Evaluation of the ProgHW/SW Architectural Design Space of Bandwidth Estimation

Tianqi Fang, Lisong Xu, Witawas Srisa-an, and Jay Patel

University of Nebraska-Lincoln
{tfang,xu,witty}@cse.unl.edu, jpatel9@huskers.unl.edu

Abstract. Bandwidth estimation (BWE) is a fundamental functionality in congestion control, load balancing, and many network applications. Therefore, researchers have conducted numerous BWE evaluations to improve its estimation accuracy. Most current evaluations focus on the algorithmic aspects or network conditions of BWE. However, as the architectural aspects of BWE gradually become the bottleneck in multi-gigabit networks, many solutions derived from current works fail to provide satisfactory performance. In contrast, this paper focuses on the architectural aspects of BWE in the current trend of programmable hardware (ProgHW) and software (SW) co-designs. Our work makes several new findings to improve BWE accuracy from the architectural perspective. For instance, we show that offloading components that can directly affect inter-packet delay (IPD) is an effective way to improve BWE accuracy. In addition, to handle the architectural deployment difficulty not appeared in past studies, we propose a modularization method to increase evaluation efficiency.

Keywords: Bandwidth Estimation, Programmable Hardware, Evaluation

1 Introduction

Bandwidth estimation (BWE) is an essential functionality used in various network fields ranging from cloud applications to congestion control [3,4,33,27,46]. For example, BWE can improve the performance of Hadoop by optimizing the bandwidth utilization among a group of virtual machines (VMs) [27]. However, inaccurate BWE can cause packet loss and degraded throughput [46,24]. Therefore, how to improve BWE accuracy is an ever-lasting research topic.

While researchers and engineers have conducted many studies and evaluations of BWE, most of them focus on either the algorithmic designs and parameters of BWE [44,48,41,42,34] or the impact of network conditions on BWE [10,42,22], but the architectural optimizations have not been addressed adequately. Nevertheless, as networks become faster (e.g., 10Gbps, 100Gbps) and more complex nowadays, the architectural aspects of BWE become increasingly important [31]. Consider time precision as an example. BWE relies heavily on packet timing measurement. On faster networks, packet transmission time becomes shorter, which makes measurement more sensitive to timing errors caused

by interrupt coalescing [35] or OS scheduling [25] of software-based architectures. In addition, concurrency issues of software-based architectures can also interfere with BWE accuracy [14].

In the paper, we aim to fill the gap above by systematically evaluating the architectural design space of BWE, especially in the trend of programmable hardware (ProgHW)/software (SW) co-designs. The comparison between our work and related works is shown in Figure 1. Recently, one of the major architectural upgrades is that pure SW-based network stacks have gradually been replaced by ProgHW/SW co-designs, such as Azure SmartNIC [18], Microsoft Catapult [36] or AWS F1 [43]. This new ProgHW/SW paradigm can provide a group of design choices unavailable in the past to improve BWE accuracy in high-speed networks. Although a few works have used ProgHW to improve timing measurement accuracy [46,19,20], some important questions have not been studied carefully, and we aim to answer them in the paper. For example, compared with BWE optimization techniques, what are the gains and costs of ProgHW-based designs? Considering the variety of BWE components, what are the trade-offs and cost-efficiencies of different ProgHW/SW combinations?

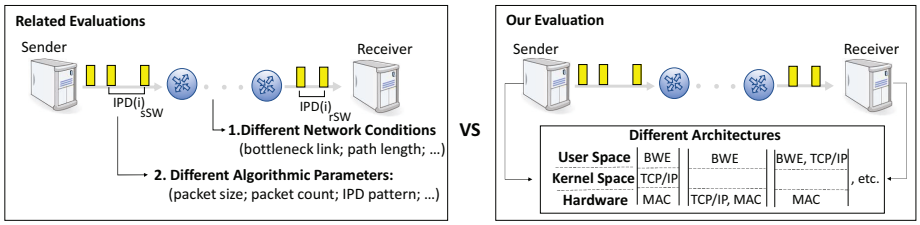


Fig. 1: Comparison between Related Works and Our Work

The main cause for the inadequate architectural evaluation is that the evaluation of the ProgHW/SW space is more laborious and challenging than evaluations carried out on pure SW. There are two reasons. First, the evaluation period of ProgHW/SW often takes much longer time than that of pure SW. Compared with SW-level compilation, ProgHW-level compilation or ProgHW offloading has many extra steps, such as netlist synthesis, place and route, and timing, power, and area constraints. These extra steps are time-consuming. Second, ProgHW/SW space provides more combinations than pure SW, which increases the workload of the architectural comparison. Specifically, for one endpoint, a different BWE algorithm may have different components, and each component can choose to stay at either ProgHW or SW to make up a different architecture. For two endpoints, a sender and a receiver can use different architectures to execute BWE. These diverse combinations increase the number of evaluation cases.

We deal with the ProgHW/SW evaluation challenges by leveraging an insight: most factors that impact BWE performance can be attributed to a single factor: inter-packet delay (IPD), and architectures of different BWE algorithms have

many common components related to IPD. Therefore, we classify and study different ProgHW/SW architectures based on how they impact IPD. IPD describes the difference among packet latencies, which differentiates our classification from other works that focus on the absolute values of packet latencies [47,21]. In addition, we modularize the BWE components that process and transmit IPD information and reuse those modules in different architectures, thereby saving time from ProgHW-level compilation and reducing evaluation difficulty.

Our contributions are summarized as follows:

- In terms of the evaluation object, we build an IPD-based classification to systematically evaluate different ProgHW/SW architectures of BWE. Furthermore, we study topics that have not been addressed well, such as heterogeneous combinations and the comparison between SW-level and ProgHW-level optimization techniques. In addition, we make several new findings as follows:
  - If the limited supply of cloud ProgHW only allows one end (sender or receiver) to be deployed with a new BWE ProgHW/SW configuration, then deployment on the receiver alone can achieve a similar effect to the two-end configuration and can only consume half the ProgHW resources at the same time.
  - Although ProgHW/SW designs and pure SW designs have comparable performance in low-speed networks, the former show much better performance in high-speed networks. Specifically, in a 100Gbps network, ProgHW/SW designs can improve average IPD accuracy by 45% (max: 64%) and average BWE accuracy by 20% (max: 35%).
  - Although offloading modules from SW to ProgHW can have better timing accuracy, offloading more does not necessarily achieve better performance in BWE. We find that the offloading of modules that directly update IPD can maximize BWE accuracy.
- In terms of the evaluation methodology, we propose an IPD-modular method that improves the evaluation efficiency by multiple times.
- We implement BWE modules in ProgHW and make them meet the requirements of BWE evaluations and be portable across different architectures.

The paper is organized as follows: Section 2 introduces our motivation and the background of BWE. Section 3 presents our IPD-based classification of architectures. Section 4 presents the implementation details of our modularization. Section 5 evaluates ProgHW/SW architectural space for BWE. Section 6 summarizes related works and Section 7 concludes the paper.

2 Motivation and Background

In this section, we first present our motivation to evaluate the ProgHW/SW architectural space of BWE and then introduce the working principles of different BWE types, in which **we show that most BWE algorithms have a close relation with IPD, which bases our classification and modularization of ProgHW/SW architectures**.

2.1 Motivation

Our motivation to do the ProgHW/SW architectural evaluation can be summarized into three points:

1) As network speed keeps growing, algorithmic optimizations alone do not meet BWE accuracy requirements anymore, and architectural factors have an increasing impact. For example, the work [23] suggests a technique of increasing measurement samples to improve BWE accuracy based on the law of large numbers. However, the study [25] shows that this technique can only keep timing error within a few microseconds in typical Linux architectures, which is not acceptable in current multi-gigabit speed where packet gaps are at sub-microsecond level.

2) The prevalence of ProgHW brings up a set of new architectural optimizations infeasible in the past, such as TCP/IP offloading, or BWE functions offloading, but existing evaluations lack a systematic comparison among those new architectures. For example, Emmerich et al. [17] only analyze kernel-bypass architectures. Besides, most evaluations also miss a comparison between ProgHW designs and traditional BWE optimizations, such as BASS [48].

3) Several topics of ProgHW designs have not been addressed sufficiently in existing works. First, although a few works leverage ProgHW to improve packet transmission accuracy [46,19,20], the costs of such practice are inadequately discussed. Balancing benefits and costs has practical value at the current stage. Compared with SW, developing and deploying BWE algorithms on ProgHW/SW often take much longer time. For example, we find that compiling a typical BWE algorithm pathload [23] only takes a few seconds on SW, but it takes 8-9 hours on a ProgHW/SW architecture to complete. The reason is that ProgHW compilation does not only need to translate a high-level language to a netlist but also needs to guarantee the closure of timing, power, and area. Second, heterogeneous combinations are less-studied where the sender and the receiver have different architectures. This topic is important because high-end ProgHW is limited and expensive at the current time [30], and there might not be enough ProgHWs for both endpoints.

