SoftNIC: A Software NIC to Augment Hardware

Sangjin Han
Keon Jang
Aurojit Panda
Shoumik Palkar
Dongsu Han
Sylvia Ratnasamy

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-155
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

May 27, 2015

SoftNIC: A Software NIC to Augment Hardware

Sangjin Han, Keon Jang*, Aurojit Panda, Shoumik Palkar, Dongsu Han†, Sylvia Ratnasamy

University of California, Berkeley *Intel Labs †KAIST

Abstract

As the main gateway for network traffic to a server, the network interface card (NIC) is an ideal place to incorporate diverse network functionality, such as traffic control, protocol offloading, and virtualization. However, the slow evolution and inherent inflexibility of NIC hardware have failed to support evolving protocols, emerging applications, and rapidly changing system/network architectures. The traditional software approach to this problem—implementing NIC features in the host network stack—is unable to meet increasingly challenging performance requirements.

In this paper we present SoftNIC, a hybrid software-hardware architecture to bridge the gap between limited hardware capabilities and ever changing user demands. SoftNIC provides a programmable platform that allows applications to leverage NIC features implemented in software and hardware, without sacrificing performance. Our evaluation results show that SoftNIC achieves multi-10G performance even on a single core and scales further with multiple cores. We also present a variety of use cases to show the potential of software NIC augmentation.

1. Introduction

Modern Network Interface Cards (NICs) are constantly evolving to support new features. Over the last decade, the role of NICs in modern server systems has grown beyond simply relaying traffic between a server's CPUs and its network link—they now host advanced *NIC features*, such as protocol offloading, packet classification, rate limiting, and virtualization. Three factors have contributed to this trend: (1) new applications whose performance requirements cannot be met by legacy networking stacks, necessitating hardware augmentation; (2) the increasing popularity of virtualization, where it is desirable for NICs to support switching functionality; and (3) the rise of multi-tenant datacenters and cloud computing for which NICs must provide isolation mechanisms.

However, the rise of smart NICs has been no free lunch. As we discuss in §2, NICs are increasingly complex: they support an ever-increasing number of features that are difficult to configure, compose, or evolve. Despite this, NICs often fail to meet application needs and their lack of flexibility impedes innovation. At the heart of the problem is that applications (software) evolve faster than the timescales at which new NIC support (hardware) emerges.

The common belief is that the pitfalls of feature-rich NICs must be tolerated because they are necessary to achieve the performance we want; *i.e.,* implementing NIC features in software is not a practical alternative. What should give us pause, however, is the mounting evidence that the performance gap between hardware and software network processing is significantly getting smaller than previously believed [14, 21, 26, 59].

We argue that a hybrid hardware-software approach can effectively augment or extend NIC features, while providing flexibility and high performance. This software-augmented NIC—which we term SoftNIC—serves as a fallback or even an alternative to the hardware NIC. SoftNIC is flexible and programmable; it serves as an incubator for next-generation hardware features, as well as a permanent home for features that are too complex for hardware implementation, too specialized to merit scarce NIC resources, or whose performance needs are easily met with software.

Building a system such as SoftNIC is challenging. To support applications' needs, SoftNIC must provide rich programmability. However it must do so without compromising on performance. Modern NICs offer high throughput (10–40 Gbps) and low end-to-end latency ($< 10\,\mu s$) [4]. At the same time, they also provide *guarantees* on performance and isolation among virtual machines (VMs) or applications. SoftNIC must offer similar performance capabilities, but meeting these requirements with a software implementation can be very challenging. For example, legacy kernels and hypervisors are notoriously slow and inefficient at network processing. And while specialized packet processing frameworks do offer high performance, they lack the rich programmability we seek [26, 59] or do not address the problem of performance guarantees [35].

Hence, our paper make three contributions:

1. First, we propose a new architecture for extending NIC features. Developers can use SoftNIC to develop features in software while incurring minimal performance overhead and leveraging features from the hardware NIC. Application developers can use SoftNIC as a hardware abstraction layer (HAL) to build software that uses NIC features without worrying about cases where they are unavailable or incomplete.
2. Second, we present a scheduling framework that can support a wide range of operator-specified performance isolation policies and provide fine-grained performance guarantees for applications (§3.4).
3. Third, we present several use cases of SoftNIC, showing how SoftNIC can improve the flexibility and/or performance of both existing and forward-looking applications. Our evaluation in §6 demonstrates that SoftNIC is a convenient and versatile platform for implementing advanced NIC features that benefit a variety of applications.

2. Motivation

The list of features that need to be supported by NICs has grown rapidly: new applications appear with new requirements; new protocols emerge and evolve; and there is no shortage of new ideas on system/network architectures that require new NIC features (*e.g.,* [6, 29, 47, 51, 54, 57]). Unfortunately, NIC hardware is not a silver bullet, as many issues arise due to the inherent inflexibility of hardware.

1. NICs may not support all desired features. Due to the slow development cycle (typically years), there often exists a *demand-availability gap*—a disparity between user demands for NIC features and their actual availability in products. For example, even minor updates (*e.g.,* do not duplicate the TCP CWR flag across all segments when using TSO, to avoid interference with congestion control algorithms [12, 58]) that can be implemented by changing a few lines of code when implemented in software, often requires a full hardware revision.
2. Once a feature is hardwired in the NIC, it is almost impossible to control its fine-grained behavior, often rendering the feature useless. For example, in the case of protocol offloading (*e.g.,* checksum offload and TSO/LRO), tunneled packets (*e.g.,* VXLAN) cannot take advantage of unless the NIC understands the encapsulation format [36], even though hardware essentially has the logic for protocol offloading built-in. Lack of fine-grain control also makes it difficult to combine features to acheive the end goal. For example, although SR-IOV provides better performance and isolation with hypervisor bypass, it cannot be used by cloud providers unless it supports additional features to enforce traffic policies such as security/QoS rules, advanced switching, and VM mobility support [54].
3. NIC hardware has resource restrictions, *e.g.,* the number of rate limiters, flow table size, and packet buffers, limiting its applicability. As a result, several systems [22, 32, 38, 50] have resorted to software work around the resource issue.

We argue that these issues will persist or even worsen in the future. Figure 1 presents the number of lines of code for NIC device drivers as an indirect index of NIC hardware complexity. The trends over the NIC generations show that the complexity has grown tremendously; *e.g.,* the driver for a high-end NIC ($>$ 10 GbE) on average contains 6.8$\times$ more code than that of a 1 GbE NIC driver. This ever increasing hardware complexity has led to an increase in the time and cost for designing and verifying new NIC hardware.

SoftNIC presents a new approach to extending NIC functionality; its adds a software shim layer between the NIC hardware and the network stack, so it can augment NIC features with software. It enables high-performance packet processing in software, while taking advantage of hardware features. By design, it supports high performance, extensibility, modular design, and backwards compatibility with existing software.

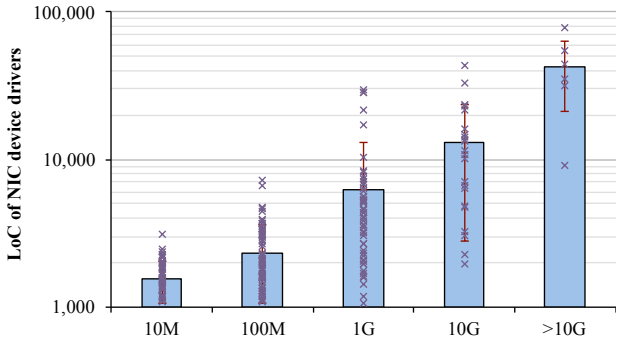


Figure 1: Growing complexity of NIC hardware, indirectly measured with the lines of device driver code in Linux 3.19.

3. SoftNIC Design

SoftNIC is a programmable platform that allows applications to leverage software and hardware implementations of NIC features. SoftNIC is implemented as a shim layer between applications and NIC hardware, in order to achieve the best of both worlds: the flexibility of software and the performance of hardware.

SoftNIC provides a software augmentation layer for NICs, which imposes a few issues to be addressed. We set three main design goals (§3.1) and then provide details on the overall architecture (§3.2), the packet processing pipeline (§3.3), and the scheduler used for resource allocation (§3.4). In this section we only describe the design aspects of SoftNIC, deferring implementation details on performance to §4.

3.1 Design Goals

G1: Programmability and Extensibility
SoftNIC must allow users to configure functionality to support a diverse set of uses. In particular, users should be able to compose and configure NIC features as required, and SoftNIC should make it easy to add support for new protocols and NIC functions. Also a clean separation between control and data plane is desired, so that an external controller can dynamically reconfigure the data path to implement user policies.

