SoNIC: Precise Realtime Software Access and Control of Wired Networks

Ki Suh Lee, Han Wang, Hakim Weatherspoon
Computer Science Department, Cornell University
kslee,hwang,hweather@cs.cornell.edu

Abstract
The physical and data link layers of the network stack contain valuable information. Unfortunately, a systems programmer would never know. These two layers are often inaccessible in software and much of their potential goes untapped. In this paper we introduce SoNIC, Software-defined Network Interface Card, which provides access to the physical and data link layers in software by implementing them in software. In other words, by implementing the creation of the physical layer bitstream in software and the transmission of this bitstream in hardware, SoNIC provides complete control over the entire network stack in realtime. SoNIC utilizes commodity off-the-shelf multi-core processors to implement parts of the physical layer in software, and employs an FPGA board to transmit optical signal over the wire. Our evaluations demonstrate that SoNIC can communicate with other network components while providing realtime access to the entire network stack in software. As an example of SoNIC's fine-granularity control, it can perform precise network measurements, accurately characterizing network components such as routers, switches, and network interface cards. Further, SoNIC enables timing channels with nanosecond modulations that are undetectable in software.

1 Introduction

The physical and data link layers of the network stack offer untapped potential to systems programmers and network researchers. For instance, access to these lower layers can be used to accurately estimate available bandwidth [23, 24, 32], increase TCP throughput [37], characterize network traffic [19, 22, 35], and create, detect and prevent covert timing channels [11, 25, 26]. In particular, idle characters that only reside in the physical layer can be used to accurately measure interpacket delays. According to the 10 Gigabit Ethernet (10 GbE) standard, the physical layer is *always* sending either data or idle characters, and the standard requires at least 12 idle characters (96 bits) between any two packets [7]. Using these physical layer (PHY[1]) idle characters for a measure of interpacket delay can increase the precision of estimating available bandwidth. Further, by controlling interpacket delays, TCP throughput can be increased by reducing bursty behavior [37]. Moreover, capturing these idle characters from the PHY enables highly accurate traffic analysis and replay capabilities. Finally, fine-grain control of the interpacket delay enables timing channels to be created that are potentially undetectable to higher layers of the network stack.

Unfortunately, the physical and data link layers are usually implemented in hardware and not easily accessible to systems programmers. Further, systems programmers often treat these lower layers as a black box. Not to mention that commodity network interface cards (NICs) do not provide nor allow an interface for users to access the PHY in any case. Consequently, operating systems cannot access the PHY either. Software access to the PHY is only enabled via special tools such as BiFocals [15] which uses physics equipment, including a laser and an oscilloscope.

As a new approach for accessing the PHY from software, we present SoNIC, Software-defined Network Interface Card. SoNIC provides users with unprecedented flexible realtime access to the PHY from software. In essence, all of the functionality in the PHY that manipulate bits are implemented in software. SoNIC consists of commodity off-the-shelf multi-core processors and a field-programmable gate array (FPGA) development board with peripheral component interconnect express (PCIe) Gen 2.0 bus. High-bandwidth PCIe interfaces and powerful FPGAs can support full bidirectional data transfer for two 10 GbE ports. Further, we created and implemented optimized techniques to achieve not only high-performance packet processing, but also high-performance 10 GbE bitstream control in software. Parallelism and optimizations allow SoNIC to process multiple 10 GbE bitstreams at line-speed.

With software access to the PHY, SoNIC provides the opportunity to improve upon and develop new network research applications which were not previously feasible. First, as a powerful network measurement tool, SoNIC can generate packets at full data rate with minimal interpacket delay. It also provides fine-grain control over the interpacket delay; it can inject packets with no variance in the interpacket delay. Second, SoNIC accurately captures incoming packets at any data rate including the maximum, while simultaneously timestamping each packet with sub-nanosecond granularity. In other

[1]We use PHY to denote the physical layer throughout the paper.

words, SoNIC can capture exactly what was sent. Further, this precise timestamping can improve the accuracy of research based on interpacket delay. For example, SoNIC can be used to profile network components. It can also create timing channels that are undetectable from software application.

The contributions of SoNIC are as follows:

- We present the design and implementation of SoNIC, a new approach for accessing the entire network stack in software in realtime.
- We designed SoNIC with commodity components such as multi-core processors and a PCIe pluggable board, and present a prototype of SoNIC.
- We demonstrate that SoNIC can enable flexible, precise, and realtime network research applications. SoNIC increases the flexibility of packet generation and the accuracy of packet capture.
- We also demonstrate that network research studies based on interpacket delay can be significantly improved with SoNIC.

2 Challenge: PHY Access in Software

Accessing the physical layer (PHY) in software provides the ability to study networks and the network stack at a heretofore inaccessible level: It can help improve the precision of network measurements and profiling/monitoring by orders of magnitude [15]. Further, it can help improve the reliability and security of networks via faithful capture and replay of network traffic. Moreover, it can enable the creation of timing channels that are undetectable from higher layers of the network stack. This section discusses the requirements and challenges of achieving realtime software access to the PHY, and motivates the design decisions we made in implementing SoNIC. We also discuss the Media Access Control (MAC) layer because of its close relationship to the PHY in generating valid Ethernet frames.

The fundamental challenge to perform the PHY functionality in software is maintaining synchronization with hardware while efficiently using system resources. Some important areas of consideration when addressing this challenge include *hardware support, realtime capability, scalability and efficiency, precision, and a usable interface*. Because so many factors go into achieving realtime software access to the PHY, we first discuss the 10 GbE standard before discussing detailed requirements.

2.1 Background

According to the IEEE 802.3 standard [7], the PHY of 10 GbE consists of three sublayers: the Physical Coding Sublayer (PCS), the Physical Medium Attachment (PMA) sublayer, and the Physical Medium Dependent (PMD) sublayer (See Figure 1). The PMD sublayer is responsible for transmitting the outgoing symbolstream over the physical medium and receiving the incoming

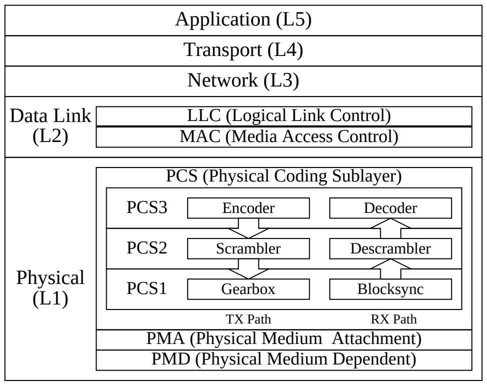

Application (L5)
Transport (L4)
Network (L3)
Data Link (L2)
LLC (Logical Link Control)
MAC (Media Access Control)
Physical (L1)
PCS (Physical Coding Sublayer)
PCS3
Encoder
Decoder
PCS2
Scrambler
Descrambler
PCS1
Gearbox
Blocksync
TX Path
RX Path
PMA (Physical Medium Attachment)
PMD (Physical Medium Dependent)

Figure 1: 10 Gigabit Ethernet Network stack.

symbolstream from the medium. The PMA sublayer is responsible for clock recovery and (de-)serializing the bitstream. The PCS performs the blocksync and gearbox (we call this PCS1), scramble/descramble (PCS2), and encode/decode (PCS3) operations on every Ethernet frame. The IEEE 802.3 Clause 49 explains the PCS sublayer in further detail, but we will summarize below.