2.2 Bandwidth Estimation Background

Network bandwidth is an attribute of a network path, and it specifies how fast a user can send data through this path. Bandwidth is useful information, but it is often hard to obtain from routers due to technical and privacy issues, so people develop various BWE algorithms that can estimate this information from endpoints.

This part presents the working principles and classification of major BWE algorithms. Please refer to Table 1 for definitions of symbols. There are three metrics for bandwidth: capacity (bw-capa), available bandwidth (avai-bw), and achievable throughput [24]. Capacity is the maximum rate that a network path can support. In a real network path, some portion of capacity may be occupied by cross traffic, then the rest capacity for our usage is called available bandwidth.

Format	
$Symbol(i)_l$	i denotes the i-th packet and l denotes any of four locations: sSW(sender's SW), rSW(receiver's SW), sHW(sender's NIC port), and rHW(receiver's NIC port)
$\Delta Symbol(i)_l$	$Symbol(i+1)_l - Symbol(i)_l$
Symbol	
$t(i)_l$	Measured timepoint of i-th packet at location l
$t(i)_l^R$	Real timepoint of i-th packet at location l
$tdr(i)_l$	Clock drift at l: $t(i)_l - t(i)_l^R$
$d(i)_{SW}$	Measured delay of i-th packet from sender to receiver by SW: $t(i)_{rSW} - t(i)_{sSW}$
$d(i)_{HW}$	Measured delay of i-th packet from sender to receiver by HW: $t(i)_{rHW} - t(i)_{sHW}$
$d(i)^R$	Real delay: $t(i)_{rHW}^R - t(i)_{sHW}^R$
$IPD(i)_l$	$IPD(i)_l = \Delta t(i)_l = t(i+1)_l - t(i)_l$
$\Theta IPD(i)_{SW}$	$IPD(i)_{rSW} - IPD(i)_{sSW}$
$\Theta IPD(i)_{HW}$	$IPD(i)_{rHW} - IPD(i)_{sHW}$
$n(i)_s^R$	Real delay from SW to ProgHW on sender
$n(i)_r^R$	Real delay from ProgHW to SW on receiver

Table 1: Symbols and Notations

While the first two metrics only consider network speed, the achievable throughput also considers an endpoint's processing speed and protocols. In a nutshell, achievable throughput indicates the maximum throughput that a system can achieve under a given protocol, network speed, and processing speed.

From the perspective of working principles, BWE algorithms can be classified into the packet-pair type and the packet-train type as shown in Figure 2. The details of each algorithm in Figure 2 can be found in [25]. The two types differ in timing features but share a basic idea: a sender sends out a set of packets with a pre-defined timing feature. If the sending rate exceeds the bandwidth, a receiver will detect a change in the timing feature when those packets arrive. BWE algorithms iteratively adjust sending rates to find the turning point where the change occurs, and the turning point is the estimated bandwidth value.

We first define IPD as follows. Examples of IPD such as $IPD(i)_{sSW}$ and $IPD(i)_{rSW}$ are shown in Figure 2.

Definition 1. *Inter-packet Delay (IPD) is the timing delay between any two consecutive packets.*

For the **packet-pair** type, a pair of packets are sent out back-to-back or in a pre-defined IPD value. Then, a receiver captures the receiving IPD and compares it with the pre-defined IPD to infer bw-capa or avai-bw. The formal expression is as follows:

Definition 2. *A packet-pair BWE algorithm is a function of the difference between the receiving and sending IPDs. Assume there are n pairs of packets (i.e., 2×n packets) in transmission, the estimated value BW is as follows:*

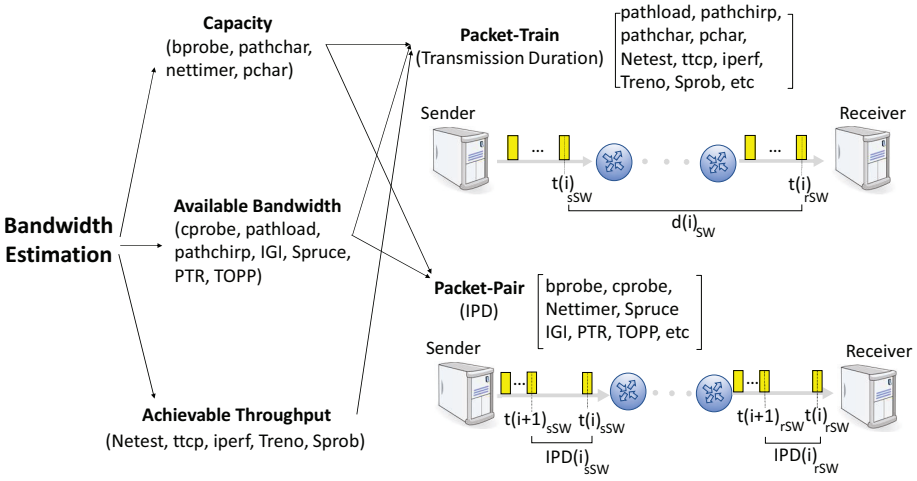


Fig. 2: Bandwidth Estimation Classification

$$BW = f(\Theta IPD(1)_{SW}, \Theta IPD(3)_{SW}, \cdots, \Theta IPD(2n-1)_{SW})$$

For the **packet-train** type, a sequence of packets are transmitted from a sender to a receiver. For the i-th packet, the difference between its receiving timepoint $t(i)_{rSW}$ and sending timepoint $t(i)_{sSW}$ is called its delay $d(i)_{SW}$. The packet-train type analyzes the change in each packet's delay to estimate the bandwidth. This type includes different patterns of probing rate such as constant rate [23] or exponential rate [38]. The formal expression is as follows:

Definition 3. *A packet-train BWE algorithm is a function of changes in packet delays from the sender to the receiver. Assume there are n packets in transmission, the estimated value BW is as follows:*

$$BW = f(\Delta d(1)_{SW}, \Delta d(2)_{SW}, \cdots, \Delta d(n-1)_{SW})$$
$$where\ \forall i \in [1, n), \Delta d(i)_{SW} = IPD(i)_{rSW} - IPD(i)_{sSW}$$

We observe that the delay change $\Delta d(i)_{SW}$ of the i-th packet can be expressed by the difference between the receiver's IPD and the sender's IPD: $IPD(i)_{rSW} - IPD(i)_{sSW}$. We can derive this expression by using timepoints as follows:

$$\begin{aligned}\Delta d(i)_{SW} &= (t(i+1)_{rSW} - t(i)_{rSW}) - (t(i+1)_{sSW} - t(i)_{sSW}) \\ &= IPD(i)_{rSW} - IPD(i)_{sSW}\end{aligned}$$

As shown above, there is an important observation that most BWE algorithms rely on relative timing or changes in timing rather than absolute timing. Furthermore, the relative timing features of both BWE types can be converted into

a unified form: IPD, which motivates our ProgHW/SW architectural evaluation to focus on IPD rather than absolute timing quantities.

3 Our Classification of Bandwidth Estimation Architectures

In this section, we first present our IPD-based classification of ProgHW/SW architectures for BWE. Meanwhile, we explain the functionality of each module in those architectures. The implementation details of those modules are shown in Section 4. Then, we analyze the timing context of ProgHW/SW architectures and explain how to keep IPD accuracy in such context.

According to our investigation of different types of BWE, we have an important insight that the factors that impact BWE performance can be attributed to a single factor: IPD, and architectures of different BWE types share many components in common to process and transmit IPD. Therefore, we study different architectures based on how they impact IPD, and we modularize those common components to reduce the evaluation difficulty caused by the time-consuming ProgHW-level compilation. The insight is further discussed in Section 3.2.

3.1 IPD-based Architectural Classification

We identify and modularize common BWE components that process and transmit IPD information, and our classification of different ProgHW/SW architectures is based on different allocations of those modules. Specifically, there are three modules for the sender: packet generator, IPD modulator, and IPD transceiver, and three modules for the receiver: IPD gauge, IPD transceiver, and IPD processor. In a BWE process, a sender uses an IPD modulator to set predefined timing features, and a receiver uses an IPD gauge and an IPD processor to measure and analyze the change in those timing features. We will illustrate those modules by using the traditional SW architecture of type 1.

Type 1 (No IPD Optimization) This type does not involve any specialized optimization to improve IPD accuracy. One feature of type 1 is that most BWE modules locate in user space. The traditional SW architecture is a representative of this type, whose architecture is shown in Figure 3. There are several steps in a complete procedure of BWE. On a sender, the packet generator generates a sequence of packets. Then, the sender's IPD modulator specifies the IPD information of packets through a system timer. Next, these packets pass through the IPD transceiver whose major component is the TCP/IP stack, and they reach the MAC (Ethernet) TX port. After being transmitted through the network path, these packets reach a receiver. On the receiver, the IPD transceiver uploads IPD information, and then the IPD gauge module measures the IPD of packets. Lastly, the measured IPD is used by the IPD processor to infer network bandwidth.