G2: Application Performance Isolation
Hardware NICs support mechanisms that can be used to implement policies regarding per-link bandwidth resource used by an application, *e.g.,* "limit the bandwidth usage for this application to 3 Gbps" or "enforce max-min fairness across all applications." SoftNIC should provide flexible mechanisms to support a variety of policies on application-level performance. However, implementing these policies imposes a unique challenge for SoftNIC—the processor[1] itself is a limited resource and must be properly scheduled to process traffic spanning multiple links and applications.

[1] We only manage processor time used *within* the SoftNIC dataplane; the processor time used by applications themselves is out of SoftNIC's control.

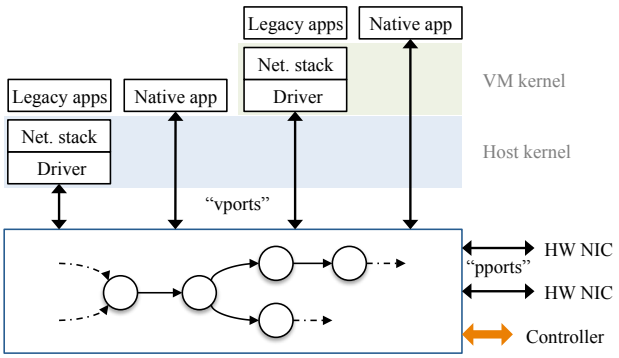


Figure 2: SoftNIC architecture

G3: Backward Compatibility
Finally, using SoftNIC should require no modifications to existing software or hardware. At the same time SoftNIC should be an enabler for a gradual transition to new hardware and software designs, *e.g.,* allowing applications to use new network APIs to improve performance.

3.2 Overall Architecture

Figure 2 shows the overall system architecture of SoftNIC. The packet processing pipeline is represented as a dataflow (multi)graph that consists of modules, each of which implements a NIC feature. Ports act as sources and sinks for this pipeline. Packets received at a port flow through the pipeline to another port. Each module in the pipeline performs module-specific operations on packets. Our dataflow approach is heavily inspired by Click [35], although we both simplify and extend Click's design choices for SoftNIC (§7).

SoftNIC's dataflow graph supports two kinds of ports. (i) Virtual ports (*vports*) are the interface between SoftNIC and upper-layer software. A vport connects SoftNIC to a *peer*; a peer can either be the SoftNIC device driver (which is used to support legacy applications relying on the kernel's TCP/IP stack) or a SoftNIC-aware application that bypasses the kernel. A peer can reside either in the host[2], or in a VM. (ii) Physical ports (*pports*) are the interface between SoftNIC and the NIC hardware (and are hence not exposed to peers). Each pport exposes a set of primitives that are natively implemented in its NIC hardware (*e.g.,* checksum offloading for specific protocols).

A vport pretends to be an ideal NIC port that supports all features required by its peer. The modules in the packet processing pipeline are responsible for actually implementing the features, either in software, by offloading to hardware, or with combination of both. SoftNIC thus abstracts away the limitations of the underlying NIC hardware from peers, effectively providing a hardware abstraction layer (HAL) for NICs. SoftNIC both allows for rapid prototyping of new NIC functionality in software, and is also useful in cases where hardware provides incomplete functionality (*e.g.,* by providing software implementation that allow a feature to be used with a new protocol) or insufficient capacity (*e.g.,* by using both software and hardware flow tables).

SoftNIC provides a control channel that allows for a clean separation between the control and data plane. An explicit control channel allows an external controller to dictate data path policy, while SoftNIC itself focuses on providing data-plane mechanisms for this policy. The control channel supports three types of operations: (1) updating the data path (e.g., adding/removing modules or ports), (2) configuring resource allocations (e.g., limiting CPU/bandwidth usages for applications), (3) managing individual modules (e.g., updating flow tables or collecting statistics). In this paper we solely focus on the design and implementation of SoftNIC, rather than the external controller.

3.3 Modular Packet Processing Pipeline

The SoftNIC pipeline is composed of modules, each of which implements a NIC feature. An implementation of a module defines several handlers, including ones for processing packets and timer events. Each module instance may contain some internal state, and these instances are the nodes in SoftNIC's data flow graph that specifies the order in which modules process packets (Figure 3). When a node has multiple outgoing edges (*e.g.,* classification modules), the module decides which of the edges a packet is sent out.

Often in-band communication between modules is desirable for performance and modularity. For example, a parser module performs header parsing and annotates the packet with the result as metadata, so that downstream modules can reuse this information. Along the pipeline, each packet carries its metadata fields abstracted as a list of key-value pairs. Modules specify which metadata fields they require as input and the fields they produce. Explicitly declaring metadata fields is useful in two ways. First, if a module in the pipeline requires a field that is not provided by any upstream modules, SoftNIC can easily raise a configuration error. Second, any unused metadata field need not be preserved, and SoftNIC can reduce the total amount of space required per-packet. Note that this optimization can be performed at configuration time and does not incur any runtime overhead.

Pipeline Example: We walk through a simple example to illustrate how packets are processed with metadata. In Figure 3, the tables on the edges show packet metadata at the point when a packet traverses the edge. The dotted arrows represent the input/output metadata fields for each module. The user configures the SoftNIC pipeline so that transmitted packets are processed by (i) a switching service, (ii) TCP segmentation offload (TSO), and (iii) checksum offloading. The NIC in this example has no TSO functionality, and only supports checksum offloading for TCP/IPv4 packets. Nevertheless, the vport appears as a fully featured NIC to the peer, and provides both TSO and protocol-agnostic checksumming.

[2] Throughout this paper, we use the loosely defined term "host" to refer to a hypervisor, a Dom0, a root container, or a bare-metal system, to embrace various virtualization scenarios.

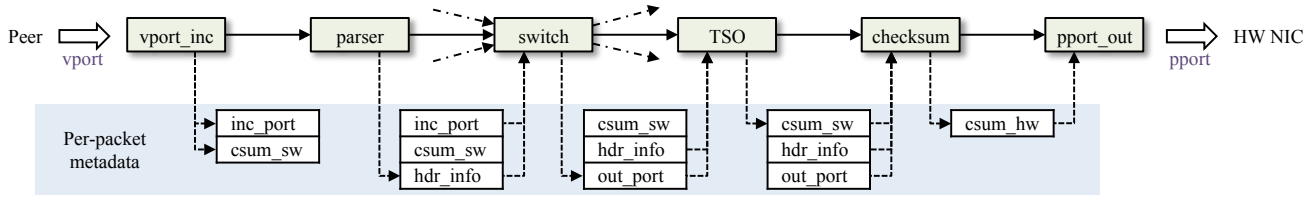


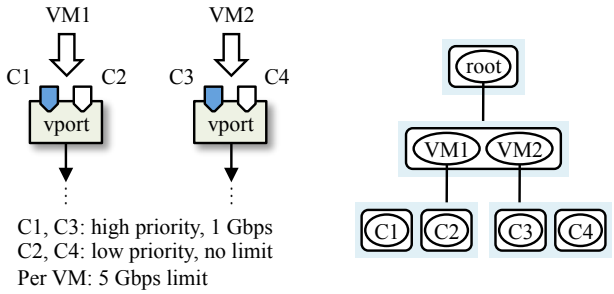
Figure 3: A pipeline example with parser, switch, TSO, and checksum offloading modules. Metadata evolves as a packet traverses the pipeline.

When the peer sends packets to the vport, each packet is annotated with its desired offloading behavior. For example, an annotation indicating that a packet requires checksum offloading is of the form "calculate checksum over byte range X using algorithm Y, and update the field at offset Z." The packet data and its annotations are packaged as a packet descriptor and pushed into the vport queue. In contrast to hardware NICs, the size and format of the descriptor are flexible and can change depending on the features exposed by the vport. Packet processing proceeds as follows.

1. **vport_inc** pulls a packet descriptor, creates a SoftNIC packet buffer with the packet data, and adds metadata fields for the input port ID (`inc_port`) and checksum offloading description (`csum_sw`).
2. **parser** inspects the packet's L2–L4 header and records the results in `hdr_info`.
3. **switch** updates the MAC table using `inc_port` and `hdr_info`. Then it uses the destination address to determine the output edge along which the packet is sent. In this example the chosen edge goes to the TSO module.
4. **TSO** begins by checking whether the packet is a TCP packet larger than the MTU of `out_port`[3]. If so, the module segments the packet into multiple MTU-sized packets (and all metadata fields are copied), and updates `csum_sw` appropriately for each.
5. **checksum** uses `csum_sw` and `hdr_info` to determine if checksum calculation is required, and further if this needs to be done in software. When checksum computation can be carried out by the NIC containing `out_port`, the module simply sets `csum_hw` to "on", otherwise the module computes the checksum in software.
6. **pport_out** sends the packet to the NIC, with a flag indicating whether the hardware should compute the checksum, in the hardware-specific packet descriptor format.