When Ethernet frames are passed from the data link layer to the PHY, they are reformatted before being sent across the physical medium. On the transmit (TX) path, the PCS encodes every 64-bit of an Ethernet frame into a 66-bit *block* (PCS3), which consists of a two bit *synchronization header* (syncheader) and a 64-bit *payload*. As a result, a 10 GbE link actually operates at 10.3125 Gbaud ($10G \times \frac{66}{64}$). The PCS also scrambles each block (PCS2) to maintain DC balance[2] and adapts the 66-bit width of the block to the 16-bit width of the PMA interface (PCS1; the gearbox converts the bit width from 66- to 16-bit width.) before passing it down the network stack. The entire 66-bit block is transmitted as a continuous stream of *symbols* which a 10 GbE network transmits over a physical medium (PMA & PMD). On the receive (RX) path, the PCS performs block synchronization based on two-bit syncheaders (PCS1), descrambles each 66-bit block (PCS2) before decoding it (PCS3).

Above the PHY is the Media Access Control (MAC) sublayer of the data link layer. The 10 GbE MAC operates in full duplex mode; it does not handle collisions. Consequently, it only performs data encapsulation/decapsulation and media access management. Data encapsulation includes framing as well as error detection. A Cyclic Redundancy Check (CRC) is used to detect bit corruptions. Media access management inserts at least 96 bits (twelve `idle /I/` characters) between two Ethernet frames.

On the TX path, upon receiving a layer 3 packet, the MAC prepends a preamble, start frame delimiter (SFD), and an Ethernet header to the beginning of the frame. It also pads the Ethernet payload to satisfy a

[2] Direct current (DC) balance ensures a mix of 1's and 0's is sent.

minimum frame-size requirement (64 bytes), computes a CRC value, and places the value in the Frame Check Sequence (FCS) field. On the RX path, the CRC value is checked, and passes the Ethernet header and payload to higher layers while discarding the preamble and SFD.

2.2 Hardware support

The hardware must be able to transfer raw symbols from the wire to software at high speeds. This requirement can be broken down into four parts: a) Converting optical signals to digital signals (PMD), b) Clock recovery for bit detection (PMA), and c) Transferring large amounts of bits to software through a high-bandwidth interface. Additionally, d) the hardware should leave recovered bits (both control and data characters in the PHY) intact until they are transferred and consumed by the software. Commercial optical transceivers are available for a). However, hardware that simultaneously satisfies b), c) and d) is not common since it is difficult to handle 10.3125 Giga symbols in transit every second.

NetFPGA 10G [27] does not provide software access to the PHY. In particular, NetFPGA pushes not only layers 1-2 (the physical and data link layer) into hardware, but potentially layer 3 as well. Furthermore, it is not possible to easily undo this design since it uses an on-board chip to implement the PHY which prevents direct access to the PCS sublayer. As a result, we need a new hardware platform to support software access to the PHY.

2.3 Realtime Capability

Both hardware and software must be able to process 10.3125 Gigabits per second (Gbps) continuously. The IEEE 802.3 standard [7] requires the 10 GbE PHY to generate a continuous bitstream. However, synchronization between hardware and software, and between multiple pipelined cores is non-trivial. The overheads of interrupt handlers and OS schedulers can cause a discontinuous bitstream which can subsequently incur packet loss and broken links. Moreover, it is difficult to parallelize the PCS sublayer onto multiple cores. This is because the (de-)scrambler relies on state to recover bits. In particular, the (de-)scrambling of one bit relies upon the 59 bits preceding it. This fine-grained dependency makes it hard to parallelize the PCS sublayer. The key takeaway here is that everything must be efficiently pipelined and well-optimized in order to implement the PHY in software while minimizing synchronization overheads.

2.4 Scalability and Efficiency

The software must scale to process multiple 10 GbE bitstreams while efficiently utilizing resources. Intense computation is required to implement the PHY and MAC layers in software. (De-)Scrambling every bit and computing the CRC value of an Ethernet frame is especially intensive. A functional solution would require multiple duplex channels to each independently perform the CRC, encode/decode, and scramble/descramble computations at 10.3125 Gbps. The building blocks for the PCS and MAC layers will therefore consume many CPU cores. In order to achieve a scalable system that can handle multiple 10 GbE bitstreams, resources such as the PCIe, memory bus, Quick Path Interconnect (QPI), cache, CPU cores, and memory must be efficiently utilized.

2.5 Precision

The software must be able to precisely control and capture interpacket gaps. A 10 GbE network uses one bit per symbol. Since a 10 GbE link operates at 10.3125 Gbaud, each and every symbol length is 97 pico-seconds wide ($= 1/(10.3125 * 10^9)$). Knowing the number of bits can then translate into having a precise measure of time at the sub-nanosecond granularity. In particular, depending on the combination of data and control symbols in the PCS block[3], the number of bits between data frames is not necessarily a multiple of eight. Therefore, on the RX path, we can tell the exact distance between Ethernet frames in bits by counting *every bit*. On the TX path, we can control the data rate precisely by controlling the number of `idle` characters between frames: An idle character is 8 (or 7) bits and the 10 GbE standard requires at least 12 idle characters sent between Ethernet frames.

To achieve this precise level of control, the software must be able to access every bit in the raw bitstream (the symbolstream on the wire). This requirement is related to point d) from Section 2.2. The challenge is how to generate and deliver *every* bit from and to software.

2.6 User Interface

Users must be able to easily access and control the PHY. Many resources from software to hardware must be tightly coupled to allow realtime access to the PHY. Thus, an interface that allows fine-grained control over them is necessary. The interface must also implement an I/O channel through which users can retrieve data such as the count of bits for precise timing information.

3 SoNIC

The design goals of SoNIC are to provide 1) access to the PHY in software, 2) realtime capability, 3) scalability and efficiency, 4) precision, and 5) user interface. As a result, SoNIC must allow users realtime access to the PHY in software, provide an interface to applications, process incoming packets at line-speed, and be scalable. Our ultimate goal is to achieve the same flexibility and control of the entire network stack for a wired network, as a software-defined radio [33] did for a wireless network, while maintaining the same level of precision as BiFocals [15]. Access to the PHY can then enhance the accuracy of network research based on interpacket delay.

[3]Figure 49-7 [7]

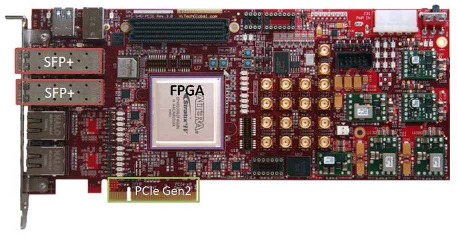


Figure 2: An FPGA development board [6]

In this section, we discuss the design of SoNIC and how it addresses the challenges presented in Section 2.

3.1 Access to the PHY in software

An application must be able to access the PHY in software using SoNIC. Thus, our solution must implement the bit generation and manipulation functionality of the PHY in software. The transmission and reception of bits can be handled by hardware. We carefully examined the PHY to determine an optimal partitioning of functionality between hardware and software.

As discussed in Section 2.1, the PMD and PMA sublayers of the PHY do not modify any bits or change the clock rate. They simply forward the symbolstream/bitstream to other layers. Similarly, PCS1 only converts the bit width (gearbox), or identifies the beginning of a new 64/66 bit block (blocksync). Therefore, the PMD, PMA, and PCS1 are all implemented in hardware as a forwarding module between the physical medium and SoNIC's software component (See Figure 1). Conversely, PCS2 (scramble/descramble) and PCS3 (encode/decode) actually manipulate bits in the bitstream and so they are implemented in SoNIC's software component. SoNIC provides full access to the PHY in software; as a result, all of the functionality in the PHY that manipulate bits (PCS2 and PCS3) are implemented in software.