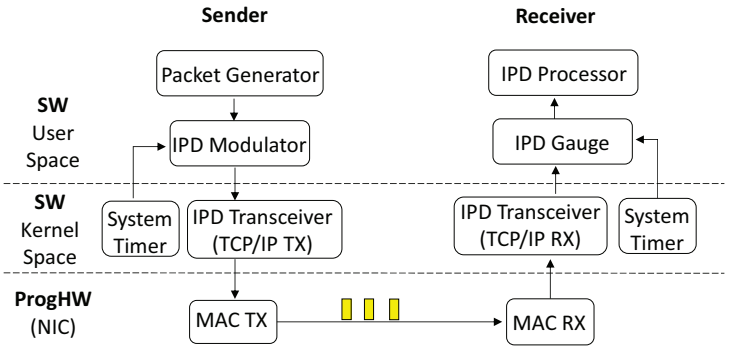


Fig. 3: Type 1 (No IPD Optimization)

Type 1 architecture finds it difficult to keep IPD accuracy because of many timing-noise factors. For the system timer, there is a timestamp counter (TSC) in the kernel space, which is accessed by timing operations such as the system call function *gettimeofday*. On the sender, after sending one packet, the process keeps polling TSC to wait for a specified IPD. Once that IPD is reached, the sender timestamps and sends the next packet. On the receiver, once the packet arrives, the process gets the arrival time by reading the current value of TSC. The TSC access operation on both ends generates about 1–2 μs timing noise [25]. The IPD transceiver may generate timing noise of several microseconds, and it covers the TCP/IP stack, PCIe switching, and other transceiving services. The BWE functions or other daemon services may also disturb the accuracy of the timing [37]. Because of these inaccuracy factors, it becomes increasingly difficult for the traditional SW architecture to measure IPD accurately with the trend of faster network speed and shorter IPD.

Type 2 (IPD Noise Mitigation) The feature of this type is that both IPD modulator and IPD gauge are still in user space as type 1, but architectures are improved to mitigate timing noise. There are several choices to do the mitigation: TCP/IP stack offloading, kernel bypass, or BWE functions offloading as shown in Figure 4. Both TCP/IP stack offloading and kernel bypass aim to reduce the number of data copies in packet transmission so that timing is more stable, and performance is better. The difference between these two is that TCP/IP offloading moves TCP/IP stack down to ProgHW while kernel bypass moves it up to user space. TCP/IP offloading has several related works [7,39].

BWE functions offloading, to the best of our knowledge, has rarely been studied, so we implement our custom version of this architecture by offloading the packet generator and IPD processor modules down to ProgHW. The implementation details are presented in Section 4. BWE functions offloading is based on TCP/IP offloading, which means that TCP/IP stack is also offloaded to ProgHW in the BWE functions offloading architecture.

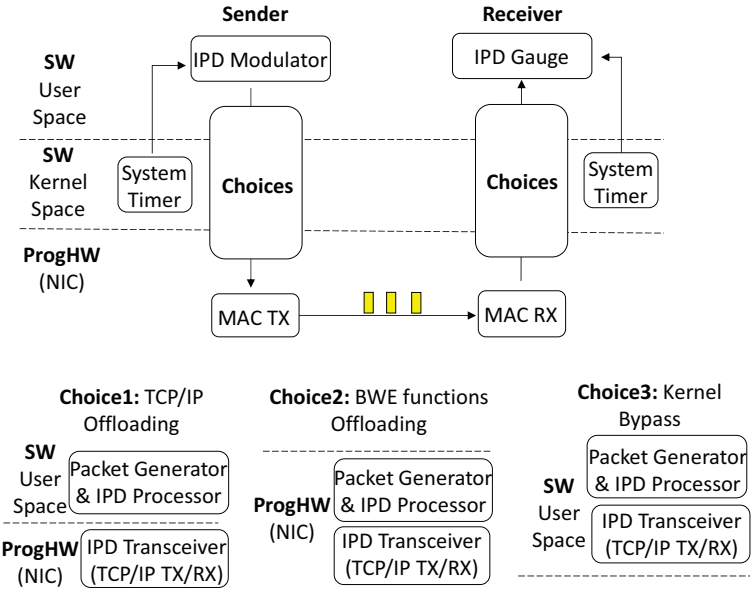


Fig. 4: Type 2 (IPD Noise Mitigation)

Type 3 (IPD HW Modulation) The feature of this type is that both the IPD modulator and IPD gauge are placed close to NIC port. The purpose of such placement is to restore IPD information tampered by the timing noise of the ProgHW-SW transmission path. Specifically, both the IPD modulator and IPD gauge modules adopt a HW timer rather than a SW timer to improve timing stability. On the sender, ProgHW modulates the IPD of packets according to the IPD specification of SW. On the receiver, ProgHW records receiving IPD and sends the IPD information to SW.

This type uses the combination of stream control signals and the timer of ProgHW to achieve accurate IPD modulation and gauge [20,16]. Specifically, on the sender, a HW timer is used to measure the delay of packet transmission. Then, the delay is compared with a specified IPD. If the delay is smaller, stream control signals block the following packets until the specified IPD is reached. On the receiver, a look-up table is dedicated to storing IPD information of packets. The receiving IPD information is then uploaded to the IPD processor.

3.2 ProgHW/SW Timing Context Analysis

In this part, we analyze the timing context of ProgHW/SW architectures and provide a criterion to keep IPD accuracy. We first build up a model to summarize timing-noise factors and then explain how our criterion addresses those factors.

As shown in Figure 6, we use an example of transmitting packets from sender to receiver to illustrate the ProgHW/SW timing context. Please refer to

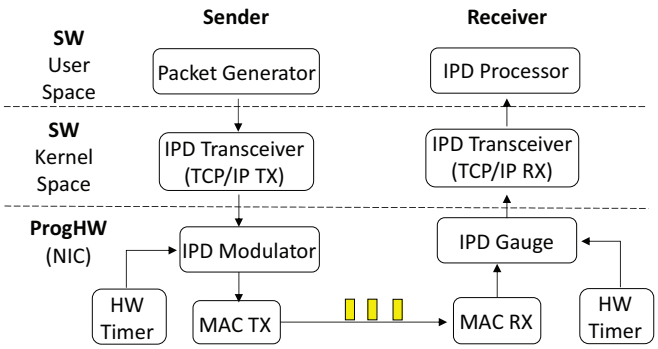


Fig. 5: Type 3 (IPD HW Modulation)

Table 1 for definitions of symbols. Because of timing-noise factors in ProgHW/SW, timing measurement is sometimes inaccurate. If a SW-based design aims to measure delay $d(i)^R$ of packets transmission from sender to receiver, its measured value $d(i)_{SW}$ may differ from the real value $d(i)^R$. Specifically, a packet is first transmitted from SW to ProgHW and experiences timing noise $n(i)_s^R$. Then, the sender's ProgHW sends the packet out and the receiver's ProgHW captures it. Lastly, the packet is uploaded to the receiver's SW and experiences receiving timing noise $n(i)_r^R$. The difference between the measured value and real value (i.e., measured error): $d(i)_{SW} - d(i)^R$ can be expanded as $(n(i)_s^R + n(i)_r^R) + (tdr(i)_{sSW} + tdr(i)_{rSW})$, where $n(i)_s^R$ and $n(i)_s^R$ are generated by timing-noise factors such as system scheduling, data copy, and many others [32,25]. One thing worth noting is that both $n(i)_s^R$ and $n(i)_s^R$ are variables, so they may vary for different packets.

Clock Drift: Another timing-noise factor comes from clock drift such as $tdr(i)_{sSW}$ or $tdr(i)_{rSW}$. Because of imperfections and accessing noise in a real-world timer, there might be a difference between a real-world timer and an ideal reference timer [15]. In the ProgHW/SW timing context, there are four timers (a sender's SW and ProgHW, and a receiver's SW and ProgHW) and they usually have different frequencies and accessing manners. Therefore, clock drift will happen if the four timers are not synchronized with each other. However, it is difficult to achieve synchronization by traditional methods such as NTP. The reason is that packet transmission time is nanosecond-level while NTP takes milliseconds.

Because of timing-noise factors shown in Figure 6, SW timing could differ from real timing in packet transmission, and it is difficult to make those two the same. Fortunately, it is unnecessary to keep absolute timing accurate. According to Definition 2 and 3, major BWE algorithms depend on relative timing values rather than absolute timing values to carry out estimation. Therefore, the timing accuracy requirements of both the packet-pair and the packet-train BWE types are based on relative timing values. Specifically, their requirements shown below

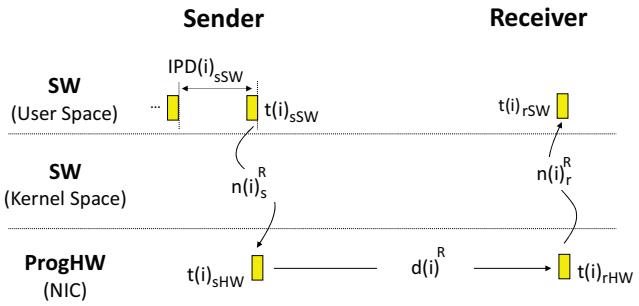


Fig. 6: ProgHW/SW Timing Context

state that the specified or measured relative timing values are equal to the real values.

