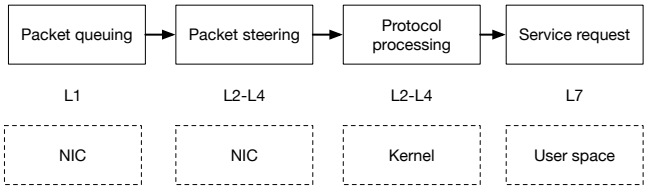
Partitioning is increasingly being adopted as a strategy to improve application-level parallelism. In the thread-per-core approach, an arriving packets needs to be steered to the CPU core that can serve the request. This approach allows the CPU cores to run independently by eliminating the need to synchronize threads on application-level data access, and avoiding OS-level locking (in some cases). However, steering requests to the correct CPU core either requires clients to specify the partition in the request [23], or the application threads need to redirect the requests. This is because the approaches to steer the packets using traditional non-programmable NICs are restricted to L2-L4 protocol headers. A programmable NIC can solve the problem of request steering by inspecting L7 packet headers.

In spite of its benefits, the thread-per-core approach for partitioning data and resources has its limitations. For skewed workloads, this approach can overload some CPU cores while leaving the others underutilized. This can be addressed by binning CPU cores in clusters, and making each CPU cluster responsible for a partition of the data. Each CPU core cluster could use the traditional shared-memory approach which is known to scale to small core counts [11]. The CPU core clusters could also be partitioned around sub-NUMA clusters, which groups CPU cores by memory controllers [28].

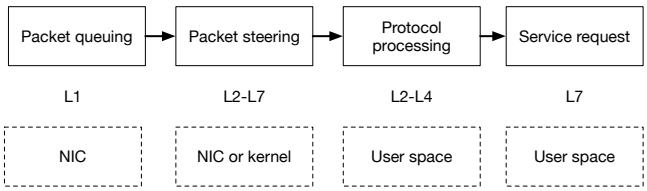
3.2 Partition-aware packet steering

Request processing is performed in different stages across the NIC, the kernel, and user space as shown in Figure 3(a). The NIC first performs L1 processing to queue packets in the NIC RX queues and then performs packet steering via RSS or Flow director by L2-L4 packet headers. Finally, the OS network stack performs protocol processing and hands over the packet to user space for L7 protocol processing. Note that the kernel forwards a packet to a user space thread based on its own packet steering policy, and this steering has no knowledge of application-specific partitioning. When data is partitioned using the thread-per-core approach, the user space thread which receives the packet needs to first forward it to the remote thread responsible for serving the packet, and then notify the remote thread [7, 35].

We measured the time required to notify a thread on an Intel Xeon E5-2686 v4 @ 2.30GHz with two non-uniform memory access (NUMA) nodes running Ubuntu 18.04.3 LTS with Linux 4.15.0-1051-aws. In our experimental setup, we had two threads running on different CPUs on the system. The first thread notifies the second thread by writing to an eventfd file descriptor that the second thread reads. The thread notification time is the time difference between the call to the write system call and the return of the read system



(a) Request processing flow with the OS network stack.



(b) Request processing flow with XDP and eBPF.

Figure 3: Request processing. *Packets traverse through multiple stages–packet queuing, packet steering, and protocol processing—before the application thread services the request; L1-L7 denote the seven layers of the OSI model.*

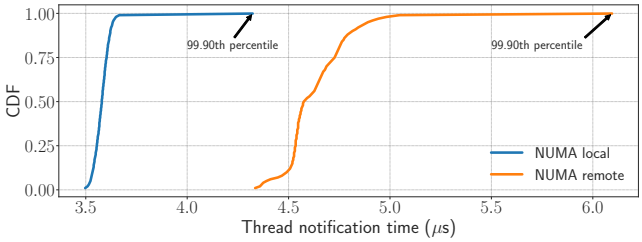


Figure 4: Thread notification time. *The cumulative distribution function (CDF) of the thread notification time highlights that the time required to notify a user space thread is significantly larger than the time between successive packet arrivals on a fast NIC; a 40 Gbps NIC can receive a 64 byte packet close to every* 12 ns *[15].*

call. As shown in Figure 4, the 99.9th percentile of thread wakeup delay on the same NUMA node is 4.32 µs and 6.09 µs on a remote NUMA node. In contrast, a 40 Gbps NIC can receive a 64 byte packet close to every 12 ns [15].