This example highlights how SoftNIC modules can flexibly implement NIC features and opportunistically offload computation to hardware. For ease of exposition we have only described the transmission path, the receive path is implemented similarly. Also, note a packet does not always need to flow between a vport and a pport: *e.g.,* virtual switching (between vports) [52] or multi-hop routing (between pports) [5].

[3] While in this dataflow graph the output port can be "inferred", explicit use of `out_port` allows greater flexibility: *e.g.,* allowing us to separate classification (which determines the `out_port`) from splitting (which diverts the packet flow in the pipeline).



(a) Example policy (b) Class tree

Figure 4: An example of high-level performance policy with four traffic classes `C1`–`C4`, and its corresponding class tree representation. Each circle is either a class (leaf) or a class group (non-leaf). `VM1` and `VM2` have the same priority (within the same box).

3.4 Resource Scheduling for Performance Guarantees

In contrast to NIC hardware, which is inherently parallel at the gate level, SoftNIC needs a scheduler to decide what packet and module get to use the processor. For example, a simple form of scheduling[4] would involve using weighted round-robin scheduling to pick an input port, fetching a packet from this port and then processing it until it is sent out another port. This scheme is used by many software packet processing systems [21, 34, 35, 52, 60] for its simplicity. However, its crude form of fairness—the same number of packets across input ports—may not achieve the operator's desired *policy* for applications-level performance.

The ultimate goal of the SoftNIC scheduler is to allow the operator to specify and enforce policies for applications. Instead of mandating a single policy (*e.g.,* priority scheduling) that must be used in all deployment scenarios, SoftNIC provides a set of flexible mechanisms that can be easily composed to implement a broad set of high-level policies. The scheduler makes policy-compliant allocations of processor and bandwidth resources using the mechanisms.

[4] Software packet processing frameworks require two complementary types of scheduling: (i) packet scheduling, where a functional module (*e.g.,* `PrioSched` element in Click [35]) selects the next packet to process; and (ii) CPU scheduling, where the framework determines when and how often each module is executed (*i.e.,* gets processor time). For the former, we discuss how our approach differs from Click's in §7. In this section, we focus on the latter, as it has received less attention from the research community.

Let us consider a typical policy example, as presented in Figure 4(a). In this example, each VM gets a fair share of bandwidth up to 5 Gbps. Each VM has two types of traffic: interactive traffic (`C1` and `C3`) and background traffic (`C2` and `C4`). The interactive traffic has higher priority but is limited to 1 Gbps per VM. Note that the policy includes: (i) fair sharing between VMs, (ii) rate limiting for each VM and its interactive traffic, (iii) fixed priorities between the traffic types, and (iv) hierarchical composition of the above. The design of SoftNIC scheduler is centered around these four primitives to capture common policy patterns.

The scheduling unit of SoftNIC data path execution is a traffic *class*, each of whose definition is flexible and not dictated by SoftNIC. Possible class definitions include: input/output port, VM, tenant, protocol, L7 application, VLAN tag, IP prefix, and so on. The operator determines the scheduling discipline for traffic classes: *e.g.,* "`C1` has higher priority than `C2`" by setting appropriate scheduling parameters.

In SoftNIC, every packet is mapped to one (and only one) of the traffic classes at any given time. The initial mapping of a packet to a class is "given" to SoftNIC. A port consists of a set of input queues, each of which is mapped to a traffic class. Effectively, the peer (for a vport) or the hardware NIC (for a pport) declares the initial class of each packet by pushing the packet into its corresponding queue. The class of a packet may change on the fly; in the dataflow graph, the operator can specify *transformer edges*, which can associate a new class to packets that flow along the edge. Each transformer edge has a map $class_{old} \rightarrow class_{new}$. Class transformation is useful in cases where packet classes cannot be predetermined externally: e.g., (i) a hardware NIC has limited classification capability for incoming packets, or (ii) traffic classes are defined as output ports or input/output port pairs, so the class of a packet is logically unknown (thus a "unspecified" class) until its output port has been determined by a switch/classifier module.

Scheduling parameters are specified as a *class tree*, which is a hierarchical structure of traffic classes. For example, Figure 4(b) shows the class tree for the previous policy. The scheduler starts by examining the children of `root` and proceeds recursively until it finds an eligible leaf. First, assuming neither has exceeded its 5 Gbps limit, the scheduler chooses between `VM1` and `VM2` (they both have the same priority), using a weighted fair queuing mechanism. Without loss of generality, let us assume that `VM1` is chosen. Since `C1` has higher priority than `C2`, the scheduler will pick `C1`, unless it has exceeded its 1 Gbps limit or no packet is pending. Packet processing for `C1` begins by dequeuing a packet from one of its input queues. SoftNIC measures processor time and bandwidth usage during packet processing. Usage accounting is again recursively done for `C1`, `VM1`, and `root`. We generalize this scheduling scheme and provide implementation details in §4.4.

The operator specify policies by providing a class tree, as shown in Figure 4(b). It is straightforward to translate the example policy into the corresponding class tree. For more complex policies, manual translation might be harder. For this case, we plan to develop a controller that compiles a policy specified in a high-level language to SoftNIC configuration (a dataflow graph and its class tree). Designing and implementing such a language and a compiler are left to future work. The controller is also responsible for performing admission control, which ensures that the number of applications using SoftNIC does not exceed the available resources. Admission control can also be used in conjunction with priority scheduling to guarantee minimum bandwidth and processor allocations for applications.

4. Implementation Details

10/40 G Ethernet has become the norm for datacenter servers and NFV applications. In order to keep up with such high-speed links, SoftNIC needs to process (tens of) millions of packets per second with minimal packet processing overheads. Latency is also critical; recent advances in network hardware (e.g., [17]) and protocol design (e.g., [7]) have allowed microsecond-scale in-network latency, effectively shifting the bottleneck to end-host software [62]. SoftNIC should therefore incur minimal processing delay and jitter.

Meeting both these performance requirements and our design goals is challenging. In this section, we describe the implementation details of SoftNIC, with emphasis on its performance-related aspects.

4.1 Overview

The SoftNIC prototype is implemented in 14k lines of C code, running with unmodified Linux and QEMU/KVM. We expect supporting other operating systems or virtualization platforms would be straightforward. To simplify development for us and module developers, SoftNIC runs as a user-mode program on the host. This has performance implications; in our system, the minimum cost of a user-user (between an application process and SoftNIC) context switch is 3 µs, while it is only 0.1 µs for a user-kernel mode switch. SoftNIC uses one or more dedicated cores to eliminate the costs of context switching, as explained below.

SoftNIC runs a separate control thread alongside worker threads. The control thread communicates with an external controller via UNIX or TCP sockets. The control channel supports various message formats, namely binary, type-length-value, and JSON [13], to satisfy the performance and programmability requirements of operators.

4.2 Core Dedication

SoftNIC runs on a small number of dedicated cores—as will be shown later, one core is enough for typical workloads—for predictable, high performance. The alternative, running SoftNIC threads on the same cores as applications (thus on every core) is not viable for us. Since SoftNIC is a user-mode

program, OS process scheduling can unexpectedly introduce up to several milliseconds of delay under high load. This level of jitter would not allow us to provide (sub)microsecond-scale latency overhead.

We dedicate cores by setting core affinity for SoftNIC worker threads and prevent the host kernel from scheduling other system tasks on SoftNIC cores with the `isolcpus` Linux kernel parameter. In addition to reducing jitter, core dedication has three additional benefits for SoftNIC: (i) context switching costs are eliminated; (ii) processor cache is better utilized; (iii) inter-core synchronization among SoftNIC threads is cheap as it involves only a few cores.

Core dedication allows us to make another optimization: we can utilize busy-wait polling instead of interrupts, to reduce latency further. A recent study reports 5–75 µs latency overheads per interrupt, depending on the processor power management states at the moment [17]. In contrast, with busy-wait polling, the current SoftNIC implementation takes less than 0.02 µs for a vport and 0.03 µs for a pport to react to a new packet. Polling leads to a small increase in power consumption when the system is idle: our system roughly consumes an additional 3–5 watts per idle-looping core.

We do not dedicate a core to the control thread, since control-plane operations are relatively latency-insensitive. The control thread performs blocking operations on sockets, thus consuming no CPU cycles when idle.

4.3 Pipeline Components

Physical Ports (pports): We build on the Intel Data Plane Development Kit (DPDK) 1.8.0 [26] library for high-performance packet I/O. We chose DPDK over other alternatives [18, 21, 59] for two reasons: (i) it exposes hardware NIC features besides raw packet I/O; and (ii) it allows direct access to the NIC hardware without any kernel intervention on the critical path.