Definition 4. *Packet-pair timing accuracy requirement: Assume there are n pairs of packets ($2\times n$ packets) in transmission, the requirement is shown below.*

$$\forall i \in [1,n], k = 2i-1 \;\; \Theta IPD(k)_{SW} = \Theta IPD(k)^R$$

Definition 5. *Packet-train timing accuracy requirement: Assume there are n packets in transmission, the requirement is shown below.*

$$\forall i \in [1,n), \Delta d(i)_{SW} = \Delta d(i)^R$$

Furthermore, we observe that the timing accuracy requirements of both types can be converted into a unified form: the accuracy of IPD. For the packet-pair type, its relative timing value is already IPD. For the packet-train type, its relative timing value: delay $\Delta d(i)$ can be expanded to a function of IPD. This expansion is shown in Theorem 2. In a nutshell, our IPD-based criterion states that a ProgHW/SW architecture can meet the timing accuracy requirements of both types by synchronizing IPD between ProgHW and SW on both sender and receiver. We formalize our criterion with two theorems shown below.

Assumption 31. The clock jitter in ProgHW is negligible for IPD measurement. Formally, if there are n packets in transmission, then we have

$$\forall i \in [1,n-1], tdr(i+1)_{sHW} - tdr(i)_{sHW} = 0$$
$$tdr(i+1)_{rHW} - tdr(i)_{rHW} = 0$$

Note that clock jitter is different from clock drift. Clock jitter means temporal timing variation while clock drift means spatial timing variation. According to the past study [11], the clock jitter of ProgHW is around several picoseconds, which accounts for less than 0.1% of IPD measurement.

Theorem 1. *$\forall i \in [1,n]$, $k = 2i-1$, if $IPD(k)_{sSW} = IPD(k)_{sHW}$ and $IPD(k)_{rSW} = IPD(k)_{rHW}$, then the Packet-pair timing accuracy requirement (Definition 4) can be satisfied.*

Proof. To satisfy the requirement, we need to prove that $\Theta IPD(k)_{SW} = \Theta IPD(k)^R$ for all pairs of probing packets. We first expand $\Theta IPD(k)_{SW}$ and $\Theta IPD(k)^R$ as follows.

$$\begin{aligned}\Theta IPD(k)_{SW} &= IPD(k)_{rSW} - IPD(k)_{sSW} \\ \Theta IPD(k)^R &= IPD(k)_{rHW} - IPD(k)_{sHW} \\ &\quad + \Delta tdr(k)_{sHW} - \Delta tdr(k)_{rHW}\end{aligned}$$

According to Assumption 31, the values of $\Delta tdr(k)_{sHW}$ and $\Delta tdr(k)_{rHW}$ can be both zero. Therefore, If $IPD(k)_{sSW} = IPD(k)_{sHW}$ and $IPD(k)_{rSW} = IPD(k)_{rHW}$, then $\Theta IPD(k)_{SW} = \Theta IPD(k)^R$.

Theorem 2. *$\forall i \in [1, n)$, if $IPD(i)_{sSW} = IPD(i)_{sHW}$ and $IPD(i)_{rSW} = IPD(i)_{rHW}$, then the Packet-train timing accuracy requirement (Definition 5) can be satisfied.*

Proof. To satisfy the requirement, we need to prove that $\Delta d(i)_{SW} = \Delta d(i)^R$ for all probing packets. We first expand $\Delta d(i)_{SW}$ and $\Delta d(i)^R$ as follows.

$$\begin{aligned}\Delta d(i)_{SW} &= d(i+1)_{SW} - d(i)_{SW} \\ \Delta d(i)^R &= d(i+1)^R - d(i)^R\end{aligned}$$

The One-way delays $d(i+1)_{SW}$ and $d(i)_{SW}$ can be expressed in the form of timepoints:

$$\begin{aligned}d(i+1)_{SW} &= t(i+1)_{rSW} - t(i+1)_{sSW} \\ d(i)_{SW} &= t(i)_{rSW} - t(i)_{sSW}\end{aligned}$$

Therefore, their difference can also be expressed in the form of timepoints:

$$\begin{aligned}\Delta d(i)_{SW} &= (t(i+1)_{rSW} - t(i)_{rSW}) - (t(i+1)_{sSW} - t(i)_{sSW}) \\ &= IPD(i)_{rSW} - IPD(i)_{sSW}\end{aligned}$$

Following the similar deduction procedure and considering Assumption 31, we can get

$$\Delta d(i)^R = IPD(i)_{rHW} - IPD(i)_{sHW}$$

Therefore, If $IPD(i)_{sSW} = IPD(i)_{sHW}$ and $IPD(i)_{rSW} = IPD(i)_{rHW}$, then $\Delta d(i)_{SW} = \Delta d(i)^R$.

As shown above, synchronizing IPD between ProgHW and SW is critical to achieving timing accuracy for different BWE algorithms because of the relative timing feature of BWE. Architectures of type 3 satisfy the criterion, but type 1 and 2 do not strictly follow the criterion.

4 Implementation of Modules

According to Section 3, different architectures share many modules in common to process and transmit IPD information, and this section introduces our implementation details of those modules. We have two requirements for those modules: (1) they are portable across different architectures, and (2) they are suitable for BWE evaluations. The IPD transceiver module [7,8] and packet generator [16,40] of existing works satisfy those requirements, but the IPD modulator, IPD gauge, and IPD processor do not, so we focus on the last three. We use our IPD processor to make up the architecture of BWE functions offloading (type 2), and we use our IPD modulator and IPD gauge to make up the architecture of IPD HW modulation (type 3).

4.1 Preliminaries of FPGA

To better understand the implementation details, we present an introduction of two ProgHWs used in our work: NetFPGA-SUME [50] and Alveo U280 FPGA [2]. FPGA is a widely used ProgHW. NetFPGA-SUME and Alveo U280 are network-oriented FPGAs for high-speed networking development.

The layouts of NetFPGA and Alveo are presented in Figure 7. Both FPGAs use a group of modules to build up specified functionalities. For NetFPGA, the MAC RX and MAC TX modules transfer packets between a network and a FPGA while the DMA RX and DMA TX modules transfer packets between a FPGA and a host computer. Besides, the MAC RX and MAC TX modules have four copies. The User Data Path module serves as a flexible packet buffer where users can design their custom functionalities. For Alveo U280, TCP/IP is often implemented on the Network Kernel module. The CMAC and GT kernels cooperate to achieve 100Gbps network speed and they usually operate at 200MHz or more. The User Kernel module is open for users to create custom designs such as BWE algorithms.

FPGA designs have two types of communication: the communication between a host computer and a FPGA, and the communication among modules on a FPGA. The former uses peripheral component interconnect express (PCIe) interface while the latter uses Advanced Extensible Interface (AXI). AXI has two types: AXI-Lite and AXI-Stream. AXI-Lite is used to configure the registers of each module and AXI-Stream is used to control the transmission of packets (i.e., data). The AXI-Lite master module in Figure 7 serves as an arbiter for different AXI-Lite signals. For more details on AXI, please refer to [1]. Packets are buffered in a data structure called first-in&first-out buffer (FIFO), whose control signals are often combined with AXI-Stream to regulate the start and end of a new packet.

There are two types of memory resources that are widely used in packet transmission. The first one is on-chip memory named Block RAM (BRAM), and the other is external memory resources including high bandwidth memory (HBM) and Double Data Rate (DDR) memory.

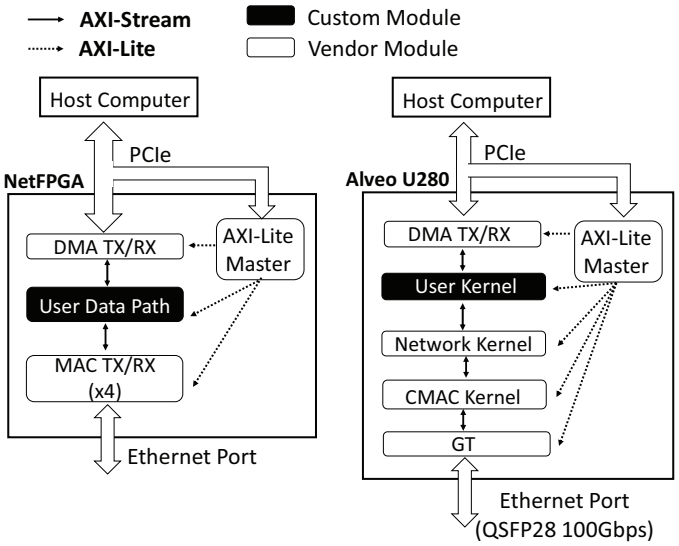


Fig. 7: Layout of NetFPGA (Left) and Alveo U280 (Right)

4.2 Implementation Details

To make modules portable across different architectures, all of these modules follow a unified interface: AXI. Specifically, we use AXI-Stream to control the transmission of packets, and AXI-Lite to set parameters, such as sending rate or packet count.