As shown in Figure 3(b), a programmable NIC with XDP and eBPF can perform partition-aware packet steering. The NIC performs L1 processing to queue packets, and a program running on the NIC inspects L2-L7 protocol headers to steer packets directly to the CPU core that can serve the request. The user space thread then performs the protocol processing and serves the request. For example, in a key-value store, the partition identifier can be the request key, which determines the target of the operation requested by a client. In an RPC call, the partition identifier can be the name of the RPC function or a parameter of the RPC call, which determines a service thread that implements the RPC function.

If an application workload does not access the resource partition and data sets uniformly, the NIC packet steering can perform load balancing between the CPU cores. For example, instead of steering packets to a single CPU core, the NIC can round-robin request processing between all CPU cores. The trade-off with load balancing is that the request-processing CPU cores must either access CPU remote memory or use software steering to complete the request. Another possible optimization at NIC level is response caching. For example, if processing a request needs a lot of CPU cycles, caching the response at NIC level can be beneficial. For requests that are not CPU-intensive, caching responses can still improve performance, because caching eliminates transferring data over the PCIe bus. However, the trade-off with response caching that the NIC program needs more application-specific knowledge to perform the cache lookup.

3.3 Example: A Key-Value Store

Key-value (KV) stores are a widely understood topic [6, 13, 14, 20, 23]. Although KV stores have been criticized recently [2], they serve as an easy to understand example of a network-intensive application. We believe that KV stores can benefit from application-level partitioning and packet steering and that the lessons are applicable to a broader range of networked applications that need to embrace parallelism.

Request processing overview. For our discussion, we assume two types of requests, *get* and *set*, both of which contain a request key k. Request processing in the KV store starts when a packet containing the client request arrives on the NIC. The NIC forwards the packet to one if its RX queues, depending on how the device driver configures the NIC. The NIC obtains a DMA descriptor from the NIC RX queue, and DMAs packet data to a region of the DRAM pointed to by the DMA descriptor. The device driver notices that a new packet has arrived and forwards it to the network stack. The network stack performs the L2-L4 protocol processing, after which it hands over the packet to the application. The application parses the request and performs necessary operations. For example, for a *get* request, a value is looked up from a data structure by the request key. Finally, a response message is generated for the request and handed over to the network stack, which performs protocol processing and hands over the packet to the NIC.

Design. The design goals for our KV store are as follows.

- Improve hardware utilization with CPU and NIC offload.
- Exploit multicore CPU parallelism efficiently.
- Do not expose application-level partitioning to clients.

As shown in Figure 1, we propose a design that combines application-level partitioning and packet steering with programmable NIC. Similar to previous approaches [14, 23], it runs one thread per CPU core, and it partitions the keyspace in DRAM between the CPU cores. That is, a partitioning function $p(k)$ maps a key k_0 to CPU C_n, key k_1 to CPU C_m, and so on. For example, the MICA KV store partitions the keys by using some bits of the hash of the keys [23]. With this application-level partitioning in place, a program running on the NIC parses L7 packet headers to determine request keys and uses the same partitioning function $p(k)$ to steer packets to the target CPU core. For example, with the Memcache protocol, the request key is part of the request headers of all operations. The application thread parses the complete request, performs the requested operation, generates a response, and places it on the NIC TX queue. The difference to previous approaches is that the NIC steers packets directly to the target thread. This eliminates overheads

of software steering approaches, while keeping the partitioning scheme transparent to clients.

Implementation. We propose to implement our approach using POSIX APIs for OS threads, memory management, and so on, and Linux's XDP interface for networking. The KV store spawns an OS thread for each CPU core assigned for the application with `pthread_create` and allocates memory regions individually for each CPU core with `mmap`. The packet steering logic is implemented as an eBPF program. The KV store uses the `AF_XDP` socket type to open a kernel-bypass channel between the XDP subsystem and the user space threads. The NIC program determines the CPU of a request key by parsing the L7 protocol headers, looks up the per-CPU `AF_XDP` socket (XSK) and uses the `bpf_redirect_map` function to forward the packet to the XSK. The userspace process then receives the packet via the XSK and performs the necessary protocol and request processing. For example, in the case of Memcached over UDP, the application needs to parse the UDP headers and the Memcached request, and perform the requested operation such as *get* or *set*, to retrieve or update a value, respectively.