Each pport is associated with two module instances. **pport_out** sends packets to the hardware NIC after translating packet metadata into hardware-specific offloading primitives. SoftNIC provides an interface for feature modules to discover the capabilities of each pport, so that modules can make decision about whether a feature can be partially or fully offloaded to hardware. **pport_in** receives incoming packets and translates their hardware-offload results into metadata that can be used by other modules.

Virtual Ports (vports): A vport has a set of RX and TX queues. Each queue contains two one-way ring buffers; one for transmitting packet buffers and the other for receiving completion notification (so that the sender can reclaim the packet buffers). The ring buffers provide lock-free operations for multiple consumer/producer cores [44], to minimize inter-core communication overheads. A vport's ring buffers are allocated in a single contiguous memory region that is shared between SoftNIC and the peer. Memory sharing is done using `mmap()` for host peers and IVSHMEM [40] for VM guest peers. When no interrupts are involved, communication via shared memory allows SoftNIC to provide VM peers and host peers similar performance for packet exchange.

For conventional TCP/IP applications, we implement a device driver as a Linux kernel module that can be used by either hosts or guests. We expect that porting this device driver to other operating systems will be straightforward. The driver exposes a vport as a regular Ethernet adapter to the kernel. No modifications are required in the kernel network stack and applications. Our implementation is similar to virtio [63], but we support not only guest VMs but also the host network stack.

For kernel-bypass applications that implement their own specialized/streamlined network stack (*e.g.,* [9, 28, 42, 51]), we provide a user-level library that allows applications to directly access vport queues, supporting zero copy if desired. In contrast, vport peering with the kernel device driver requires copying the packet data; providing zero-copy support for the kernel would require non-trivial kernel modifications, which are beyond the scope of this work.

When packets are transmitted from SoftNIC to a peer, SoftNIC notifies the peer via inter-core interrupts. This notification is not necessary when a peer sends packet because SoftNIC performs polling. The peer can disable interrupts temporarily (to avoid receive livelock [48]) or permanently (for pure polling [14]).

NIC Feature Modules: To test and evaluate the effectiveness of our framework, we implemented a number of NIC features that are commonly used in datacenter servers: checksum offloading, TCP segmentation/reassembly and VXLAN tunneling (§6.1), rate limiter (§6.2), flow steering (§6.3), stateless/stateful load balancer (§6.4), time stamping, IP forwarding, link aggregation, and switching. Our switch module implements a simplified OpenFlow [43] switch on top of MAC-learning Ethernet bridging.

4.4 SoftNIC Scheduler

The basic unit of scheduling in SoftNIC is a traffic class (§3.4). Each class is associated with a set of queues (§4.6) and a per-core timer. Packets belonging to the class are enqueued on one of these queues before processing. The per-core timer is used to schedule deferred processing that can be used for a variety of purposes, *e.g.,* flushing reassembled TCP packets (LRO), interrupt coalescing, periodically querying hardware performance counter, scheduling packet transmission, etc. The scheduler is responsible for selecting the next class and initiating processing for this class.

SoftNIC scheduling parameters are provided to the scheduler as a class tree (§3.4). A class tree is a hierarchical tree, whose leaves map to individual traffic classes while non-leaf nodes represent class groups. Each class group is a recursive combination of classes or other class groups. The scheduling discipline for each node in the class tree is specified in terms

Algorithm 1: Recursively traverse class tree to pick the next traffic class to service.

```
1 Node Pick(n)
2     if n is leaf then
3         return n ; // We found a traffic class.
4     else
5         maxp ← max(n.children.priority);
6         group ← n.children.filter(
7                         priority = maxp);
8         next ← StrideScheduler (group);
9         return Pick (next);
```

of a 5-tuple $\langle priority, limit_{bw}, limit_{cpu}, share, share\_type \rangle$, where each element specifies:

$priority$: strict priority among its all siblings
$limit_{bw}$: limit on the throughput (bits/s)
$limit_{cpu}$: limit on processor time (cycles/s)
$share$: share relative to its siblings with the same priority
$share\_type$: the type of proportional share: bits or cycles

The SoftNIC scheduling loop proceeds in three steps:

Step 1: Pick next class. The scheduler chooses the next class to service by recursively traversing the class tree starting at the root by calling the `pick` function (Algorithm 1) on the tree root. Given a node n, the `pick` function first checks (line 2) whether n is a leaf node (*i.e.,* a traffic class) in which case `pick` returns n. If n is not a leaf (and is thus a class group), $pick$ find the highest priority level among n's children ($maxp$, line 5) and then finds the subset of its children assigned this priority ($group$, line 6). Finally, it uses stride scheduling [73] to select a child ($next$) from this subset (line 8) and returns the result of calling itself recursively on $next$ (line 9).

Step 2: Servicing the class. Once `pick` has returned a class c to the scheduler, the scheduler checks if c has any pending timer events. If so, the scheduler runs the deferred processing associated with the timer. If no timer events are pending, the scheduler dequeues a batch of packets (§4.5) from one of the classes queues (we use round-robin scheduling between queues belonging to the same class) and calls the receive handler for the first module—a port module or a module next to a transformer edge—in the packet processing pipeline (§3.3) for c. Packet processing continues until all packets in the batch have been either enqueued or dropped. Note that within a class, we handle all pending timer events before processing any packets, *i.e.,* timer events are processed at a higher priority, so that we can get high-accuracy for packet transmission scheduling, as shown in §6.2.

Step 3: Account for resource usage. Once control returns to the scheduler, it updates the class tree to account for resources (both processor and bandwidth) used in processing class c. This accounting is done with the class c and all of its parents up to the root of the class tree. During this update, the scheduler also temporarily prunes the tree of any nodes that have exceeded their resource limits. This pruning (and grafting) is done with the token bucket algorithm.

As stated in §3.4, sometimes assigning a packet to the correct class might require SoftNIC to execute one or more classification modules before sending a packet out a transformer edge. For accurate resource accounting in this case, SoftNIC associates an implicit queue with each transformer edge so that packets whose class has been changed are enqueued and wait for the new class to be scheduled before further processing.

4.5 Packet Buffers and Batched Packet Processing

Packet Buffer: SoftNIC extends the DPDK's packet buffer structure (`rte_mbuf`) [26] by reserving an area in each buffer to store metadata. We rely heavily on scattered packet buffers (*i.e.,* non-contiguous packet data) to avoid packet copy, for operations such as segmentation/reassmembly offloading and switch broadcasting.

Packet Metadata: A naive implementation of metadata (§3.3), *e.g.,* using a hashmap of $string \rightarrow value$, can have significant performance penalty, as in SoftNIC tens of modules may process tens of millions of packets per second. To avoid the performance penalty, previous software packet processing systems use either a static set of metadata fields as `struct` fields (*e.g.,* BSD mbuf [45]) or per-packet scratchpad where all modules have to agree on how each byte should be partitioned and reused (*e.g.,* Click [35]). Both approaches are not only inefficient in space (thus increasing CPU cache pressure) and inextensible, but also are error-prone.

SoftNIC achieves extensibility (modules can introduce new fields without compile-time agreements) and space efficiency (unused bytes can be safely reclaimed)—both with minimal performance overheads—by taking advantage of the explicit declaration of metadata fields by each module (§3.3). Since all fields are known ahead of time, offsets for metadata fields in a pipeline can be precomputed, and SoftNIC provides the offsets to every module during the pipeline (re)configuration phase. Subsequently, modules can access a metadata field by reading from its corresponding offset from the packet buffer.

Pervasive Batching: Packets in SoftNIC are processed in batches, *i.e.,* packet I/O from/to ports is done in batches, and packet processing modules operate on batches of packets, rather than individual packets. Batching is a well-known technique for improving code/data locality and amortizing overheads in software packet processing [9, 14, 21, 33, 59]. Specifically, batching amortizes the cost of (i) remote cache access for vports, (ii) hardware access over the PCIe bus for pports, and (iii) virtual function calls for module execution. SoftNIC uses a dynamic batch size that is adaptively varied to minimize the impact on latency as in IX [9]: the batch size grows (to a cap) only when SoftNIC is overloaded and there is queue buildup for incoming packets.

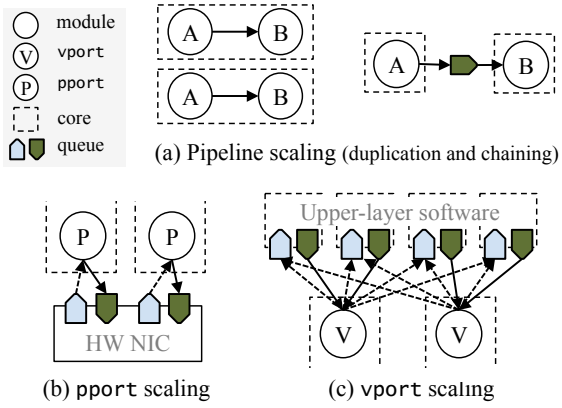


Figure 5: SoftNIC multi-core scaling strategies. The example assumes two SoftNIC cores and four cores for the peer.