For this partitioning between hardware and software, we chose an Altera Stratix IV FPGA [4] development board from HiTechGlobal [6] as our hardware platform. The board includes a PCIe Gen 2 interface (=32 Gbps) to the host PC, and is equipped with two SFP+ (Small Form-factor Pluggable) ports (Figure 2). The FPGA is equipped with 11.3 Gbps transceivers which can perform the 10 GbE PMA at line-speed. Once symbols are delivered to a transceiver on the FPGA they are converted to bits (PMA), and then transmitted to the host via PCIe by direct memory access (DMA). This board satisfies all the requirements discussed in the previous Section 2.2.

3.2 Realtime Capability

To achieve realtime, it is important to reduce any synchronization overheads between hardware and software, and between multiple pipelined cores. In SoNIC, the hardware does not generate interrupts when receiving or transmitting. Instead, the software decides when to initiate a DMA transaction by *polling* a value from a shared data memory structure where only the hardware writes. This approach is called *pointer polling* and is better than interrupts because there is always data to transfer due to the nature of continuous bitstreams in 10 GbE.

In order to synchronize multiple pipelined cores, a *chasing-pointer FIFO* from Sora [33] is used which supports low-latency pipelining. The FIFO removes the need for a shared synchronization variable and instead uses a flag to indicate whether a FIFO entry is available to reduce the synchronization overheads. In our implementation, we improved the FIFO by avoiding memory operations as well. Memory allocation and page faults are expensive and must be avoided to meet the realtime capability. Therefore, each FIFO entry in SoNIC is pre-allocated during initialization. In addition, the number of entries in a FIFO is kept small so that the amount of memory required for a port can fit into the shared L3 cache.

We use the Intel Westmere processor to achieve high performance. Intel Westmere is a Non-Uniform Memory Access (NUMA) architecture that is efficient for implementing packet processing applications [14, 18, 28, 30]. It is further enhanced by a new instruction `PCLMULQDQ` which was recently introduced. This instruction performs carry-less multiplication and we use it to implement a fast CRC algorithm [16] that the MAC requires. Using `PCLMULQDQ` instruction makes it possible to implement a CRC engine that can process 10 GbE bits at line-speed on a single core.

3.3 Scalability and Efficiency

The FPGA board we use is equipped with two physical 10 GbE ports and a PCIe interface that can support up to 32 Gbps. Our design goal is to support two physical ports per board. Consequently, the number of CPU cores and the amount of memory required for one port must be bounded. Further, considering the intense computation required for the PCS and MAC, and that recent processors come with four to six or even eight cores per socket, our goal is to limit the number of CPU cores required per port to the number of cores available in a socket. As a result, for one port we implement four dedicated kernel threads each running on different CPU cores. We use a PCS thread and a MAC thread on both the transmit and receive paths. We call our threads: TX PCS, RX PCS, TX MAC and RX MAC. Interrupt requests (IRQ) are re-

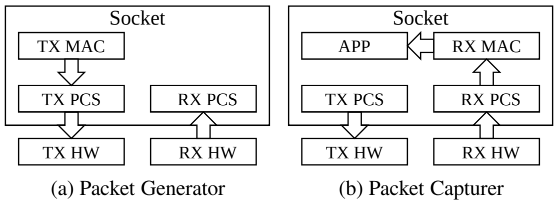


Figure 3: Example usages of SoNIC

routed to unused cores so that SoNIC threads do not give up the CPU and can meet the realtime requirements.

Additionally, we use memory very efficiently: DMA buffers are preallocated and reused and data structures are kept small to fit in the shared L3 cache. Further, by utilizing memory efficiently, dedicating threads to cores, and using multi-processor QPI support, we can linearly increase the number of ports with the number of processors. QPI provides enough bandwidth to transfer data between sockets at a very fast data rate ($>$ 100 Gbps).

A significant design issue still abounds: communication and CPU core utilization. The way we pipeline CPUs, i.e. sharing FIFOs depends on the application. In particular, we pipeline CPUs differently depending on the application to reduce the number of active CPUs; unnecessary CPUs are returned to OS. Further, we can enhance communication with a general rule of thumb: take advantage of the NUMA architecture and L3 cache and place closely related threads on the same CPU socket.

Figure 3 illustrates examples of how to share FIFOs among CPUs. An arrow is a shared FIFO. For example, a packet generator only requires TX elements (Figure 3a); RX PCS simply receives and discards bitstreams, which is required to keep a link active. On the contrary, a packet capturer requires RX elements (Figure 3b) to receive and capture packets. TX PCS is required to establish and maintain a link to the other end by sending `/I/`s. To create a network profiling application, both the packet generator and packet capturer can run on different sockets simultaneously.

3.4 Precision

As discussed in Section 3.1, the PCS2 and PCS3 are implemented in software. Consequently, the software receives the entire raw bitstream from the hardware. While performing PCS2 and PCS3 functionalities, a PCS thread records the number of bits in between and within each Ethernet frame. This information can later be retrieved by a user application. Moreover, SoNIC allows users to precisely control the number of bits in between frames when transmitting packets, and can even change the value of any bits. For example, we use this capability to give users fine-grain control over packet generators and can even create virtually undetectable covert channels.

3.5 User Interface

SoNIC exposes fine-grained control over the path that a bitstream travels in software. SoNIC uses the `ioctl` system call for control, and the character device interface to transfer information when a user application needs to retrieve data. Moreover, users can assign which CPU cores or socket each thread runs on to optimize the path.

To allow further flexibility, SoNIC allows additional application-specific threads, called APP threads, to be pipelined with other threads. A character device is used to communicate with these APP threads from userspace. For instance, users can implement a logging thread pipelined with receive path threads (RX PCS and/or RX MAC). Then the APP thread can deliver packet information along with precise timing information to userspace via a character device interface. There are two constraints that an APP thread must always meet: Performance and pipelining. First, whatever functionality is implemented in an APP thread, it must be able to perform it faster than 10.3125 Gbps for any given packet stream in order to meet the realtime capability. Second, an APP thread must be properly pipelined with other threads, i.e. input/output FIFO must be properly set. Currently, SoNIC supports one APP thread per port.

```
1: #include "sonic.h"
2:
3: struct sonic_pkt_gen_info info = {
4:     .pkt_num   = 1000000000UL,
5:     .pkt_len   = 1518,
6:     .mac_src   = "00:11:22:33:44:55",
7:     .mac_dst   = "aa:bb:cc:dd:ee:ff",
8:     .ip_src    = "192.168.0.1",
9:     .ip_dst    = "192.168.0.2",
10:    .port_src  = 5000,
11:    .port_dst  = 5000,
12:    .idle   = 12, };
13:
14: fd1 = open(SONIC_CONTROL_PATH, O_RDWR);
15: fd2 = open(SONIC_PORT1_PATH, O_RDONLY);
16:
17: ioctl(fd1, SONIC_IOC_RESET)
18: ioctl(fd1, SONIC_IOC_SET_MODE, SONIC_PKT_GEN_CAP)
19: ioctl(fd1, SONIC_IOC_PORT0_INFO_SET, &info)
20: ioctl(fd1, SONIC_IOC_RUN, 10)
21:
22: while ((ret = read(fd2, buf, 65536)) > 0) {
23:  // process data }
24:
25: close(fd1);
26: close(fd2);
```

Figure 4: E.g. SoNIC Packet Generator and Capturer

Figure 4 illustrates the source code of an example use of SoNIC as a packet generator and capturer. After `SONIC_IOC_SET_MODE` is called (line 18), threads are pipelined as illustrated in Figure 3a and 3b. After `SONIC_IOC_RUN` command (line 20), port 0 starts generating packets given the information from `info` (line 3-12) for 10 seconds (line 20) while port 1 starts capturing packets with very precise timing information. Captured information is retrieved with `read` system calls (line 22-23) via a character device. As a packet generator, users can set the desired number of `/I/`s between packets (line 12). For example, twelve `/I/` characters will achieve the maximum data rate. Increasing the number of `/I/` characters will decrease the data rate.