4 DISCUSSION

We propose an approach that combines application-level partitioning and packet steering with a programmable NIC. At the same time, there are various aspects of the approach that we want to explore.

Other use cases. We have proposed an approach to combine application-level partitioning and packet steering, but only given an example of a KV store. We have also assumed that the KV store supports requests on a single key, which makes partitioning easy because there is a simple mapping between the request and a CPU core. However, such mapping does not exist in many use cases. For example, a multi-key request and range query possibly need to access data on multiple CPU cores. It is possible, however, to take advantage of the programmable packet processor for multi-key requests and range queries in a different manner. The packet processor can load balance the requests between multiple cores to eliminate request processing imbalance. For example, the packet processor can round-robin packet processing on the CPU cores of all the keys. Also, extending XDP to support packet duplication can allow multi-key requests to be sent to all the cores required to serve the request.

Partitioning. The partitioning scheme of the thread-per-core model is known to have problems with skewed workloads. For example, if clients access a subset of the data more than other data, some CPU cores can be overloaded, whereas the rest of the CPU cores are underutilized. If partitioning logic is transparent to clients, as in our proposed approach, it is easier for applications to take advantage of more complex partitioning. For example, Intel CPUs starting from the Skylake microarchitecture support sub-NUMA clustering where CPU cores are partitioned internally into clusters that share the L3 cache and a memory controller [28]. With application-level packet steering, it becomes viable to implement a shared-something model. In a shared-something model, application data is partitioned by sub-NUMA clustering topology and not by individual cores. Application threads running on a cluster of cores can utilize lockless data structures which scale on small core counts [11].

Packet encryption. Packet encryption imposes a problem for our proposed approach. Current packet steering approaches leverage L2-L4 packet headers, which are often unencrypted even if an application uses a secure protocol such as TLS to encrypt the rest of the packet data. However, in our proposed approach, the NIC uses L7 protocol headers to steer packets, but it cannot do that if it is unable to access the packet data [31]. One possible solution to this problem is to decrypt packets at the NIC. As host CPU is a bottleneck, it makes sense to offload packet decryption to the NIC, which is already supported by some NICs [4]. An open challenge is to integrate NIC TLS offloading with XDP to make it available for eBPF programs. Another possible solution to the issue of steering encrypted packets is to use homomorphic encryption techniques, which allow computation on encrypted data. For example, if a database system already supports secure query processing with homomorphic encryption [26], packet steering could operate on the encrypted request key because the rest of the application also operates on them. It is not clear if homomorphic encryption for packet steering is feasible, but it is a research direction we think is worth exploring, because it is already suggested for other programmable NIC offload functions [31].

XDP and eBPF. We are implementing a prototype KV store using XDP and eBPF because they are available on Linux and combine programmable packet processor with kernel-bypass. Although XDP and eBPF are still under development, they are already a good fit for implementing our proposed approach. However, some parts of XDP and eBPF functionality are not clear. We therefore intend to explore their limits and eliminate them if possible while implementing a prototype. One XDP limitation for our approach is if the `AF_XDP` kernel-bypass interface can deliver packets directly to user space. For example, the Netronome NIC driver does not currently support redirecting packets from eBPF offload on the NIC directly to `AF_XDP` socket or passing information from the NIC to the host in packet metadata [17]. However, both limitations are addressable with software changes without changes to the underlying hardware. Another issue with `AF_XDP` is that the memory buffer for packets uses fixed-size chunks. These memory chunks need to be large enough to accommodate the largest possible packets, which incurs a high internal fragmentation for small packets. It is not clear how this limitation can be lifted or mitigated. One advantage of eBPF is that it is flexible enough to accommodate various parsing identifiers for request steering.

Performance. We intend to explore the performance of our approach by implementing a prototype KV store and evaluating its performance against previous solutions such as MICA. Parsing L7 protocol headers at the NIC is expensive, but it is not clear how much it eliminates other overheads. However, we know from previous works, such as MICA and HERD, that combining application-level partitioning and packet steering improves performance.Furthermore, in some cases, we can reuse the results of the L7 packet header parsing in the application code. For example, when determining the partition of a key, we calculate a hash of the key. With XDP, we can pass this calculated hash as part of the packet in the user space thread, and reuse the hash when updating an internal data structure, for example. Another aspect we want to explore is the

impact of scalability limitations of multicore SoC-based NICs [8] to our approach.