When combined with the modular pipeline of SoftNIC, batch processing provides additional performance benefits. Since the processing of a packet batch is naturally "staged" across modules, cache miss or register dependency stalls during processing one packet can be likely hidden by processing another packet in the batch on an out-of-order execution processor [31].

A packet batch is simply represented as an array of packet buffer pointers. When packets in a batch need to take different paths in the pipeline (*e.g.,* classifier or switch), the module can simply split the batch by moving pointers from one array to another without incurring significant overheads.

4.6 Multi-Core Scaling

While SoftNIC running on a single core provides enough horse power to drive the common network workload of a datacenter server (a 40 G link or higher with a typical packet size distribution, see §5.2), SoftNIC can scale on multiple cores to support more challenging workloads. Figure 5 illustrates how it is done. SoftNIC provides two means to scale the packet processing pipeline. The default scheme is *duplication*, each core runs the identical pipeline of modules in parallel. The other scheme is *chaining*, where the pipeline is partitioned with in-memory queue connection. One scheme is not always better than the other, as the resulting performance is highly dependent on: cache usage of modules, synchronization overhead, number of SoftNIC cores, etc. SoftNIC currently needs manual configuration for the chaining scheme.

When a pport is duplicated across cores, SoftNIC leverages the multi-queue functionality of the hardware NIC as shown in Figure 5(b). Each SoftNIC core runs its own RX/TX queue pair, without incurring any cache coherence traffic among SoftNIC cores. Similarly, SoftNIC creates multiple queue pairs for a vport so that the peer itself can linearly scale. By partitioning the incoming (from the viewpoint of SoftNIC) queues of vports and pports, SoftNIC preserves in-order packet processing on a flow basis, provided that peers and hardware NICs do not interleave a flow across multiple queues.

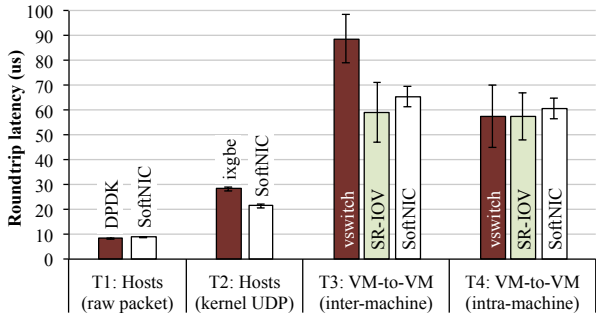


Figure 6: Round-trip latency between two application processes.

5. Performance Evaluation

The main promise of SoftNIC is to provide a flexible framework for implementing various NIC features, without sacrificing performance. In this section, we focus on the overheads of SoftNIC framework itself, without considering the performance of individual feature modules. We quantify the overheads by measuring how fast SoftNIC performs as a NIC, in terms of end-to-end latency (§5.1), throughput and multi-core scalability (§5.2).

Experiment setting: We use two directly connected servers, each equipped with two Intel Xeon 2.6 GHz E5-2650v2 processors (16 cores in total), 128 GB of memory, and four Intel 82599 10 GbE ports with an aggregate bandwidth of 40 Gbps. We disable the CPU's power management features (C/P-states) for reproducible latency measurement [17]. We use unmodified Linux 3.14.16 (for both host and guest), QEMU/KVM 1.7.0 for virtualization, and ixgbe 3.19.1-k NIC device driver in the test cases where SoftNIC is not used.

5.1 End-to-End Latency

Figure 6 shows the end-to-end, round-trip latency measured with UDP packets (TCP results are similar). In T1, the application performs direct packet I/O using a dedicated hardware NIC with DPDK (8.22 µs) or a vport (8.82 µs), thus bypassing the kernel protocol stack. SoftNIC adds a small latency overhead of 0.6 µs per round trip, or 0.15 µs per direction per server (one round trip involves SoftNIC four times). We posit that the advantage of using SoftNIC—allowing multiple kernel-bypass and conventional TCP/IP applications to coexist without requiring exclusive hardware access—outweighs the costs.

T2–4 shows the latency for conventional applications when using the kernel TCP/IP support, measured with the netperf UDP_RR [1] test. Surprisingly, for the non-virtualization case (T2), the baseline (ixgbe) latency of 28.3 µs was higher than SoftNIC's 21.4 µs. This is due to a limitation of 82599; when LRO is enabled, the NIC buffers incoming packets (thus inflating latency), even if they do not require reassembly. When LRO is disabled, latency decreases to 21.1 µs, which comes very close to that of SoftNIC.

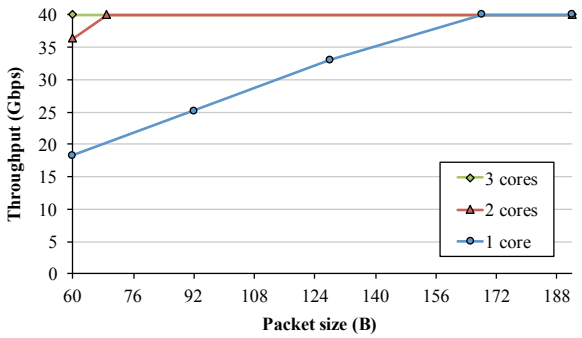


Figure 7: SoftNIC throughput and multi-core scalability.

With server virtualization, T3[5], we compare SoftNIC with virtual switching in the host network stack (vswitch) and hardware NIC virtualization (SR-IOV). As expected, using a vswitch incurs significant latency overhead because packets go through the slow host network stack four times (vs. two times in T4) in a single round trip. For SR-IOV and SoftNIC, packets bypass the host network stack. When compared with the bare-metal case (T2), both exhibit higher latency, because packet I/O interrupts to the VMs have to go through the host kernel. We expect that the latency for VMs using SR-IOV and SoftNIC will be close to bare-metal with recent progresses in direct VM interrupt injection [19, 23], and the latency of SoftNIC will remain comparable to hardware NIC virtualization.

In summary, the results confirm that SoftNIC does not add significant overhead to end-to-end latency for both bare-metal and virtual machines. We conclude that SoftNIC is viable even for latency-sensitive applications.

5.2 Throughput

We demonstrate that SoftNIC sustains enough throughput on a single core to support high-speed links and scales well on multiple cores. For the experiment, we could not use conventional TCP/IP applications due to the low performance (roughly 700 kpps per core) of Linux TCP/IP stack, which is not enough to saturate SoftNIC. Instead, we write a simple application that performs minimal forwarding (packets received from a port are forwarded to another port) with the SoftNIC vport raw packet interface. The application runs on a single core and is never the bottleneck since effectively the heavy-weight packet I/O is offloaded to SoftNIC.

Figure 7 depicts the bidirectional throughput with varying packet sizes as we increase the number of SoftNIC cores. In the worst case with 60 B minimum-sized Ethernet packets, SoftNIC on a single core can sustain about 27.2 Mpps (18.3 Gbps) for both RX and TX directions simultaneously, fully saturating our 40 G link capacity with 168 B packets or larger. With multiple SoftNIC cores, the throughput almost scales linearly. Considering the average packet size of 850 B in datacenters [10], we conclude that *SoftNIC on a single core provides enough performance to drive a 40 G link* in realistic scenarios. For higher speed links (e.g., 100 Gbps for end hosts in the future) or applications with emphasis on small-packet performance (e.g., VoIP gateway), SoftNIC needs to run on multiple cores.

Given that the number of cores in a typical datacenter server is large (16-32 as of today) and keeps pace with the link speed increase, we expect the required number of SoftNIC cores for future high-speed links will remain relatively small. We also note that this small investment can bring huge net gain—-as we will see in the following macrobenchmarks—because SoftNIC effectively takes over the burden of implementing NIC features from the host network stack, running them with much higher efficiency.

[5] For completeness, we also show T4 where two VMs reside on the same physical machine.

6. Case Studies

SoftNIC provides an effective platform to implement a wide variety of NIC features that are either not readily available or cannot be fully implemented in hardware, while providing higher performance than achievable by a software implementation in host network stacks. This section highlights the programmability and performance of SoftNIC with various NIC feature examples.

6.1 Segmentation Offloading for Tunneled Packets

Many NIC features perform protocol-dependent operations. However, the slow update cycle of NIC hardware cannot keep up with the emergence of new protocols and their extensions. This unfortunate gap further burdens already slow host network stacks [54], or restricts the protocol design space (e.g., MPTCP [24, 56] and STT [36]). SoftNIC is a viable alternative to the problem as it can be easily reprogrammed to support new protocols and their extensions while minimizing performance overheads.