The IPD processor module has rarely been studied in the context of ProgHW, and we implement a key BWE function, named pair-wise comparison test (PCT) to build this module. PCT is used to examine if the BWE sending rate saturates the available bandwidth of a network path. Specifically, if the ratio approaches a threshold between the number of receiving IPDs that are larger than sending IPDs and the total of receiving IPDs, the sending rate is considered to be larger than the available bandwidth, and a BWE algorithm will begin turning down its sending rate to avoid network congestion. In addition, we design two counters on the receiver side. One is used to count the total number of receiving IPDs, and the other is used to count the number of receiving IPDs larger than sending IPDs. We compare two counters to check if the PCT condition is satisfied. Furthermore, we use the direct access mechanism in PCIe for IPD transfer between the host memory and the FPGA board. This way consumes fewer memory resources than creating a dedicated global memory for transfer. Then, we build up the architecture of BWE functions offloading (type 2) by combining our IPD processor with the IPD transceiver from the design [7] for NetFPGA-SUME and from the design [8] for Alveo U280.

For the IPD modulator and IPD gauge, we refer to the design: Combov [20], and we make two changes to suit BWE evaluations. First, we enlarge the packet

FIFO size of these modules. The default size is 8 packets × 64 bytes, which is not large enough to hold hundreds of packets in some packet-train BWE algorithms, such as pathload [23]. Thus, we resize the FIFO to 1000 packets × 1500 bytes which are larger than the maximum value of most BWE algorithms. Second, to avoid overflow, we set a proper bit width for both sender's and receiver's IPD arrays. These arrays are used to store sending and receiving IPDs. According to our study, both the packet-train and the packet-pair types spend less than 30 minutes doing estimation and the default timing precision of two FPGA boards is $8ns$ [2,50], which means that the width should be at least 38 bits to store IPDs ($30mins \times 60 \times 10^9/8ns < 2^{38}$). In addition, for the IPD modulator, we use the read-valid signal of AXI-Stream to make sure that packet transmission follows the specified IPD. For the IPD gauge, we use the transmission-last signal of AXI-Stream to record the arrival time of a new packet. We use our implementation of IPD modulator and IPD gauge to build the IPD HW modulation architecture (type 3). Besides, we use BRAM to implement packet FIFOs for packet transmission among FPGA modules.

5 Evaluation

This section presents evaluations of both our IPD-modular evaluation method and the ProgHW/SW architectural space of BWE. Our ProgHW/SW source code of the experiments is available at [9]. This work does not raise any ethical issues.

5.1 Evaluation Environments

We use Alveo U280 FPGA [2] and NetFPGA-SUME [50] to examine different ProgHW/SW architectures. Alveo U280 FPGA is used for 100Gbps experiments in Open Cloud Testbed (OCT) [30]) while NetFPGA-SUME is used for less than or equal to 10Gbps experiments in our local testbed built with mininet version 2.2.2. Specifically, we deploy two Alveo U280 FPGA boards on two VMs of OCT and each is equipped with 32 Virtual CPU cores and 64GB RAM. We deploy NetFPGA-SUME on a Dell Precision 3630 machine with Intel Xeon-E5 16 Cores and 64GB RAM. The network topology of NetFPGA-SUME is shown in Figure 8 where two nodes on the top generate cross traffic and two nodes on the bottom run BWE algorithms and optionally run other concurrent applications. The testbed for Alveo U280 is similar to Figure 8 with two differences. First, the bottleneck link is a Dell Z9100 100G switch. Second, because a user can create no more than two nodes in OCT, there are no cross-traffic nodes (i.e., no Node2 and Node3 in Figure 8). The Operating system is Ubuntu 2020.4 LTS with the Linux kernel 5.4. To compile ProgHW source code, we use Xilinx Vivado Design Suite v2020.1. We choose two representative BWE algorithms: bprobe [13] of the packet-pair type and pathload [23] of the packet-train type for our experiments.

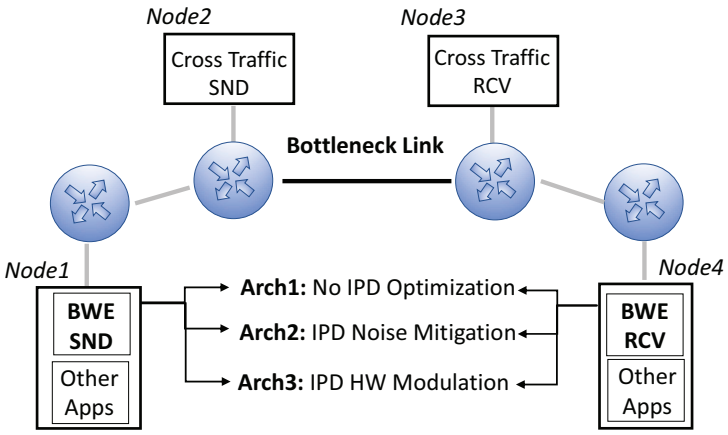


Fig. 8: Experiment Network Topology

5.2 Evaluation of IPD-Modular Method

In this section, we compare the efficiency of our modular evaluation method against the dedicated evaluation method of past works [42,34]. The dedicated evaluation method means that we compile every design for every new study case. In contrast with that, our method converts common components of different BWE algorithms into several modules that can be reusable in different ProgHW/SW architectures. In this way, if modules of two architectures overlap, we only need to compile those common modules once and reuse them for the other architecture. For example, as shown in Section 3, the architectures of BWE functions offloading and TCP/IP offloading share ProgHW-based TCP/IP stack, so we only need to compile ProgHW-based TCP/IP once. The compilation time (hrs: hours) comparison is shown in Table 2. The compilation of IPD modulator and IPD gauge in Combov takes 2 hours. In addition, the main difference between the packet-pair and the packet-train types is on packet generator and IPD processor modules, so we spent another 2 hours recompiling these two for the packet-train type after finishing the packet-pair type.

Summary: The IPD-modular method greatly reduces total compilation time. Spotting and reusing common components among different BWE designs can save time from ProgHW recompilation. As the number of studied algorithms and architectures goes up, the advantage of the IPD-modular evaluation method can become bigger.

5.3 Evaluation of ProgHW/SW Architectures

Group1 - IPD Accuracy and Cost Effectiveness of Different Architectures: In this group, we evaluate how well different ProgHW/SW architectures keep IPD accuracy. We use the hardware timestamp functionality in FPGAs to specify IPD values and set the clock cycle to 8 ns in this experiment. Results

BWE Type	Arch	Dedicated Eval	IPD-Mod Eval
Packet-pair (bprobe)	BWE Func Offloading	9 hrs	9 hrs
	TCP/IP Offloading	8 hrs	/
	Combov	8 hrs	2 hrs
Packet-train (pathload)	BWE Func Offloading	9 hrs	2 hrs
	TCP/IP Offloading	8 hrs	/
	Combov	8 hrs	/
Total		50 hrs	13 hrs

(Note: "/" means no need to do recompilation)
Table 2: Evaluation Efficiency Comparison

are shown in Figure 9. We use the formula below to define the IPD measurement error (IPD_{err}) where #IPD is the number of measured IPD samples. For each experiment set, we collect 40 samples. Furthermore, we check the cost efficiency of each architecture by metrics of offloading workload and ProgHW resources consumption. ProgHW resources are described by three key metrics: the number of Look-up Tables (LUTs), Flip-flops (FFs), and BRAMs. The results are shown in Table 3.

$$IPD_{err} = \sqrt{\frac{1}{\#IPD} \cdot \sum_{i=1}^{\#IPD} (IPD[i] - IPD_{actual})^2}$$

For both sender and receiver, according to Figure 9, IPD noise mitigation (type 2) and IPD HW modulation (type 3) have better IPD accuracy than the pure SW architecture, especially in short IPD (e.g., 120ns). The advantage of type 2 comes from the kernel-bypass effect, which requires less data copy from SW to ProgHW. However, this effect cannot completely remove the IPD noise of systems. Type 3 shows better IPD accuracy than type 2, and the former can keep IPD error within 1%. The main reason is that accessing the ProgHW timer is more stable than accessing the SW timer. From the experiment, we find that IPD restoration is more effective than noise mitigation, and this result is consistent with our analysis in Section 3.2. In addition, we also find that type 3 does not completely remove IPD measurement errors. This is because the type 3 design needs 1 clock cycle to read the measured IPD to a register. In terms of cost effectiveness, we find that more offloading does not necessarily lead to better performance in ProgHW/SW designs. As shown in Table 3, although Combov uses 70% fewer resources than BWE functions offloading architecture,