5 RELATED WORK

Packet steering. Multi-queue NICs support receive-side scaling (RSS) and Flow Director, which distribute packets to multiple NIC receive queues [1]. The OS maps the NIC queues to different CPUs, which enables parallel processing of the packets. However, the distribution hash function and hash type are fixed in hardware, which limits application design choices. Programmable RSS provides a custom packet distribution hash that leverages eBPF [32]. Our proposed approach extends programmable RSS because we combine packet steering with application-level partitioning and kernel-bypass. Our partitioned packet steering approach similar to the one proposed by Floem [30], except we design around Linux XDP and eBPF subsystems instead of a having a separate compiler. Floem is a data-flow programming language, compiler, and runtime that targets NIC-accelerated applications. One of the proposed NIC offloads approaches with Floem is a CPU-NIC co-design architecture for sharded applications, where the NIC performs packet steering, and the CPU executes the application logic, similar to our proposed approach. In contrast, we focus on how to enable this co-design approach with the eBPF virtual machine and the XDP interface, which are available on Linux. We believe that this is useful to study the limits of existing OS interfaces along with eBPF and XDP. Our approach also complements iPipe [25] by leveraging eBPF and ensuring that the application code is agnostic to the capabilities of the underlying hardware.

Application-level partitioning. The MICA in-memory key-value store partitions data between CPU cores and uses NIC Flow Director to map requests to specific CPUs [23]. However, for mapping requests to a CPU, MICA requires clients to specify the CPU via a UDP port. This requires clients to know how keys are mapped to the CPU cores. The HERD key-value store uses a per-core request memory buffer to directly RDMA to the target CPU core [14], which has the same problem of exposing application partitioning scheme to clients.

NIC offload approaches. KV-Direct is an in-memory key value store, which is fully implemented to run on the NIC [20]. NetCache implements key-value store on a programmable switch to offload request processing from servers to the network to improve performance [13]. The approach is similar to KV-Direct but focuses on exploiting the programmability of modern ASIC switches, instead of a programmable NIC. Offloading full application to the NIC improves performance, but is impractical for general purpose applications because of the high development cost incurred due to inability to reuse existing code.

Dynamic CPU core allocation. Shenango is a runtime system that targets low latency applications and CPU efficiency [29]. Typically low-latency is achieved with a busy-polling technique which wastes CPU cycles. Shenango provides low-latency for applications by dedicating a single CPU core that polls the NIC for arriving packets and steers them to dynamically allocated application CPU cores. Each runtime in Shenango has its own MAC and IP addresses. When a packet arrives, the Shenango runtime uses a software-based Receive Side Steering (RSS) hash to determine target CPU core. We believe that our proposed packet steering approach could complement Shenango. For example, the per-runtime packet steering logic could be offloaded to a NIC, eliminating the software-based steering, further reducing CPU cycle consumption.

6 CONCLUSION

We have proposed a combination of application-level partitioning and packet steering with a programmable NIC to improve application-level parallelism. An application partitions its resources and data in DRAM between CPU cores, and a program running on a programmable NIC inspects L7 protocol headers to steer the request to its partition. We argue that XDP and eBPF make this approach practical because they provide both a programmable packet processor and kernel-bypass, and allow offload to the NIC. We believe that this approach can reduce latency and improve throughput. Furthermore, the approach complements advanced partitioning techniques because partitioning is transparent to clients.

We are currently working on a prototype in-memory KV store using this approach and are planning to compare its performance against previous works such as MICA [23]. We are also considering modifying existing application frameworks, such as libevent [21] and libuv [22], to use our approach. This would allow some existing applications to take advantage of application-level partitioning and packet steering. We are also exploring the limits of XDP and eBPF to our approach, and looking into alternatives to address them.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers, and our shepherd Abderrahmen Mtibaa for their helpful comments. We thank Björn Topel, Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Jakub Kicinski, for explaining the various technical details of XDP and eBPF to us, and providing helpful comments. This work is supported by the Business Finland 5G Force project.