As an example, we discuss TCP segmentation offloading (TSO for sender-side segmentation and LRO for receiver-side reassembly) over tunneling protocols, such as VXLAN [41], NVGRE [70], and Geneve [20]. These tunneling protocols are used to provide virtual networks for tenants over the shared physical network [36]. Most current 10 G NICs do not support segmentation offloading for inner TCP frames, since they do not understand the encapsulation format. While 40 G NICs have begun supporting segmentation for tunneled packets, VM traffic still has to go through the slow host network stack since NICs lack support for encapsulating packets itself and IP forwarding (for MAC-over-IP tunneling protocols, *e.g.*, VXLAN and Geneve).

We compare TCP performance over VXLAN on two platforms: the host network stack (Linux) and SoftNIC. Adding VXLAN support to the regular (non-tunneled) TCP TSO/LRO modules in SoftNIC was trivial, requiring only 70 lines of code modification. The results shown in Table 1 clearly demonstrate that segmentation *onloading* on SoftNIC achieves much higher throughput at lower CPU overhead.

	Throughput	Sender CPU usage (%)					Receiver CPU usage (%)				
		SoftNIC	QEMU	Host	Guest	Total	SoftNIC	QEMU	Host	Guest	Total
Linux	**14.4** Gbps	-	-	475.2	242.6	**717.8**	-	-	771.7	387.2	**1158.9**
SoftNIC	**40.0** Gbps	200.0	66.6	9.8	186.1	**462.4**	200.0	80.5	21.9	179.0	**481.4**

Table 1: TCP throughput and CPU usage breakdown over the VXLAN tunneling protocol. 32 TCP connections are generated with the netperf TCP_STREAM test, between two VMs on separate physical servers. The current implementation of SoftNIC injects packet I/O interrupts through QEMU; we expect that the QEMU overheads be eliminated by injecting interrupts directly into KVM. Overall, SoftNIC outperforms the host network stack, by a factor of 4.3 for TX and 6.7 for RX in terms of throughput per CPU cycle.

Per-flow rate (Mbps)	Number of flows	SENIC (µs)	SoftNIC (µs)
1	500	7.1	1.4
1	4096	N/A	2.0
10	1	0.23	1.4
10	10	0.24	1.4
10	100	1.3	1.6
100	1	0.087	1.1
100	10	0.173	1.4
1000	1	0.161	1.1
1000	3	0.191	1.1

Table 2: Accuracy comparison between SoftNIC and SENIC rate limiters, with the standard deviation of IPGs. The SENIC numbers are excerpted from their NetFPGA implementation [55, Table 3].

SoftNIC running on two cores[6] was able to saturate 40 Gbps, while Linux's throughput maxed out at 14.4 Gbps even when consuming more CPU cycles. In terms of CPU efficiency, *i.e.,* bits per cyle, SoftNIC is $4.3\times$ and $6.7\times$ more efficient than Linux for TX and RX respectively.[7]

We note that there is a widely held view that hardware segmentation offloading is indispensable for supporting high-speed links [36, 54–56]. Interestingly, our results show that software approaches to TSO/LRO can also support high-speed links, as long as the software is carefully designed and implemented.

6.2 Scalable Rate Limiter

Many recent research proposals [8, 30, 53, 61, 67] rely on endhost-based rate enforcement for fair and guaranteed bandwidth allocation in a multi-tenant datacenter. These systems require a large number ($<$ 1,000s) of rate limiters, far beyond the capacity of commodity NICs (*e.g.,* Intel's 82599 10 G NIC supports up to 128 rate limiters, and XL710 40 G NIC supports 384). Software rate limiters in host network stacks (*e.g.,* Linux tc [25]) can scale, but their high CPU overheads hinder support for high-speed links and precise rate enforcement [55]. We show that rate enforcement with SoftNIC achieves both scalability and accuracy for high-speed links.

We conduct an experiment with 1,000 concurrent UDP flows, whose target rate ranges between 1–11 Mbps with 10 kbps steps. The measured accuracy for each flow—the ratio of the measured rate to the target—is higher than 0.9999.

Another important metric is microscopic accuracy, *i.e.,* how evenly flows are paced at the packet level, since precise packet burstiness control on short timescales is essential for achieving low latency in a datacenter [27, 49]. Table 2 shows the standard deviation of inter-packet gaps (IPG) measured at the receiver side. Since we collect IPG measurements on a server, instead of using specialized measurement hardware, our IPG numbers are overestimated due to the receiver's measurement error. Nevertheless, SoftNIC achieves about 1-2 µs standard deviation across a wide range of scenarios.

As a point of comparison[8], we also show the results from SENIC [55] as a state-of-the-art hardware NIC. The hardware-based real-time behavior allows SENIC to achieve a very low standard deviation of 100 ns with 10 or less flows, but its IPG variation increases as the number of flows grows. At the largest data point presented in [55, Table 3], 500 flows at 1 Mbps, the standard deviation of SENIC is 7.1 µs, which is higher than SoftNIC's 1.4 µs. This is because the packet scheduler in SENIC performs a linear scan through the token bucket table on SRAM, requiring 5 clock cycles per active flow. While we are unaware of whether this linear scanning is due to hardware design limitations or implementation difficulties, SoftNIC can achieve near-constant time with a time-sorted data structure whose software implementation is trivial. We conclude that SoftNIC scales well to thousands of rate limiters, yet is fast and accurate enough for most practical purposes.

6.3 Packet Steering for Flow Affinity

Ensuring flow affinity—collocating TCP processing and application processing on the same core is known to be crucial for the performance TCP-based server applications. Existing software solutions to flow affinity [22, 50] restrict the application programming model, incurring multiple limitations: each application thread needs to be pinned to a core, connections should not be handed over among cores, and applications may require non-trivial code modifications.

[6] Unlike the regular TCP TSO/LRO with which SoftNIC can saturate 40 Gbps on a single core, we needed two SoftNIC cores for VXLAN. This is because SoftNIC has to calculate checksum for the inner TCP frame in software; Intel 82599 NICs do not support checksum offloading over an arbitrary payload range.

[7] The asymmetry between TX and RX is due to the fact that the host network stack implements TCP over VXLAN segmentation for the sender side [74], but reassembly at the receiver side is currently not supported, thus overloading the host and VM network stacks with small packets.

[8] We omit the Linux tc and Intel 82599 NIC results, as they fail to sustain the aggregate throughput or do not support enough rate limiters, respectively.

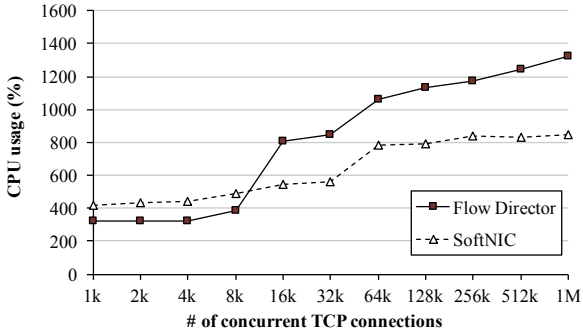


Figure 8: System-wide CPU usage to sustain 1M dummy transactions per second. The bump between 32k and 64k is due to additional TCP packets caused by delayed ACK. With SoftNIC, the system scales better with high concurrency.

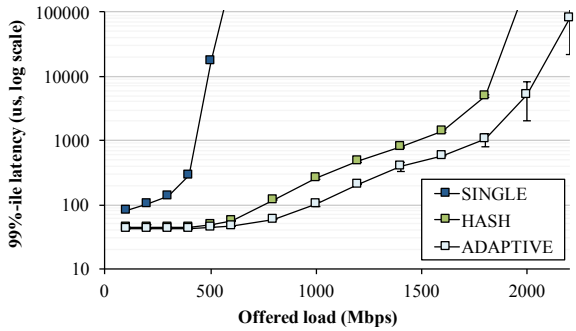


Figure 9: Snort 99%-ile tail latency with a single instance without SoftNIC (SINGLE), four instances with static (HASH) and dynamic (ADAPTIVE) load balancing with SoftNIC. Note that the absolute low throughput is due to the application bottleneck, not SoftNIC.

As a more general solution without these limitations, Intel 82559 NICs support a feature called Flow Director, which maintains a flow-core mapping table so that the NIC can deliver incoming packets to the "right core" for each flow [3]. However, the table size is limited to 8k flow entries to fit in the scarce on-chip memory. Applications that require higher concurrency (*e.g.,* front-end web servers and middleboxes often handle millions of concurrent flows) cannot benefit from this limited flow table. In SoftNIC, we implement a module SoftNIC that provides the same functionality but supports virtually unlimited flow entries by leveraging system memory.