3.6 Discussion

We have implemented SoNIC to achieve the design goals described above, namely, software access to the PHY, realtime capability, scalability, high precision, and an interactive user interface. Figure 5 shows the major components of our implementation. From top to bottom, user

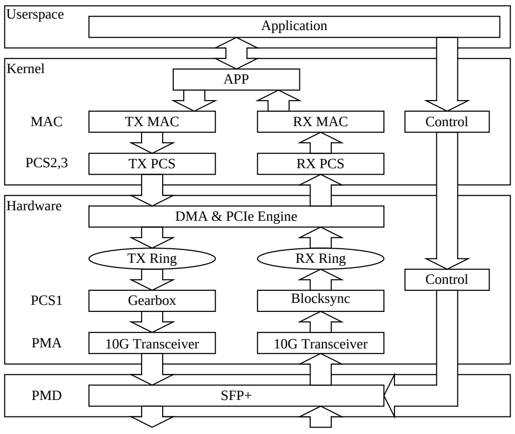


Figure 5: SoNIC architecture

applications, software as a loadable Linux kernel module, hardware as a firmware in FPGA, and a SFP+ optical transceiver. Although Figure 5 only illustrates one physical port, there are two physical ports available in SoNIC. SoNIC software consists of about 6k lines of kernel module code, and SoNIC hardware consists of 6k lines of Verilog code excluding auto-generated source code by Altera Quartus [3] with which we developed SoNIC's hardware modules.

The idea of accessing the PHY in software can be applied to other physical layers with different speeds. The 1 GbE and 40 GbE PHYs are similar to the 10 GbE PHY in that they run in full duplex mode, and maintain continuous bitstreams. Especially, the 40GbE PCS employes four PCS lanes that implements 64B/66B encoding as in the 10GbE PHY. Therefore, it is possible to access the PHYs of them with appropriate clock cycles and hardware supports. However, it might not be possible to implement four times faster scrambler with current CPUs.

In the following sections, we will highlight how SoNIC's implementation is optimized to achieve high performance, flexibility, and precision.

4 Optimizations

Performance is paramount for SoNIC to achieve its goals and allow software access to the entire network stack. In this section we discuss the software (Section 4.1) and hardware (Section 4.2) optimizations that we employ to enable SoNIC. Further, we evaluate each optimization (Sections 4.1 and 4.2) and demonstrate that they help to enable SoNIC and network research applications (Section 5) with high performance.

4.1 Software Optimizations

MAC Thread Optimizations As stated in Section 3.2, we use PCLMULQDQ instruction which performs carry-less multiplication of two 64-bit quadwords [17] to implement the fast CRC algorithm [16]. The algorithm *folds* a large chunk of data into a smaller chunk using the PCLMULQDQ instruction to efficiently reduce the size of data. We adapted this algorithm and implemented it using inline assembly with optimizations for small packets.

PCS Thread Optimizations Considering there are 156 million 66-bit blocks a second, the PCS must process each block in less than 6.4 nanoseconds. Our optimized (de-)scrambler can process each block in 3.06 nanoseconds which even gives enough time to implement decode/encode and DMA transactions within a single thread.

In particular, the PCS thread needs to implement the (de-)scrambler function, $G(x) = 1 + x^{39} + x^{58}$, to ensure that a mix of 1's and 0's are always sent (DC balance). The (de-)scrambler function can be implemented with Algorithm 1, which is very computationally expensive [15] taking 320 shift and 128 xor operations (5 shift operations and 2 xors per iteration times 64 iterations). In fact, our original implementation of Algorithm 1 performed at 436 Mbps, which was not sufficient and became the bottleneck for the PCS thread. We optimized and reduced the scrambler algorithm to a *total* of 4 shift and 4 xor operations (Algorithm 2) by carefully examining how hardware implements the scrambler function [34]. Both Algorithm 1 and 2 are equivalent, but Algorithm 2 runs 50 times faster (around 21 Gbps).

Algorithm 1 Scrambler
```
$s \leftarrow$ state
$d \leftarrow$ data
for $i = 0 \rightarrow 63$ do
    $in \leftarrow (d >> i)$ & 1
    $out \leftarrow (in \oplus (s >> 38) \oplus (s >> 57))$ & 1
    $s \leftarrow (s << 1) \mid out$
    $r \leftarrow r \mid (out << i)$
    state $\leftarrow s$
end for
```

Algorithm 2 Parallel Scrambler
```
$s \leftarrow$ state
$d \leftarrow$ data
$r \leftarrow (s >> 6) \oplus (s >> 25) \oplus d$
$r \leftarrow r \oplus (r << 39) \oplus (r << 58)$
state $\leftarrow r$
```

Memory Optimizations We use *packing* to further improve performance. Instead of maintaining an array of data structures that each contains metadata and a pointer to the packet payload, we pack as much data as possible into a preallocated memory space: Each packet structure contains metadata, packet payload, and an offset to the next packet structure in the buffer. This packing helps to reduce the number of page faults, and allows SoNIC to process small packets faster. Further, to reap the benefits of the PCLMULQDQ instruction, the first byte of each packet is always 16-byte aligned.

Evaluation We evaluated the performance of the TX MAC thread when computing CRC values to assess the performance of the fast CRC algorithm and packing packets we implemented relative to batching an array of

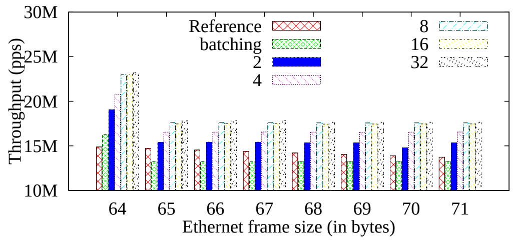


Figure 6: Throughput of packing

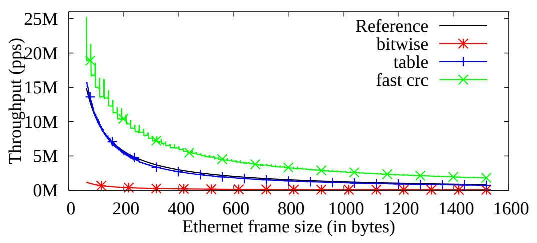


Figure 7: Throughput of different CRC algorithms

packets. For comparison, we computed the theoretical maximum throughput (Reference throughput) in packets per second (pps) for any given packet length (i.e. the pps necessary to achieve the maximum throughput of 10 Gbps less any protocol overhead).

If only one packet is packed in the buffer, packing will perform the same as batching since the two are essentially the same in this case. We doubled the factor of packing from 1 to 32 and assessed the performance of packing each time, i.e. we doubled the number of packets written to a single buffer. Figure 6 shows that packing by a factor of 2 or more always outperforms the Reference throughput and is able to achieve the max throughput for small packets while batching does not.

Next, we compared our fast CRC algorithm against two CRC algorithms that the Linux Kernel provides. One of the Linux CRC algorithms is a naive bit computation and the other is a table lookup algorithm. Figure 7 illustrates the results of our comparisons. The x-axis is the length of packets tested while the y-axis is the throughput. The Reference line represents the maximum possible throughput given the 10 GbE standard. Packet lengths range the spectrum of sizes allowed by 10 GbE standard from 64 bytes to 1518 bytes. For this evaluations, we allocated 16 pages *packed* with packets of the same length and computed CRC values with different algorithms for 1 second. As we can see from Figure 7, the throughput of the table lookup closely follows the Reference line; however, for several packet lengths, it underperforms the Reference line and is unable to achieve the maximum throughput. The fast CRC algorithm, on the other hand, outperforms the Reference line and target throughput for all packet sizes.