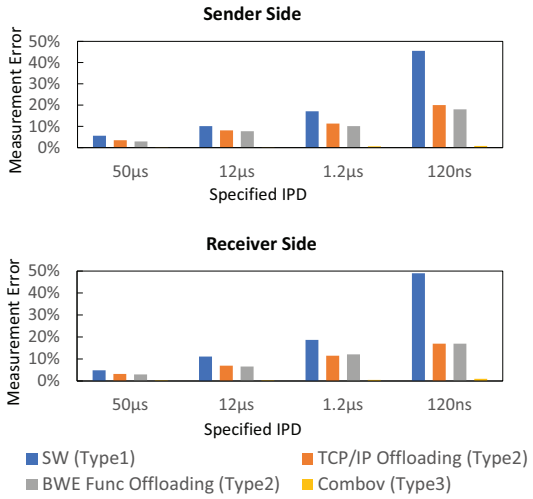


Fig. 9: IPD Measurement Error of Different Architectures

it can achieve 20% more accurate IPD than BWE functions offloading. Thus, we suggest engineers prioritize offloading modules that can directly update or recover IPD.

Group1 - Summary: In terms of IPD accuracy, type 3>type 2>type 1. In addition, more offloading does not necessarily have better performance. Specifically, although TCP/IP offloading and BWE functions offloading consume nearly 10 times more ProgHW resources than Combov, the former architectures are less accurate than the latter.

Group2 - BWE accuracy of Different Architectures: In this group, we check if the IPD accuracy improvement of ProgHW/SW architectures can benefit BWE performance. We evaluate both the packet-pair and the packet-train types. This group of experiments do not involve concurrent applications. Influences of cross traffic and concurrent applications will be discussed in the following experiment groups. Besides, the actual available bandwidth equals the capacity of the bottleneck link. In addition, we include DPDK [5], one of the most widely used kernel-bypass frameworks as a reference in this group. The size of the packets is 1408 bytes, and the data width of AXI-Stream is 512 bits. Figure 10a and 10b show the BWE performance of different ProgHW/SW architectures. The x-axis represents the actual available bandwidth, and the y-axis represents the estimation value.

From Figure 10a and 10b, we find that higher IPD accuracy can lead to higher BWE accuracy on both the packet-pair and the packet-train types. In the 100Gbps network, Combov of type 3 can even improve bprobe accuracy by 35% from pure SW architecture. In addition, we find that type 2 and type 3 can reduce BWE estimation variance. For example, in the 1Gbps network, Combov of type 3 can keep the variance from the specified value within 60Mbps while the variance

	Combov (Mod & Gauge Offloading)	**TCP/IP Offloading**	**BWE Func Offloading**
Offloading Code Lines	100	1500	1800
LUTs	14013	137286	146493
FFs	53623	222838	233761
BRAMs	26	468	480

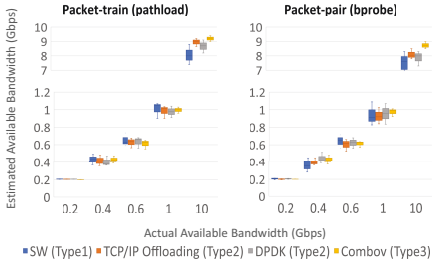
Table 3: Resources Consumption and Offloading Workload Comparison

of pure SW reaches 200Mbps. This is because of the timing-stability feature of ProgHW. We also find that relocating TCP/IP alone (e.g., TCP/IP offloading or DPDK) is not enough to achieve the best BWE accuracy. The reason is that relocating TCP/IP cannot completely remove timing noise in user space. However, the main advantage of DPDK is that it generally requires less time for development and deployment since it is a SW-based solution. Furthermore, we find that the packet-pair type gets more performance improvement than the packet-train type in type 2 and 3. This is probably because the packet-pair type only uses a single IPD sample rather than multiple samples to estimate bandwidth, so the packet-pair type is more sensitive to timing noise compared with the packet-train type.

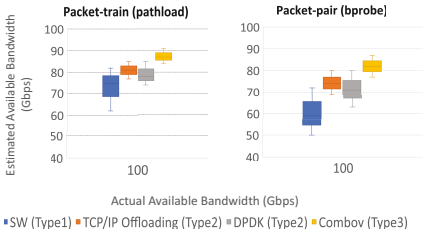
Group2 - Summary: In terms of BWE accuracy, type 3 achieves the best performance for both the packet-pair and the packet-train types, and the advantage of ProgHW-based architectures becomes bigger as networks become faster. In addition, although DPDK has comparable average performance to TCP/IP offloading, the latter can achieve a smaller estimation variance.

Group3 - Heterogeneous Combinations: This group studies the BWE performance of heterogeneous combinations where the sender and receiver have different architectures. This study is important for users to save costs and effectively use ProgHW resources. At the current stage, high-end ProgHW is expensive, and its supply is limited [30], so there might be a situation where not every endpoint can be equipped with an expected ProgHW/SW configuration.

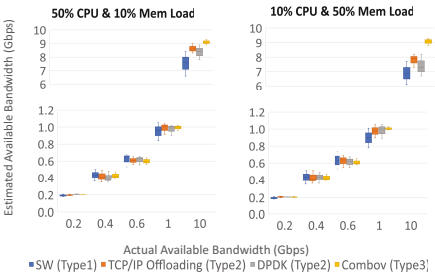
We use Combov (type 3) in this group. Experiment results are shown in Figure 11. We use the formula below to define the BWE measurement accuracy (BWE_{acc}) where #BWE is the number of measured BWE samples. For each experiment set, we collect 20 samples. According to Figure 11, the heterogeneous combination of SW sender and Combov receiver can achieve at least 80% performance of the architecture where both endpoints are equipped with Combov.



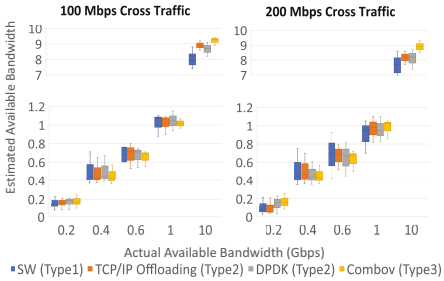
(a) Bandwidth Estimation of Different Architectures ($\leq$ 10Gbps)



(b) Bandwidth Estimation of Different Architectures (100Gbps)



(c) Bandwidth Estimation (pathload) with Concurrent Applications



(d) Bandwidth Estimation (pathload) with Cross Traffic

Fig. 10: Bandwidth Estimation Experiment Results

$$BWE_{acc} = 1 - BWE_{err}$$

$$BWE_{err} = \sqrt{\frac{1}{\#BWE} \cdot \sum_{i=1}^{\#BWE} (BWE_i - BWE_{actual})^2}$$

One possible explanation for this phenomenon is that the duty of the sender and the receiver is different. The sender aims to saturate avai-bw by continuously increasing the rate to transmit packets. The receiver uses IPD information to calculate bandwidth. The SW-based sender uses interrupt coalescing [35] and the Combov sender uses packet buffering. Both those techniques can achieve fast rate to saturate avai-bw, so the sender replacement does not have a significant difference. However, interrupt coalescing on the receiver can damage each packet's timing information, which reduces BWE accuracy. If limited ProgHW resources only allow one endpoint to use ProgHW, then deployment on the receiver may achieve better performance than on the sender.

Group3 - Summary: The receiver side has a larger impact on BWE performance than the sender side in terms of ProgHW/SW configurations. This

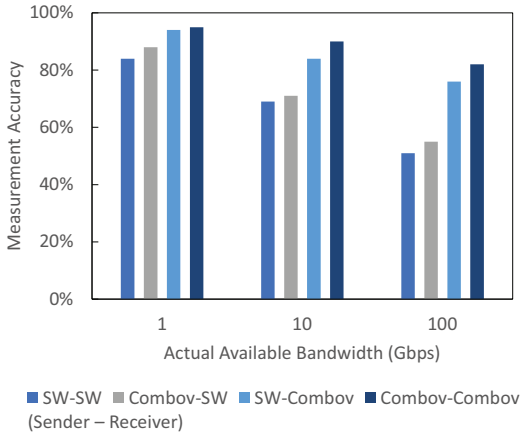


Fig. 11: BWE Accuracy of Heterogenous Combinations

finding is useful to save costs. Specifically, we can assign ProgHW/SW configurations to the receiver alone to achieve comparable performance to the two-end configurations.

Group4 - Impact of Concurrent Applications: BWE programs sometimes inevitably share CPU and memory resources with other applications on the same machine. In this group, we evaluate how different ProgHW/SW architectures perform with the existence of concurrent applications. We use Look-Busy [6] to simulate both CPU-intensive and I/O-intensive applications. We set 50% CPU load and 10% memory load for a CPU-intensive application and 10% CPU load and 50% memory load for an I/O-intensive application. The results are shown in Figure 10c.