Figure 8 shows the effectiveness of Flow Director and its SoftNIC counterpart. We use a simple benchmark program that exchanges 512 B dummy requests and responses with a fixed rate of 1M transactions per second. We vary the number of concurrent TCP connections and measure the total CPU usage of the system. When there are a small number of concurrent connections, SoftNIC's overhead is slightly higher due to the cost of the dedicated SoftNIC core[9]. Once the number of connections exceeds 8k and the hardware flow table begins to overflow, Flow Director exhibits higher CPU overheads due to increased cache bouncing and lock contention among cores. In contrast, SoftNIC shows a much more gradual increase (due to CPU cache capacity misses) in CPU usage with high concurrency. With 1M connections, for instance, SoftNIC effectively saves 4.8 CPU cores, which is significant given that SoftNIC is a drop-in solution.

6.4 Scaling Legacy Applications

Many legacy applications are still single-threaded, as parallelization may require non-trivial redesign of software. SoftNIC can be utilized to scale single-threaded network applications, given that they do not need state to be shared across cores. We use Snort [2] 2.9.6.1, an intrusion prevention system as an example legacy application to show this scaling. There are two requirements for scaling Snort: (i) the NIC must distribute incoming packets across multiple Snort instances; (ii) moreover, such demultiplexing has to be done on a flow basis, so that each instance correctly performs deep-packet inspection on every flow. While we can meet these requirements with receive-side scaling (RSS), an existing NIC feature in commodity hardware NICs, it is often infeasible; RSS requires that any physical ports used by Snort not be shared with other applications.

The flexible SoftNIC pipeline provides a straightforward mechanism for scaling Snort. SoftNIC provides a backwards-compatible mechanism to distribute traffic from a physical link between multiple instances. To do this, we create a vport for each Snort instance and connect all vports and a pport with a load balancer module. The load balancer distributes flows across vports using the hash of the flow's 5-tuple. With this setup, each Snort instance can transparently receive and send packets through its vport. Figure 9 shows the 99th percentile end-to-end (from a packet generator to a sink) latency, using packet traces captured at a campus network gateway. We find that even at 500 Mbps a single instance of Snort (SINGLE) has a tail latency of approximately 366 ms, while using four Snort instances with our hash-based loadbalancer (HASH) limits latency to 57 µs (a $6,000\times$ improvement).

Furthermore, SoftNIC allows us for rapid prototyping of more advanced load balancing schemes. We implemented an adaptive load balancer, ADAPTIVE, which tracks load at each instance and assigns new flows to the least loaded vport. As compared to HASH, this scheme mitigates transient imbalance among instances, prevents hash-collision based attacks [71], and retains flow stickiness upon dynamic change in number of instances. Our current heuristic estimates the load of each port using "load points"; we assign a vport 10,000 load points for its new flow, 1,000 points per packet, and 1 point per byte. When a new flow arrives, we assign it to the vport which accumulated the fewest load points in the last 1 ms time window. The adaptive load balancer maintains a flow table for flow stickiness of subsequent packets. ADAPTIVE performs significantly better than HASH, improving tail latency by $5\times$ at 1.8 Gbps. We expect that implementing

[9] For a fair comparison, we always account 100% for the SoftNIC case to reflect its busy-wait polling. Throughout the experiment, however, the actual utilization of the SoftNIC core was well below 100%.

such a load balancing scheme in hardware would not be as straightforward; this example demonstrates the programmability of SoftNIC.

7. Related Work

Click: The modular packet processing pipeline of SoftNIC is inspired by the Click modular router [35]. We briefly discuss how we adapt Click's design for SoftNIC. In general, Click's elements are defined in a much more fine-grained manner, *e.g.,* the `Strip` element removes a specified number of bytes from the packet, while SoftNIC modules embody entire NIC functions (*e.g.,* switching is a single module). We chose to go with relatively coarse-grained modules because frequent transitions among modules has a significant impact on performance—small, dynamic modules provide compilers with limited optimization opportunities.

Furthermore, in SoftNIC we assume that each module internally implements its own queues as necessary, whereas Click's scheduling is centered around providing explicit queues in the pipeline. This design choice simplifies our scheduling. The absence of explicit queues greatly streamlines packet processing in SoftNIC; it can simply pick a traffic class and process packets using run to completion, without having to deal with "push" and "poll" calls as in Click. Another advantage of forgoing explicit queues is scalability. For example, in Click, supporting 1,000 rate limiters requires there be 1,000 explicit queues and token bucket elements in the dataflow graph, requiring the scheduler to consider all of these elements individually. In contrast, with the rate limiter of SoftNIC (§6.2), the scheduler only needs to pick a traffic class to serve, simplifying the decision making.

For CPU scheduling, Click executes elements with weighted round-robin, enforcing fairness in terms of the number of executions. This is not a meaningful measure for either bandwidth (packet sizes differ) or processor time (packets may consume vastly different CPU cycles). We extend this scheduling model with explicit traffic class support, fixed priority, hierarchical composition, and accurate resource usage accounting. These extensions enable SoftNIC to provide high-level performance isolation and guarantees across applications.

Potential applications of SoftNIC: SoftNIC can be an effective platform for supporting and enhancing emerging networked systems. We list a few interesting examples: application-aware packet steering [32, 38], software-based RDMA operations with richer semantics [15], packet pacing [27], NFV deployment for public clouds [16], high-performance user-level TCP/IP stack [28, 66], per-packet priority support [7, 46], and TDMA over Ethernet [72]. SoftNIC can also be used as a platform for operating systems that aim to separate network policy enforcement from the data plane, *e.g.,* IX [9] and Arrakis [51]. Complimentary to our work, SoNIC [37] proposes a NIC architecture to provide access to the PHY and MAC layer. One may combine it with SoftNIC for complete software control over the entire stack of NIC functionality.

Ideal hardware NIC model: Although SoftNIC co-exists with existing hardware NICs, an important benefit of our shim-layer approach is that SoftNIC could act as an enabler for improved NIC designs in the long term. While there is growing consensus on exploiting NIC resources [68, 69], the *black-box* (*e.g.,* "do checksum for this SCTP packet") feature interface and implementation of current NIC hardware presents difficulties. We believe that NICs should instead offer *components* (*e.g.,* "calculate checksum for this payload range with CRC32 and update the result here"), so that software can compound such building blocks into complete features. In this way, NICs can provide fine-grained behavior control, better state visibility, future proofness. We expect that the P4 [11] work can be a good starting point for this ideal NIC model.

Alternative approaches: SoftNIC has several benefits over other approaches. Implementing NIC features in legacy network stacks is slow, which is why hardware approaches gained popularity in the first place. Another alternative is to make the NIC hardware more programmable (*e.g.,* FPGA [39, 64] and network processors [65]). However, the degree of programmability is still limited as compared to software on CPUs, and hardware resource limitations still apply. Having a general-purpose processor on the NIC card, as a bump-in-the-wire, would be another option that is similar in spirit to SoftNIC. We argue that SoftNIC's approach (reusing existing server resources) is superior, in terms of efficiency/elasticity of resources and maintaining a global view of system.

8. Conclusion and Future Work

SoftNIC is a programmable platform to augment hardware NICs with software. Built upon many new and existing ideas, SoftNIC provides the performance comparable to hardware and the flexibility of software for implementing sophisticated NIC features, while simultaneously being backwards compatible with existing software and hardware.

In this work, our main focus was on the data plane, assuming an external controller interacts via the control channel that SoftNIC provides. Our next step is to design and implement a sophisticated controller that translates high-level user policies into the SoftNIC mechanisms (the packet processing pipeline and scheduling parameters).

We expect that SoftNIC can be a useful tool for researchers and systems developers. There are multiple ongoing projects that leverage SoftNIC for diverse use cases, including: a new congestion control algorithm relying on accurate timestamps, packet burstiness control for end-to-end latency bound in a multi-tenant datacenter, and a prototype platform for network function virtualization (NFV). We plan on releasing SoftNIC to the wider community as an open source project under a BSD license.

References

[1] Netperf network benchmark software. http://netperf.org.

[2] Snort Intrusion Detection System. https://snort.org.

[3] Intel 8259x 10G Ethernet Controller. Intel 82599 10 GbE Controller Datasheet, 2009.

[4] Product Brief: Intel Ethernet Controller XL710 10/40 GbE. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xl710-10-40-gbe-controller-brief.pdf, 2014.

[5] ABU-LIBDEH, H., COSTA, P., ROWSTRON, A., O'SHEA, G., AND DONNELLY, A. Symbiotic Routing in Future Data Centers. In *ACM SIGCOMM* (2010).

[6] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI* (2012).

[7] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal Near-Optimal Datacenter Transport. In *ACM SIGCOMM* (2013).

[8] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards Predictable Datacenter Networks. In *ACM SIGCOMM* (2011).

[9] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI* (2014).