Lastly, we evaluated the performance of pipelining and using multiple threads on the TX and RX paths. We tested a full path of SoNIC to assess the performance as packets travel from the TX MAC to the TX PCS for transmission and up the reverse path for receiving from the RX PCS to the RX MAC and to the APP (as a logging thread). We do not show the graph due to space constraints, but all threads perform better than the Reference target throughput. The overhead of FIFO is negligible when we compare the throughputs of individual threads to the throughput when all threads are pipelined together. Moreover, when using two ports simultaneously (two full instances of receive and transmit SoNIC paths), the throughput for both ports achieve the Reference target maximum throughput.

4.2 Hardware Optimizations

DMA Controller Optimizations Given our desire to transfer large amounts of data (more than 20 Gbps) over the PCIe, we implemented a high performance DMA controller. There are two key factors that influenced our design of the DMA controller. First, because the incoming bitstream is a continuous 10.3125 Gbps, there must be enough buffering inside the FPGA to compensate for a transfer latency. Our implementation allocates four rings in the FPGA for two ports (Figure 5 shows two of the rings for one port). The maximum size of each ring is 256 KB, with the size being limited by the amount of SRAM available.

The second key factor we needed to consider was the efficient utilization of bus bandwidth. The DMA controller operates at a data width of 128 bits. If we send a 66-bit data block over the 128-bit bus every clock cycle, we will waste 49% of the bandwidth, which was not acceptable. To achieve more efficient use of the bus, we create a `sonic_dma_page` data structure and separated the syncheader from the packet payload before storing a 66-bit block in the data structure. Sixteen two-bit syncheaders are concatenated together to create a 32-bit integer and stored in the `syncheaders` field of the data structure. The 64-bit packet payloads associated with these syncheaders are stored in the `payloads` field of the data structure. For example, the ith 66-bit PCS block from a DMA page consists of the two-bit sync header from `syncheaders[`$i/16$`]` and the 64-bit payload from `payloads[`i`]`. With this data structure there is a 32-bit overhead for every page, however it does not impact the overall performance.

PCI Express Engine Optimizations When SoNIC was first designed, it only supported a single port. As we scaled SoNIC to support multiple ports simultaneously, the need for multiplexing traffic among ports over the single PCIe link became a significant issue. To solve this issue, we employ a two-level arbitration scheme to provide fair arbitration among ports. A lower level arbiter is a fixed-priority arbiter that works within a sin-

Configuration	Same Socket?	# pages	Throughput (RX)		# pages	Throughput (TX)		Realtime?
			Port 0	Port 1		Port 0	Port 1	
Single RX		16	25.7851					
Dual RX	Yes	16	13.9339	13.899				
	No	8	14.2215	13.134				
Single TX					16	23.7437		
Dual TX	Yes				16	14.0082	14.048	
	No				16	13.8211	13.8389	
Single RX/TX		16	21.0448		16	22.8166		
Dual RX/TX	Yes	4	10.7486	10.8011	8	10.6344	10.7171	No
		4	11.2392	11.2381	16	12.384	12.408	Yes
		8	13.9144	13.9483	8	9.1895	9.1439	Yes
		8	14.1109	14.1107	16	10.6715	10.6731	Yes
	No	4	10.5976	10.183	8	10.3703	10.1866	No
		4	10.9155	10.231	16	12.1131	11.7583	Yes
		8	13.4345	13.1123	8	8.3939	8.8432	Yes
		8	13.4781	13.3387	16	9.6137	10.952	Yes

Table 1: DMA throughput. The numbers are average over eight runs. The delta in measurements was within 1% or less.

gle port and arbitrates between four basic Transaction Level Packet (TLP) types: Memory, I/O, configuration, and message. The TLPs are assigned with fixed priority in favor of the write transaction towards the host. The second level arbiter implements a virtual channel, where the Traffic Class (TC) field of TLP's are used as demultiplexing keys. We implemented our own virtual channel mechanism in SoNIC instead of using the one available in the PCIe stack since virtual channel support is an optional feature for vendors to comply with. In fact, most chipsets on the market do not support the virtual channel mechanism. By implementing the virtual channel support in SoNIC, we achieve better portability since we do not rely on chip vendors that enable PCI arbitration.

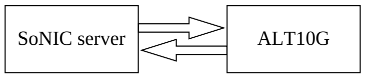
Evaluation We examined the maximum throughput for DMA between SoNIC hardware and SoNIC software to evaluate our hardware optimizations. It is important that the bidirectional data rate of each port of SoNIC is greater than 10.3125 Gbps. For this evaluation, we created a DMA descriptor table with one entry, and changed the size of memory for each DMA transaction from one page (4K) to sixteen pages (64KB), doubling the number of pages each time. We evaluated the throughput of a single RX or TX transaction, dual RX or TX transactions, and full bidirectional RX and TX transactions with both one and two ports (see the rows of Table 1). We also measured the throughput when traffic was sent to one or two CPU sockets.

Table 1 shows the DMA throughputs of the transactions described above. We first measured the DMA without using pointer polling (see Section 3.2) to obtain the maximum throughputs of the DMA module. For single RX and TX transactions, the maximum throughput is close to 25 Gbps. This is less than the theoretical maximum throughput of 29.6 Gbps for the x8 PCIe interface, but closely matches the reported maximum throughput of 27.5 Gbps [2] from Altera design. Dual RX or TX transactions also resulted in throughputs similar to the reference throughputs of Altera design.

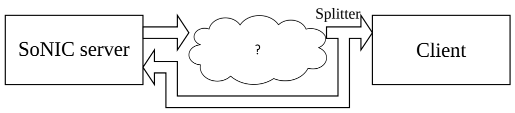
Next, we measured the full bidirectional DMA transactions for both ports varying the number of pages again. As shown in the bottom half of Table 1, we have multiple configurations that support throughputs greater than 10.3125 Gbps for full bidirections. However, there are a few configurations in which the TX throughput is less than 10.3125 Gbps. That is because the TX direction requires a small fraction of RX bandwidth to fetch the DMA descriptor. If RX runs at maximum throughput, there is little room for the TX descriptor request to get through. However, as the last column on the right indicates these configurations are still able to support the realtime capability, i.e. consistently running at 10.3125 Gbps, when *pointer polling* is enabled. This is because the RX direction only needs to run at 10.3125 Gbps, less than the theoretical maximum throughput (14.8 Gbps), and thus gives more room to TX. On the other hand, two configurations where both RX and TX run faster than 10.3125 Gbps for full bidirection are not able to support the realtime capability. For the rest of the paper, we use 8 pages for RX DMA and 16 pages for TX DMA.

5 Network Research Applications

How can SoNIC enable flexible, precise and novel network research applications? Specifically, what unique value does software access to the PHY buy? As discussed in Section 2.5, SoNIC can literally count the number of bits between and within packets, which can be used for timestamping at the sub-nanosecond granularity (again each bit is 97 ps wide, or about $\sim$0.1 ns). At the same time, access to the PHY allows users control over the number of `idles` (`/I/`s) between packets when generating packets. This fine-grain control over the `/I/`s means we can precisely control the data rate and the distribution of interpacket gaps. For example, the data rate of a 64B packet stream with uniform 168 `/I/`s is 3 Gbps. When this precise packet generation is combined with exact packet capture, also enabled by SoNIC, we can improve the accuracy of any research based on interpacket delays [11, 19, 22, 23, 24, 25, 26, 32, 35, 37].