We find that type 3 can greatly resist the influence of either CPU-intensive or memory-intensive applications compared with the other two types. One possible reason is that its IPD restoration mechanism can correct timing errors caused by concurrent applications in SW. Type 2 can also resist the influence to some extent. Type 2 offloads components down to ProgHW, so it becomes less dependent on CPU for network-related operations. In addition, we find that memory-intensive applications are more impactful than CPU-intensive applications on SW. Specifically, the former degrades SW performance by 26% while the latter degrades it by 16%.

Group4 - Summary: Type 3 can resist the influence of concurrent applications while DPDK and TCP/IP offloading of type 2 have degraded accuracy in such influence.

Group5 - Impact of Cross Traffic: A real-world network path is usually shared among many network applications which can generate cross traffic to interfere with BWE traffic. In this group, we study how different ProgHW/SW architectures perform with the existence of cross traffic. We use D-ITG 2.8.1 to generate cross traffic with a constant rate, and the packet size is 500 bytes.

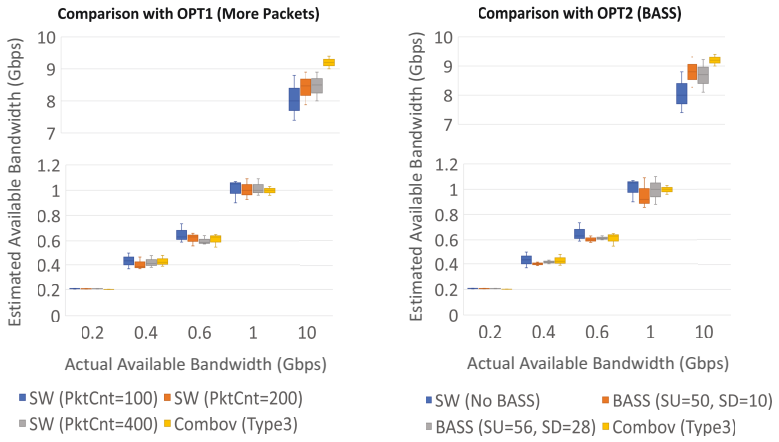


Fig. 12: Comparison with BWE Optimizations

We set the transmission rate of cross traffic to be 100Mbps and 200Mbps. The results are shown in Figure 10d.

We find that the introduction of cross traffic degrades BWE performance for all three types, but type 2 and 3 still show better performance than type 1. Furthermore, according to Figure 10d, we find that BWE performs poorly when the transmission rate of cross traffic is comparable to available bandwidth. For example, if available bandwidth is 200 Mbps, and cross traffic is 200 Mbps, more than 40% estimation error is produced. One possible reason is that the irregular insertion of cross packets into probing packets can interfere with BWE.

Group5 - Summary: Type 2 and 3 show better BWE accuracy than type 1 under the influence of cross traffic.

Group6 - Comparison with BWE Optimizations: In this group, we compare two common optimizations of BWE with the ProgHW-based architecture: Combov. The first optimization is to increase packet count (denoted by $PktCnt$) [23]. The idea is that more samples can reduce measurement variance based on the law of large numbers. The second optimization, named BASS [48], is to smooth measurement spikes. These spikes are outliers in measurement, and they may disturb bandwidth calculation. We use SU to denote the index of spike detection and SD to denote the index of spike confirmation.

According to the results shown in Figure 12, more packets can reduce measurement variance, but they only have a small improvement (around 6%) to the BWE accuracy. For the spikes smoothing optimization: BASS, it shows better BWE accuracy than Combov in networks with less than 1Gbps speed, but it is not as good as Combov if the network speed goes above 1Gbps. Nevertheless, those two optimizations are SW-based, and their deployments are easier than ProgHW deployments.

Group6 - Summary: Compared with the ProgHW-based solution, in low-speed networks (<1Gbps), two SW-level optimizations can achieve similar or

even better BWE accuracy, but they show poorer accuracy in high-speed networks ($\geq$1Gbps).

6 Related Work

6.1 Related Evaluations of Bandwidth Estimation

Researchers and engineers have conducted many evaluations to improve BWE accuracy, and these works have three classes.

The first class evaluates different algorithmic mechanisms and parameters. For example, Strauss et al. [45] evaluate the performance of pathload, IGI, and spruce under 100Mb/s bandwidth. They find that spruce is more accurate than the other two. Yin et al. [48] propose a spike smoothing strategy to improve BWE accuracy on 10Gbps networks. Alok Shriram et al. [42] and Xiliang Liu et al. [34] study BWE performance under different parameter settings such as different measurement timescales and flow sizes. To reduce the architectural bias of BWE, Alok Shriram et al. [41] establish a generic implementation framework. Their experiments focus on how different sampling intensities affect the performance of BWE algorithms. They observe that $50ms$ measurement timescale can significantly improve performance.

The second class focuses on how BWE performs under different network conditions. For example, Aceto et al. [10] propose a unified architecture to tackle inaccuracy issues caused by heterogeneous network environments. Shriram et al. [42] evaluate different BWE methods in both high-speed datacenter and OC-48 networks. Hu et al. [22] investigate network paths with less than 100 Mbps bottleneck links.

The third class analyzes the influence of architectural components of BWE. For example, Jin et al. [25] conclude that major BWE algorithms cannot accurately estimate the high-speed bandwidth (>1Gb/s) because of the limited capabilities of traditional SW systems. Their work also gives a detailed breakdown analysis of system factors such as the interrupt rate, the system timer accuracy, or the PCI bandwidth. Liao et al. [32] and Larsen et al. [28] provide an in-depth discussion of the influence of PCI switching, driver, and DMA engine. They find that the driver and buffer release can produce up to 54% overhead. Kagami et al. [26] propose a passive BWE method for the data plane in Software-Defined Network (SDN). This method improves the BWE accuracy by 10%. Furthermore, a few works leverage ProgHW to improve BWE, such as minProbe [46], Caliper [19], and Combov [20], but most of them only discuss limited design types like traffic synthesizers.

In addition, some other works try to improve the evaluation accuracy and fidelity instead of directly studying BWE algorithms. To mitigate the timing noise of the host system, Strauss et al. [45] collect measurements from multiple probe streams and use OS kernels to improve the timestamp accuracy. To reduce the bias of testing scenarios, Hui Zhou et al. [49] test BWE under more comprehensive internet paths to reveal the difficulties of major BWE algorithms. Kagami et al. [26] offload BWE to the data plane to improve evaluation quality.

6.2 Programmable Hardware Designs

There are some ProgHW/SW designs for traffic generation, but few of them are dedicated to BWE. In terms of the methods of controlling timing, ProgHW/SW designs can be classified into two types: HW-timing type and packet-insertion type. HW-timing type means that the hardware timer is used to either set the IPD of sending packets or record the receiving time of each packet. In contrast with the sub-millisecond timing accuracy of SW design [12], one of the main advantages of this type is that it supports nanosecond-level timing accuracy. For example, Netthread [40] and SPG [16] can accurately replay pre-recorded traffic. However, some designs of this type are not suitable for high-speed BWE. For example, the HW timing module of NIC-based designs [17] can only control the average bit rate, which is unsuitable for the varying-IPD BWE algorithms such as pathchirp [38]. As for the packet-insertion type, the sending IPD is determined by inserting an extra packet between two valid packets. MoonGen [17], SoNIC [29], and minProbe [46] belong to this type, which balances well between the timing accuracy and design flexibility. But the main problem of this type is that the specified IPD may get disturbed by interrupt coalescing. If the extra packets and valid packets appear in different interrupt batches, the pre-defined IPD will not hold anymore.

7 Conclusion

In the paper, we provide an IPD-based modular method to systematically classify and evaluate ProgHW/SW space of BWE. This evaluation method shows higher efficiency than traditional evaluation methods. Furthermore, we make some new findings from the architectural evaluation. According to our experiment results, the IPD HW modulation architecture shows the best improvement in BWE performance. Specifically, it can increase IPD accuracy by 45% and BWE accuracy by 20-30% in a 100Gbps network. We also find that the receiver side affects BWE more than the sender side. In the future, we plan to extend ProgHW/SW space study to more types of BWE algorithms.

Acknowledgement

The work presented in this paper was supported in part by NSF CNS-1616087 and CNS-2135539.