[10] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review 40*, 1 (2010), 92–99.

[11] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM CCR 44*, 3 (2014), 87–95.

[12] BRISCOE, B. Tunnelling of Explicit Congestion Notification. RFC 6040, November 2010.

[13] CROCKFORD, D. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, July 2006.

[14] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP* (2009).

[15] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast Remote Memory. In *USENIX NSDI* (2014).

[16] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. Network Functions Virtualisation (NFV). http://goo.gl/x1ueJL.

[17] FLAJSLIK, M., AND ROSENBLUM, M. Network Interface Design for Low Latency Request-Response Protocols. In *USENIX ATC* (2013).

[18] FUSCO, F., AND DERI, L. High Speed Network Traffic Analysis with Commodity Multi-Core Systems. In *ACM IMC* (2010).

[19] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: Bare-Metal Performance for I/O Virtualization. In *ACM ASPLOS* (2012).

[20] GROSS, J., SRIDHAR, T., GARG, P., WRIGHT, C., AND GANGA, I. Geneve: Generic Network Virtualization Encapsulation. IETF draft, http://tools.ietf.org/html/draft-gross-geneve-00.

[21] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-Accelerated Software Router. In *ACM SIGCOMM* (2010).

[22] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX OSDI* (2012).

[23] HAR'EL, N., GORDON, A., LANDAU, A., BEN-YEHUDA, M., TRAEGER, A., AND LADELSKY, R. Efficient and Scalable Paravirtual I/O System. In *USENIX ATC* (2013).

[24] HONDA, M., NISHIDA, Y., RAICIU, C., GREENHALGH, A., HANDLEY, M., AND TOKUDA, H. Is It Still Possible to Extend TCP? In *ACM IMC* (2011).

[25] HUBERT, B. Linux Advanced Routing and Traffic Control. http://www.lartc.org.

[26] INTEL. Data Plane Development Kit (DPDK). http://dpdk.org.

[27] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message completion time in the cloud. Tech. Rep. MSR-TR-2013-95, September 2013.

[28] JEONG, E., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems.

[29] JEYAKUMAR, V., ALIZADEH, M., GENG, Y., KIM, C., AND MAZIÈRES, D. Millions of little minions: Using packets for low latency network programming and visibility. In *ACM SIGCOMM* (2014).

[30] JEYAKUMAR, V., ALIZADEH, M., MAZIERES, D., PRABHAKAR, B., KIM, C., AND GREENBERG, A. EyeQ: Practical network performance isolation at the edge. In *USENIX NSDI* (2013).

[31] KALIA, A., ZHOU, D., KAMINSKY, M., AND ANDERSEN, D. G. Raising the Bar for Using GPUs in Software Packet Processing. In *USENIX NSDI* (2015).

[32] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: Predictable Low Latency for Data Center Applications. In *ACM SoCC* (2012).

[33] KIM, J., HUH, S., JANG, K., PARK, K., AND MOON, S. The Power of Batching in the Click Modular Router. In *ACM APSys* (2012).

[34] KIM, J., JANG, K., LEE, K., MA, S., SHIM, J., AND MOON, S. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *ACM EuroSys* (2015).

[35] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Trans. Comput. Syst. 18*, 3 (Aug. 2000), 263–297.

[36] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., GUDE, N., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H. L., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network virtualization in multi-tenant datacenters. In *USENIX NSDI* (2014).

[37] LEE, K.-S., WANG, H., AND WEATHERSPOON, H. SoNIC: Precise Realtime Software Access and Control of Wired Networks. In *USENIX NSDI* (2013).

[38] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage.

[39] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA: An Open Platform for Gigabit-Rate Network Switching and Routing. In *IEEE MSE* (2007).

[40] MACDONELL, C. *Shared-Memory Optimizations for Virtual Machines*. PhD thesis, University of Alberta, 2011.

[41] MAHALINGAM, M., DUTT, D., DUDA, K., AGARWAL, P., KREEGER, L., SRIDHAR, T., BURSELL, M., AND WRIGHT, C. VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. IETF draft, http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-09.

[42] MARINOS, I., WATSON, R. N., AND HANDLEY, M. Network Stack Specialization for Performance. In *ACM SIGCOMM* (2014).

[43] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR 38*, 2 (2008), 69–74.

[44] MICHAEL, M. M., AND SCOTT, M. L. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *ACM PODC* (1996).

[45] MILEKIC, B. Network Buffer Allocation in the FreeBSD Operating System. In *BSDCAN* (2004).

[46] MITTAL, R., SHERRY, J., RATNASAMY, S., AND SHENKER, S. Recursively Cautious Congestion Control. In *USENIX NSDI* (2014).

[47] MOGUL, J. C., MUDIGONDA, J., SANTOS, J. R., AND TURNER, Y. The NIC Is the Hypervisor: Bare-Metal Guests in IaaS Clouds. In *USENIX HotOS* (2013).

[48] MOGUL, J. C., AND RAMAKRISHNAN, K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems 15*, 3 (1997), 217–252.

[49] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A centralized "zero-queue" datacenter network. In *ACM SIGCOMM* (2014).

[50] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving Network Connection Locality on Multicore Systems. In *Proc. of ACM EuroSys* (2012).

[51] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. Tech. Rep. UW-CSE-13-10-01, University of Washington, May 2014.

[52] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The Design and Implementation of Open vSwitch. In *USENIX NSDI* (2015).

[53] POPA, L., YALAGANDULA, P., BANERJEE, S., MOGUL, J. C., AND SANTOS, Y. T. J. R. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *ACM SIGCOMM* (2013).

[54] PORTER, R. N. M. G., AND VAHDAT, A. FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers. In *ACM CoNEXT* (2013).

[55] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for End-Host Rate Limiting. In *USENIX NSDI* (2014).

[56] RAICIU, C., PAASCH, C., BARRE, S., FORD, A., HONDA, M., DUCHENE, F., BONAVENTURE, O., HANDLEY, M., ET AL. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *USENIX NSDI* (2012).

[57] RAM, K. K., MUDIGONDA, J., COX, A. L., RIXNER, S., RANGANATHAN, P., AND SANTOS, J. R. sNICh: Efficient Last Hop Networking in the Data Center. In *ACM/IEEE ANCS* (2010).

[58] RAMAKRISHNAN, K., FLOYD, S., AND BLACK, D. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001.

[59] RIZZO, L. netmap: a novel framework for fast packet I/O. In *USENIX ATC* (2012).

[60] RIZZO, L., AND LETTIERI, G. VALE: a Switched Ethernet for Virtual Machines. In *ACM CoNEXT* (2012).

[61] RODRIGUES, H., SANTOS, J. R., TURNER, Y., SOARES, P., AND GUEDES, D. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *USENIX WIOV* (2011).

[62] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's Time for Low Latency. In *USENIX HotOS* (2011).

[63] RUSSELL, R. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *ACM Operating Systems Review 42*, 5 (2008), 95–103.

[64] SHAFER, J., AND RIXNER, S. RiceNIC: A Reconfigurable Network Interface for Experimental Research and Education. In *ACM ExpCS* (2007).

[65] SHAH, N. Understanding Network Processors. Master's thesis, University of California, Berkeley, 2001.

[66] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *USENIX ATC* (2010).

[67] SHIEH, A., KANDULA, S., GREENBERG, A. G., KIM, C., AND SAHA, B. Sharing the data center network. In *USENIX NSDI* (2011).

[68] SHINDE, P., KAUFMANN, A., KOURTIS, K., AND ROSCOE, T. Modeling NICs with Unicorn. In *ACM PLOS* (2013).

[69] SHINDE, P., KAUFMANN, A., ROSCOE, T., AND KAESTLE, S. We Need to Talk About NICs. In *USENIX HotOS* (2013).

[70] SRIDHARAN, M., GREENBERG, A., WANG, Y., GARD, P., N.VENKATARAMIAH, DUDA, K., GANGA, I., LIN, G., PEARSON, M., THALER, P., AND TUMULURI, C. NVGRE: Network Virtualization using Generic Routing Encapsulation. IETF draft, http://tools.ietf.org/html/draft-sridharan-virtualization-nvgre-04.

[71] TOBIN, R. J., AND MALONE, D. Hash Pile Ups: Using Collisions to Identify Unknown Hash Functions. In *IEEE CRiSIS* (2012).

[72] VATTIKONDA, B. C., PORTER, G., VAHDAT, A., AND SNOEREN, A. C. Practical TDMA for Datacenter Ethernet. In *ACM EuroSys* (2012).

[73] WALDSPURGER, C. A., AND WEIHL, W. E. *Stride Scheduling: Deterministic Proportional Share Resource Management*. Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.

[74] XU, X. Generic Segmentation Offload. http://lwn.net/Articles/188489/.