(a) Evaluation setup for packet generator and capturer



(b) Evaluation setup for other applications

Figure 8: Experiment setups for SoNIC

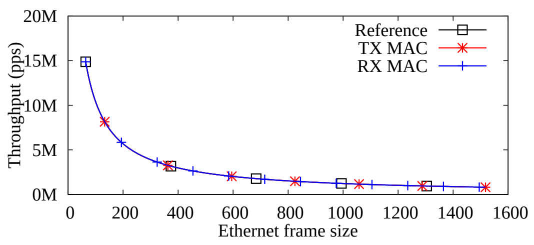


Figure 9: Throughput of packet generator and capturer

In this section, we demonstrate that SoNIC can precisely and flexibly characterize and profile commodity network components like routers, switches, and NICs. Section 5.4 discusses the profiling capability enabled by SoNIC. Further, in Section 5.5, we demonstrate that SoNIC can be used to create a covert timing channel that is not detectable by applications that do not have access to the PHY and data link layers and that do not have accurate timestamping capabilities. First, however, we demonstrate SoNIC's accurate packet generation capability in Section 5.2 and packet capture capability in Section 5.3, which are unique contributions and can enable unique network research in and of themselves given both the flexibility, control, and precision.

Interpacket delay (IPD) and *interpacket gap* (IPG) are defined as follows: IPD is the time difference between the first bit of successive packets, while IPG is the time difference between the last bit of the first packet and the first bit of the next packet.

5.1 Experiment Setup

We deployed SoNIC on a Dell Precision T7500 workstation. This workstation is a dual socket, 2.93 GHz six core Xeon X5670 (Westmere) with 12 MB of shared L3 cache and 12 GB of RAM, 6 GB connected to each of the two CPU sockets. The machine has two PCIe Gen 2.0 x8 slots, where SoNIC hardware is plugged in, and is equipped with an Intel 5520 chipset connected to each CPU socket by a 6.4 GT/s QuickPath Interconnect (QPI). Two Myricom 10G-SFP-LR transceivers are plugged into SoNIC hardware. We call this machine the SoNIC server. For our evaluations we also deployed an Altera 10G Ethernet design [1] (we call this ALT10G) on an FPGA development board. This FPGA is the same type as the one SoNIC uses and is deployed on a server identical to the SoNIC server. We also used a server identical to the SoNIC server with a Myricom 10G-PCIE2-8B2-2S dual 10G port NIC (we call this Client).

To evaluate SoNIC as a packet generator and capturer, we connected the SoNIC board and the ALT10G board directly via optic fibers (Figure 8a). ALT10G allows us to generate random packets of any length and with the minimum IPG to SoNIC. It also provides us with detailed statistics such as the number of valid/invalid Ethernet frames, and frames with CRC errors. We used this feature to stress test SoNIC for the packet generator and capturer. We compared these numbers from ALT10G with statistics from SoNIC to verify the correctness of SoNIC.

To evaluate other applications, we used port 0 of SoNIC server to generate packets to the Client server via an arbitrary network, and split the signal with a fiber optic splitter so that the same stream can be directed to both the Client and port 1 of the SoNIC server performing the packet capture (Figure 8b). We used various network topologies composed of Cisco 4948, and IBM BNT G8264 switches for the network between the SoNIC server and the Client.

5.2 Packet Generator

Packet generation is important for network research. It can stress test end-hosts, switches/routers, or a network itself. Moreover, packet generation can be used for replaying a trace, studying distributed denial of service (DDoS) attacks, or probing firewalls.

In order to claim that a packet generator is accurate, packets need to be crafted with fine-grained precision (minimum deviations in IPD) at the maximum data rate. However, this fine-grained control is not usually exposed to users. Further, commodity servers equipped with a commodity NIC often does not handle small packets efficiently and require batching [14, 18, 28, 30]. Thus, the sending capability of servers/software-routers are determined by the network interface devices. Myricom Sniffer 10G [8] provides line-rate packet injection capability, but does not provide fine-grained control of IPGs. Hardware based packet generators such as ALT10G can precisely control IPGs, but do not provide any interface for users to flexibly control them.

We evaluated SoNIC as a packet generator (Figure 3a). Figure 10 compares the performance of SoNIC to that of Sniffer 10G. Note, we do not include ALT10G in this evaluation since we could not control the IPG to generate packets at 9 Gbps. We used two servers with Sniffer 10G enabled devices to generate 1518B packets at 9 Gbps between them. We split the stream so that SoNIC can capture the packet stream in the middle (we describe this capture capability in the following section). As the graph shows, Sniffer 10G allows users to generate packets at desired data rate, however, it does not give the con-

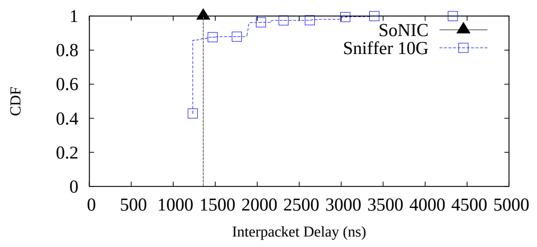


Figure 10: Comparison of packet generation at 9 Gbps

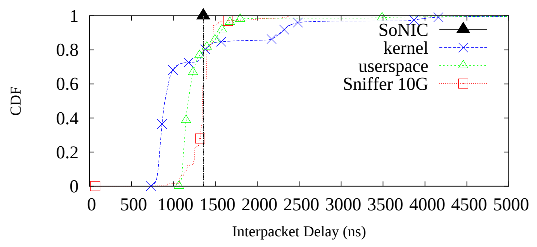


Figure 11: Comparison of timestamping

trol over the IPD; that is, 85.65% packets were sent in a burst (instantaneous 9.8 Gbps and minimum IPG (14 /I/s)). SoNIC, on the other hand, can generate packets with uniform distribution. In particular, SoNIC generated packets with no variance for the IPD (i.e. a single point on the CDF, represented as a triangle). Moreover, the maximum throughput perfectly matches the Reference throughput (Figure 9) while the TX PCS consistently runs at 10.3125 Gbps (which is not shown). In addition, we observed no packet loss, bit errors, or CRC errors during our experiments.

SoNIC packet generator can easily achieve the maximum data rate, and allows users to precisely control the number of /I/s to set the data rate of a packet stream. Moreover, with SoNIC, it is possible to inject less /I/s than the standard. For example, we can achieve 9 Gbps with 64B packets by inserting only eight /I/s between packets. This capability is not possible with any other (software) platform. In addition, if the APP thread is carefully designed, users can flexibly inject a random number of /I/s between packets, or the number of /I/s from captured data. SoNIC packet generator is thus by far the most flexible and highest performing.

5.3 Packet Capturer

A packet capturer (a.k.a. packet sniffer, or packet analyzer) plays an important role in network research; it is the opposite side of the same coin as a packet generator. It can record and log traffic over a network which can later be analyzed to improve the performance and security of networks. In addition, capturing packets with precise timestamping is important for High Frequency Trading [21, 31] or latency sensitive applications.

Similar to the sending capability, the receiving capability of servers and software routers is inherently limited by the network adapters they use; it has been shown that some NICs are not able to receive packets at line speed for certain packet sizes [30]. Furthermore, if batching is used, timestamping is significantly perturbed if done in kernel or userspace [15]. High-performance devices such as Myricom Sniffer10G [8, 20] provide the ability of sustained capture of 10 GbE by bypassing kernel network stack. It also provides timestamping at 500 ns resolution for captured packets. SoNIC, on the other hand, can receive packets of any length at line-speed with precise timestamping. For instance, we will show in Section 5.5 that we can create a covert timing channel that is undetectable to a Sniffer 10G enabled system or any other software-enabled systems[14, 18, 28, 30].

Putting it all together, when we use SoNIC as a packet capturer (Figure 3b), we are able to receive at the full Reference data rate (Figure 9). For the APP thread, we implemented a simple logging application which captures the first 48 bytes of each packet along with the number of /I/s and bits between packets. Because of the relatively slow speed of disk writes, we store the captured information in memory. This requires about 900MB to capture a stream of 64 byte packets for 1 second, and 50 MB for 1518 byte packets. We use ALT10G to generate packets for 1 second and compare the number of packets received by SoNIC to the number of packets generated.