REFERENCES

[1] [n.d.]. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt. [accessed 2019-06-12].
[2] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast Key-value Stores: An Idea Whose Time Has Come and Gone. In *HotOS*. https://doi.org/10.1145/3317550.3321434
[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR* (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890
[4] Chelsio Communications. 2019. Crypto Offload. https://www.chelsio.com/crypto-offload/. [accessed 2019-06-25].
[5] Andy Currid. 2004. TCP Offload to the Rescue. *Queue* 2, 3 (May 2004), 58–65. https://doi.org/10.1145/1005062.1005069
[6] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding for Improving Tail Latencies in In-memory Key-value Stores. In *NSDI*. 15. http://dl.acm.org/citation.cfm?id=3323234.3323242
[7] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 2019. The Impact of Thread-Per-Core Architecture on Application Tail Latency. In *ANCS*.
[8] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*. http://dl.acm.org/citation.cfm?id=3307441.3307446

[9] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc.

[10] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *CoNEXT*. https://doi.org/10.1145/3281411.3281443

[11] David A. Holland and Margo I. Seltzer. 2011. Multicore OSes: Looking Forward from 1991, Er, 2011. In *HotOS*. 1. http://dl.acm.org/citation.cfm?id=1991596.1991640

[12] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*. 14. http://dl.acm.org/citation.cfm?id=2616448.2616493

[13] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*. https://doi.org/10.1145/3132747.3132764

[14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *SIGCOMM*. https://doi.org/10.1145/2619239.2626299

[15] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *ASPLOS*. https://doi.org/10.1145/2872362.2872367

[16] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration As an OS Service. In *EuroSys*. https://doi.org/10.1145/3302424.3303985

[17] Jakub Kicinski. 2019. Private communication.

[18] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Hardware Offload to SmartNICs: cls_bpf and XDP. https://www.netronome.com/m/documents/eBPF_HW_OFFLOAD_HNiMne8_2_.pdf. [accessed 2019-06-26].

[19] Avi Kivity. [n.d.]. Adventures with Memory Barriers and Seastar on Linux. https://www.scylladb.com/2018/02/15/memory-barriers-seastar-linux/. [accessed 2019-06-26].

[20] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *SOSP*. https://doi.org/10.1145/3132747.3132756

[21] libevent: an event notification library. 2002. https://libevent.org/. [accessed 2019-06-26].

[22] libuv: Cross-platform asynchronous I/O. 2011. https://libuv.org/. [accessed 2019-06-26].

[23] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *NSDI*. 16. http://dl.acm.org/citation.cfm?id=2616448.2616488

[24] Linux Foundation. 2010. Data Plane Development Kit. https://www.dpdk.org. [accessed 2019-06-27].

[25] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications Onto smartNICs Using iPipe. In *SIGCOMM*. https://doi.org/10.1145/3341302.3342079

[26] Murali Mani. 2013. Enabling Secure Query Processing in the Cloud Using Fully Homomorphic Encryption. In *DanaC*. https://doi.org/10.1145/2486767.2486775

[27] Jeffrey C. Mogul. 2003. TCP Offload is a Dumb Idea Whose Time Has Come. In *HOTOS*. 1. http://dl.acm.org/citation.cfm?id=1251054.1251059

[28] David Mulnix. 2017. Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview. [accessed 2019-06-09].

[29] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *NSDI*. http://dl.acm.org/citation.cfm?id=3323234.3323265

[30] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-accelerated Network Applications. In *OSDI*. http://dl.acm.org/citation.cfm?id=3291168.3291217

[31] Dan R. K. Ports and Jacob Nelson. 2019. When Should The Network Be The Computer?. In *HotOS*. 7. https://doi.org/10.1145/3317550.3321439

[32] Programmable RSS Demo. 2014. https://github.com/Netronome/bpf-samples/tree/master/programmable_rss. [accessed 2019-06-12].

[33] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*. http://dl.acm.org/citation.cfm?id=2342821.2342830

[34] Karl Rupp. 2018. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/. [accessed 2019-06-25].

[35] Seastar. 2014. http://www.seastar-project.org/. [accessed 2019-06-11].

[36] Brent Stephens, Aditya Akella, and Michael M. Swift. 2018. Your Programmable NIC Should Be a Programmable Switch. In *HotNets*. https://doi.org/10.1145/3286062.3286068

[37] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB*. http://dl.acm.org/citation.cfm?id=1325851.1325981

[38] Fabian Ruffy William Tu and Mihai Budiu. 2018. Linux Network Programming with P4. http://vger.kernel.org/lpc_net2018_talks/p4c-xdp-lpc18-paper.pdf. [accessed 2019-06-16].