References

1. AXI Reference Guide (Aug 2020), https://www.xilinx.com/support/ documentation/ip_documentation/ug761_axi_reference_guide.pdf
2. Alveo U280 Product Brief (Dec 2021), https://www.xilinx.com/products/boards-and-kits/alveo/u280.html
3. Apache Hadoop (Dec 2021), https://hadoop.apache.org/

4. AWS High Performance Computing (Dec 2021), https://aws.amazon.com/hpc/
5. Intel. Intel DPDK: Data Plane Development Kit (Dec 2021), http://dpdk.org/
6. Lookbusy – a synthetic load generator (Dec 2021), http://www.devin.com/lookbusy/
7. TCP/IP Socket Opencores (Sep 2021), https://opencores.org/projects/tcp_socket
8. Vitis with 100 Gbps TCP/IP Network Stack (Apr 2022), https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP
9. FPGABandwidth (Jan 2023), https://github.com/ftqtfff/FPGABandwidth
10. Aceto, G., Botta, A., Pescapé, A., D'Arienzo, M.: Unified architecture for network measurement: The case of available bandwidth. Journal of Network and Computer Applications **35**(5), 1402–1414 (2012)
11. Aloisio, A., Giordano, R., Izzo, V.: Jitter issues in clock conditioning with fpgas. In: NPSS Real Time Conference. pp. 1–6. IEEE (2010)
12. Botta, A., Dainotti, A., Pescapé, A.: Do you trust your software-based traffic generator? IEEE Communications Magazine **48**(9), 158–165 (2010)
13. Carter, R.L., Crovella, M.E.: Measuring bottleneck link speed in packet-switched networks. Performance evaluation **27**, 297–318 (1996)
14. Chou, P.A., Miao, Z.: Rate-distortion optimized streaming of packetized media. IEEE Transactions on Multimedia **8**(2), 390–404 (2006)
15. Chowdhury, D.D.: Packet Timing: Network Time Protocol. Springer (2021)
16. Covington, G.A., Gibb, G., Lockwood, J.W., McKeown, N.: A packet generator on the NetFPGA platform. In: The 17th IEEE Symposium on Field Programmable Custom Computing Machines. pp. 235–238 (2009)
17. Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., Carle, G.: Moongen: A scriptable high-speed packet generator. In: Proceedings of the 2015 Internet Measurement Conference. pp. 275–287
18. Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al.: Azure accelerated networking: SmartNICs in the public cloud. In: The 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 51–66 (2018)
19. Ghobadi, M., Salmon, G., Ganjali, Y., Labrecque, M., Steffan, J.G.: Caliper: Precise and responsive traffic generator. In: The 20th Annual Symposium on High-Performance Interconnects. pp. 25–32. IEEE (2012)
20. Groleat, T., Arzel, M., Vaton, S., Bourge, A., Le Balch, Y., Bougdal, H., Aranaz Padron, M.: Flexible, extensible, open-source and affordable FPGA-based traffic generator. In: Proceedings of the 1st edition workshop on High performance and programmable networking. pp. 23–30 (2013)
21. Haecki, R., Mysore, R.N., Suresh, L., Zellweger, G., Gan, B., Merrifield, T., Banerjee, S., Roscoe, T.: How to diagnose nanosecond network latencies in rich end-host stacks. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 861–877 (2022)
22. Hu, N., Steenkiste, P.: Evaluation and characterization of available bandwidth probing techniques. IEEE journal on Selected Areas in Communications **21**(6), 879–894 (2003)
23. Jain, M., Dovrolis, C.: End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP Throughput. ACM SIGCOMM Computer Communication Review **32**(4), 295–308 (2002)
24. Jin, G., Tierney, B.: Netest: A tool to measure the maximum burst size, available bandwidth and achievable throughput. In: International Conference on Information Technology: Research and Education (ITRE). pp. 578–582. IEEE (2003)

25. Jin, G., Tierney, B.L.: System capability effects on algorithms for network bandwidth measurement. In: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement. pp. 27–38 (2003)
26. Kagami, N.S., da Costa Filho, R.I.T., Gaspary, L.P.: CAPEST: Offloading Network Capacity and Available Bandwidth Estimation to Programmable Data Planes. IEEE Transactions on Network and Service Management **17**(1), 175–189 (2019)
27. LaCurts, K., Deng, S., Goyal, A., Balakrishnan, H.: Choreo: Network-aware task placement for cloud applications. In: Proceedings of the conference on Internet measurement conference. pp. 191–204 (2013)
28. Larsen, S., Sarangam, P., Huggahalli, R., Kulkarni, S.: Architectural breakdown of end-to-end latency in a TCP/IP network. International journal of parallel programming **37**(6), 556–571 (2009)
29. Lee, K.S., Wang, H., Weatherspoon, H.: Sonic: Precise realtime software access and control of wired networks. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 213–225 (2013)
30. Leeser, M., Handagala, S., Zink, M.: FPGAs in the Cloud. Authorea (Nov 2021). https://doi.org/10.22541/au.163647170.02504770/v1
31. Li, Y., Miao, R., Liu, H.H., Zhuang, Y., Feng, F., Tang, L., Cao, Z., Zhang, M., Kelly, F., Alizadeh, M., Yu, M.: HPCC: High Precision Congestion Control. In: Proceedings of the ACM Special Interest Group on Data Communication. p. 44–58. Association for Computing Machinery (2019)
32. Liao, G., Znu, X., Bnuyan, L.: A new server I/O architecture for high speed networks. In: The 17th International Symposium on High Performance Computer Architecture. pp. 255–265. IEEE (2011)
33. Lin, W., Liang, C., Wang, J.Z., Buyya, R.: Bandwidth-aware divisible task scheduling for cloud computing. Software: Practice and Experience **44**(2), 163–174 (2014)
34. Liu, X., Ravindran, K., Loguinov, D.: Evaluating the potential of bandwidth estimators. In: The 4th New York Metro Area Networking Workshop (NYMAN) (2004)
35. Moreno, V., del Rio, P.M.S., Ramos, J., Garnica, J.J., Garcia-Dorado, J.L.: Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines. IEEE Communications letters **16**(11), 1888–1891 (2012)
36. Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). pp. 13–24
37. Ramos, J., del Río, P.S., Aracil, J., de Vergara, J.L.: On the effect of concurrent applications in bandwidth measurement speedometers. Computer Networks **55**(6), 1435–1453 (2011)
38. Ribeiro, V.J., Riedi, R.H., Baraniuk, R.G., Navratil, J., Cottrell, L.: Pathchirp: Efficient available bandwidth estimation for network paths. In: Passive and active measurement workshop (2003)
39. Ruiz, M., Sidler, D., Sutter, G., Alonso, G., López-Buedo, S.: Limago: An FPGA-based open-source 100 GbE TCP/IP stack. In: International Conference on Field Programmable Logic and Applications (FPL). pp. 286–292. IEEE (2019)
40. Salmon, G., Ghobadi, M., Ganjali, Y., Labrecque, M., Steffan, J.G.: NetFPGA-based precise traffic generation. In: Proceedings of NetFPGA Developers Workshop. vol. 9. Citeseer (2009)
41. Shriram, A., Kaur, J.: Empirical evaluation of techniques for measuring available bandwidth. In: International Conference on Computer Communications (INFOCOM). pp. 2162–2170. IEEE (2007)

42. Shriram, A., Murray, M., Hyun, Y., Brownlee, N., Broido, A., Fomenkov, M., et al.: Comparison of public end-to-end bandwidth estimation tools on high-speed links. In: International Workshop on Passive and Active Network Measurement. pp. 306–320. Springer (2005)
43. Skhiri, R., Fresse, V., Jamont, J.P., Suffran, B., Malek, J.: From FPGA to support cloud to cloud of FPGA: State of the art. International Journal of Reconfigurable Computing **2019** (2019)
44. Sommers, J., Barford, P., Willinger, W.: Laboratory-based calibration of available bandwidth estimation tools. Microprocessors and Microsystems **31**(4), 222–235 (2007)
45. Strauss, J., Katabi, D., Kaashoek, F.: A measurement study of available bandwidth estimation tools. In: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement. pp. 39–44 (2003)
46. Wang, H., Lee, K.S., Li, E., Lim, C.L., Tang, A., Weatherspoon, H.: Timing is everything: Accurate, minimum overhead, available bandwidth estimation in high-speed wired networks. In: Proceedings of the 2014 Conference on Internet Measurement Conference. pp. 407–420
47. Yasukata, K., Honda, M., Santry, D., Eggert, L.: Stackmap: Low-latency networking with the OS stack and dedicated NICs. In: USENIX Annual Technical Conference (USENIX ATC 16). pp. 43–56 (2016)
48. Yin, Q., Kaur, J., Smith, F.D.: Can Bandwidth Estimation Tackle Noise at Ultra-high Speeds? In: IEEE 22nd International Conference on Network Protocols. pp. 107–118 (2014)
49. Zhou, H., Wang, Y., Wang, X., Huai, X.: Difficulties in estimating available bandwidth. In: International Conference on Communications. vol. 2, pp. 704–709. IEEE (2006)
50. Zilberman, N., Audzevich, Y., Covington, G.A., Moore, A.W.: NetFPGA SUME: Toward 100 Gbps as research commodity. IEEE micro **34**(5), 32–41 (2014)