SoNIC has perfect packet capture capabilities with flexible control in software. In particular, Figure 11 shows that given a 9 Gbps generated traffic with uniform IPD (average IPD=1357.224ns, stdev=0), SoNIC captures what was sent; this is shown as a single triangle at (1357.224, 1). All the other packet capture methods within userspace, kernel or a mixture of hardware timestamping in userspace (Sniffer 10G) failed to accurately capture what was sent. We receive similar results at lower bandwidths as well.

5.4 Profiler

Interpacket delays are a common metric for network research. It can be used to estimate available bandwidth [23, 24, 32], increase TCP throughput [37], characterize network traffic [19, 22, 35], and detect and prevent covert timing channels [11, 25, 26]. There are a lot of metrics based on IPD for these areas. We argue that SoNIC can increase the accuracy of those applications because of its precise control and capture of IPDs. In particular, when the SoNIC packet generator and capturer are combined, i.e. one port transmits packets while the other port captures, SoNIC can be a flexible platform for various studies. As an example, we demonstrate how SoNIC can be used to profile network switches.

Switches can be generally divided into two categories: store-and-forward and cut-through switches. Store-and-forward switches decode incoming packets, buffers them before making a routing decision. On the other hand, cut-

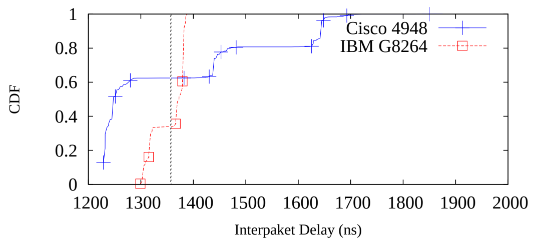


Figure 12: IPDs of Cisco 4948 and IBM G8264. 1518B packets at 9 Gbps

through switches route incoming packets before entire packets are decoded to reduce the routing latency. We generated 1518B packets with uniform 1357.19 ns IPD (=9 Gbps) to a Cisco 4948 (store-and-forward) switch and a IBM BNT G8264 (cut-through) switch. These switches show different characteristics as shown in Figure 12. The x-axis is the interpacket delay; the y-axis is the cumulative distribution function. The long dashed vertical line on the left is the original IPD injected to the packet stream.

There are several takeaways from this experiment. First, the IPD for generated packets had no variance; none. The generated IPD produced by SoNIC was *always* the same. Second, the cut-through switch introduces IPD variance (stdev=31.6413), but less than the IPD on the store-and-forward switch (stdev=161.669). Finally, the average IPD was the same for both switches since the data rate was the same: 1356.82 (cut-through) and 1356.83 (store-and-forward). This style of experiment can be used to profile and fingerprint network components as different models show different packet distributions.

5.5 Covert Channels

Covert channels in a network is a side channel that can convey a hidden message embedded to legitimate packets. There are two types of covert channels: Storage and timing channels. Storage channels use a specific location of a packet to deliver a hidden message. Timing channels modulate resources over time to deliver a message [38]. Software access to the PHY opens the possibility for both storage and timing channels. The ability to detect covert channels is important because a rogue router or an end-host can create covert channels to deliver sensitive information without alerting network administrators. We will discuss how to create covert channels with SoNIC, and thus argue that SoNIC can be used to detect any potentially undetectable covert channels in local area network or data center networks.

When SoNIC enabled devices are directly connected between two end-hosts, a secret message can be communicated without alerting the end-host. In particular, there are unused bits in the 10 GbE standard where an adversary can inject bits to create covert messages. Unfortunately, such a covert storage channel can only work for one hop. On the other hand, precisely controlling IPG can create a timing channel that can travel multiple hops in the network, and that cannot be easily detected with inaccurate timestamping. Such a covert timing channel is based on the two observations: First, timing channel detection is usually performed in the application layer, and depends on the inherently inaccurate timestamping from kernel or userspace. Secondly, switches perturb the IPD, although the difference is still bounded, i.e. an IPD does not increase to an arbitrarily long one. Therefore, if we can modulate IPGs in a way that allows switches to preserve the gaps while making them indistinguishable to kernel/userspace timestamping, it is possible to create a virtually undetectable timing channel from applications operating at layers higher than layer 2.

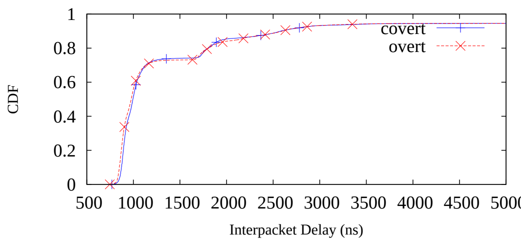


Figure 13: Packet distribution of covert channel and overt channel where $\Delta = 3562$ and $\varepsilon = 128$

We experimented with the creation of a simple timing channel. Let Δ be an uniform IPG for a packet stream. Then a small time window can be used to signal 1's and 0's. For example, IPG with $\Delta - \varepsilon$ `/I/`s represents 0 (if $\Delta - \varepsilon < 12$, we set it to 12, the minimum IPG) and $\Delta + \varepsilon$ `/I/`s represents 1. If the number of 1' and 0's meets the DC balance, the overall data rate will be similar to a packet stream with uniform IPGs of Δ `/I/`s. To demonstrate the feasibility of this approach, we created a network topology such that a packet travels from SoNIC through a Cisco 4948, IBM G8264, a different Cisco 4948, and then to the SoNIC server and the Client server with a fiber splitter (Figure 8b). SoNIC generates 1518B packets with $\Delta = 170,\ 1018,\ 3562,\ 13738$ (= 9, 6, 3, 1 Gbps respectively), with $\varepsilon = 16, 32, ..., 2048$. Then, we measured the bit error ratio (BER) with captured packets. Table 2 summarizes the result. We only illustrated the smallest ε from each Δ that achieves BER less than 1%. The takeaway is that by modulating IPGs at 100 ns scale, we can create a timing channel.

Data rate (Gbps)	Δ	δ (# `/I/`s)	δ (in ns)	BER
9	170	2048	1638.9	0.0359
6	1018	1024	819.4	0.0001
3	3562	128	102.4	0.0037
1	13738	128	102.4	0.0035

Table 2: Bit error ratio of timing channels

Figure 13 illustrates the IPDs with kernel timestamping from overt channel and covert channel when $\Delta =$

3562 and $\varepsilon = 128$. The two lines closely overlap, indicating that it is not easy to detect. There are other metrics to evaluate the undetectability of a timing channel [11, 25, 26], however they are out of the scope of this paper, and we do not discuss them.

6 Related Works

6.1 Reconfigurable Network Hardware

Reconfigurable network hardware allows for the experimentation of novel network system architectures. Previous studies on reconfigurable NICs [36] showed that it is useful for exploring new I/O virtualization technique in VMMs. NetFPGA [27] allows users to experiment with FPGA-based router and switches for research in new network protocols and intrusion detection [10, 12, 29, 39]. The recent NetFPGA 10G platform is similar to the SoNIC platform. While NetFPGA 10G allows user to access the layer 2 and above, SoNIC allows user to access the PHY. This means that user can access the entire network stack in software using SoNIC.

6.2 Timestamp

The importance of timestamping has long been established in the network measurement community. Prior work either does not provide precise enough timestamping, or requires special devices. Packet stamping in user-space or kernel suffers from the imprecision introduced by the OS layer [13]. Timestamping in hardware either requires offloading the network stack to a custom processor [37], or relies on an external clock source [5], which makes the device hard to program and inconvenient to use in a data center environment. Data acquisition and generation (DAG) cards [5] additionally offer globally synchronized clocks among multiple devices, whereas SoNIC only supports delta timestamping.

Although BiFocals [15] is able to provide an exact timestamping, the current state-of-art has limitations that prevented it from being a portable and realtime tool. BiFocals can store and analyze only a few milliseconds worth of a bitstream at a time due to the small memory of the oscilloscope. Furthermore, it requires thousands of CPU hours for converting raw optic waveforms to packets. Lastly, the physics equipment used by BiFocals are expensive and not easily portable. Its limitations motivated us to design SoNIC to achieve the realtime exact precision timestamping.

6.3 Software Defined Radio

The Software Defined Radio (SDR) allows easy, rapid prototyping of wireless network in software. Open-access platforms such as the Rice University's WARP [9] allow researchers to program both the physical and network layer on a single platform. Sora [33] presented the first SDFR platform that fully implements IEEE 802.11b/g on a commodity PC. AirFPGA [39] implemented a SDR platform on NetFPGA, focusing on building a chain of signal processing engines using commodity machines. SoNIC is similar to Sora in that it allows users to access and modify the PHY and MAC layers. The difference is that SoNIC must process multiple 10 Gbps channels which is much more computationally intensive than the data rate of wireless channels. Moreover, it is harder to synchronize hardware and software because a 10GbE link runs in a full duplex mode, unlike a wireless newtork.

6.4 Software Router

Although SoNIC is orthogonal to software routers, it is worth mentioning software routers because they share common techniques. SoNIC preallocates buffers to reduce memory overhead [18, 30], polls huge chunks of data from hardware to minimize interrupt overhead [14, 18], packs packets in a similar fashion to batching to improve performance [14, 18, 28, 30]. Software routers normally focus on scalability and hence exploit multicore processors and multi-queue supports from NICs to distribute packets to different cores to process. On the other hand, SoNIC pipelines multiple CPUs to handle continuous bitstreams.

7 Conclusion

In this paper, we presented SoNIC which allows users to access the physical layer in realtime from software. SoNIC can generate, receive, manipulate and forward 10 GbE bitstreams at line-rate from software. Further, SoNIC gives systems programmers unprecedented precision for network measurements and research. At its heart, SoNIC utilizes commodity-off-the-shelf multicore processors to implement part of the physical layer in software and employs an FPGA board to transmit optical signal over the wire. As a result, SoNIC allows cross-network-layer research explorations by systems programmers.

8 Availability

The SoNIC platform and source code is published under a BSD license and is freely available for download at `http://sonic.cs.cornell.edu`

9 Acknowledgements

This work was partially funded and supported by an IBM Faculty Award received by Hakim Weatherspoon, DARPA (No. D11AP00266), NSF CAREER (No. 1053757), NSF TRUST (No. 0424422), NSF FIA (No. 1040689), NSF CiC (No. 1047540), and an NSF EAGER (No. 1151268). We would like to thank our shepherd, Luigi Rizzo, and the anonymous reviewers for their comments. We especially thank Tudor Marian for his contributions and for the name, SoNIC, and Ethan Kao, Erluo Li, Jason Zhao, and Roman Averbukh for reviewing and comments.

References

[1] Altera. 10-Gbps Ethernet Reference Design. http://www.altera.com/literature/ug/10G_ethernet_user_guide.pdf.

[2] Altera. PCI Express High Performance Reference Design. http://www.altera.com/literature/an/an456.pdf.

[3] Altera Quartus II. http://www.altera.com/products/software/quartus-ii/subscription-edition.

[4] Altera Stratix IV FPGA. http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/stxiv-index.jsp.

[5] Endace DAG Network Cards. http://www.endace.com/endace-dag-high-speed-packet-capture-cards.html.

[6] Hitechglobal. http://hitechglobal.com/Boards/Stratix4GX.html.

[7] IEEE Standard 802.3-2008. http://standards.ieee.org/about/get/802/802.3.html.

[8] Myricom Sniffer10G. http://www.myricom.com/sniffer.html.

[9] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. Cavallaro, and A. Sabharwal. WARP, a unified wireless network testbed for education and research. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, 2007.

[10] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 conference*, 2010.

[11] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: Design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[12] M. Casado. Reconfigurable networking hardware: A classroom tool. In *Proceedings of Hot Interconnects 13*, 2005.

[13] M. Crovella and B. Krishnamurthy. *Internet Measurement: Infrastructure, Traffic and Applications*. John Wiley and Sons, Inc, 2006.

[14] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[15] D. A. Freedman, T. Marian, J. H. Lee, K. Birman, H. Weatherspoon, and C. Xu. Exact temporal characterization of 10 Gbps optical wide-area network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010.

[16] V. Gopal, E. Ozturk, J. Guilford, G. Wolrich, W. Feghali, M. Dixon, and D. Karakoyunlu. Fast CRC computation for generic polynomials using PCLMULQDQ instruction. White paper, Intel, http://download.intel.com/design/intarch/papers/323102.pdf, December 2009.

[17] S. Gueron and M. E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode. White paper, Intel, http://software.intel.com/file/24918, January 2010.

[18] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, 2010.

[19] R. Jain, , and S. A. Routhier. Packet trains: Measurements and a new model for computer network traffic. *IEEE Journal On Selected Areas in Communications*, 4:986–995, 1986.

[20] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictble low latency for data center applications. In *Proceedings of the ACM Symposium on Cloud Computing*, 2012.

[21] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009.

[22] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transaction on Networking*, 2(1), Feb. 1994.

[23] X. Liu, K. Ravindran, B. Liu, and D. Loguinov. Single-hop probing asymptotics in available bandwidth estimation: sample-path analysis. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[24] X. Liu, K. Ravindran, and D. Loguinov. Multi-hop probing asymptotics in available bandwidth estimation: stochastic analysis. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, 2005.

[25] Y. Liu, D. Ghosal, F. Armknecht, A.-R. Sadeghi, S. Schulz, and S. Katzenbeisser. Hide and seek in time: Robust covert timing channels. In *Proceedings of the 14th European conference on Research in computer security*, 2009.

[26] Y. Liu, D. Ghosal, F. Armknecht, A.-R. Sadeghi, S. Schulz, and S. Katzenbeisser. Robust and undetectable steganographic timing channels for i.i.d. traffic. In *Proceedings of the 12th international conference on Information hiding*, 2010.

[27] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Proceedings of Microelectronics Systems Education*, 2007.

[28] T. Marian, K. S. Lee, and H. Weatherspoon. Netslices: Scalable multi-core packet processing in user-space. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2012.

[29] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[30] L. Rizzo. Netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.

[31] D. Schneider. The Microsecond Market. *IEEE Spectrum*, 49(6):66–81, 2012.

[32] J. Strauss, D. Katabi, and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003.

[33] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker. Sora: high performance software radio using general purpose multi-core processors. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 2009.

[34] R. Walker, B. Amrutur, and T. Knotts. 64b/66b coding update. grouper.ieee.org/groups/802/3/ae/public/mar00/walker_1_0300.pdf.

[35] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '95, 1995.

[36] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[37] T. Yoshino, Y. Sugawara, K. Inagami, J. Tamatsukuri, M. Inaba, and K. Hiraki. Performance optimization of TCP/IP over 10 gigabit Ethernet by precise instrumentation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[38] S. Zander, G. Armitage, and P. Branch. A Survey of Covert Channels and Countermeasures in Computer Network Protocols. *Commun. Surveys Tuts.*, 9(3):44–57, July 2007.

[39] H. Zeng, J. W. Lockwood, G. A. Covington, and A. Tudor. AirFPGA: A software defined radio platform based on NetFPGA. In *NetFPGA Developers Workshop*, 2009.