Efficient Verification of Timing-Related Network Functions in High-Speed Hardware

Tianqi Fang, Lisong Xu, Witawas Srisa-an
School of Computing, University of Nebraska-Lincoln
Lincoln, NE 68588-0115, Email: {tfang, xu, witty}@cse.unl.edu

***Abstract*—To achieve a line rate in the high-speed environment of modern networks, there is a continuing effort to offload network functions from software to programmable hardware (HW). Although the offloading effort has led to greater performance, it brings difficulty in the verification of timing-related network functions (Time-NFs) as well. Time-NFs use numerical timing values to perform various network tasks. For example, congestion control algorithm BBR uses round-trip time to improve throughput. Errors in Time-NFs could cause packet loss and poor throughput. However, verifying Time-NFs in HW often involves many clock cycles that can result in an exponentially increasing number of test cases. Current verification methods either do not scale or sacrifice soundness for scalability.**

In this paper, we propose an invariant-based method to improve the verification efficiency without losing soundness. Our method is motivated by an observation that most Time-NFs follow a few fixed patterns to use timing information. Based on these patterns, we develop a set of easy-to-validate invariants to constrain the examination space. According to experiments on real Time-NFs, our method can speed up verification by 7 times on average without losing the verification soundness.

***Index Terms*—Formal Verification, Programmable Hardware, Network Functions**

I. Introduction

There is an ever-rising trend to offload network functions from software (SW) to programmable hardware (HW) to cater for multi-gigabit networks nowadays. The offloading practice has shown higher throughput and lower latency in current studies, such as Tonic [1], NetFPGA [2], and Scalable TCP/IP [3]. Therefore, this new HW-based architecture gradually becomes the backbone of modern clouds, such as Azure SmartNIC [4], and AWS F1 [5].

Although the HW offloading improves the performance of network applications, it brings up the difficulty to verify timing-related network functions (Time-NFs), which are ubiquitous at different layers of network stacks. Time-NFs use numerical timing values to carry out tasks. For example, congestion control algorithm BBR operates on an 8-RTT cycle [6]. TCP uses RTT to set retransmission time [7]. Pathload uses one-way delays (OWDs) to estimate network bandwidth [8]. Errors in Time-NFs could degrade throughput or cause severe packet loss [9], [10]. Therefore, before the deployment of a new Time-NF, engineers want to fully verify its correctness. However, the verification of HW-based Time-NFs is difficult.

The verification difficulty, named the state-explosion issue, is caused by the combination of the clock-driven feature of HW and the long duration of Time-NFs. On the one hand, the variables of a typical HW program are updated at every clock cycle. Therefore, the space of test cases is composed of all possible values at each of the tested clock cycles. Due to this clock-driven feature, more clock cycles will exponentially increase the number of test cases [11], [12], and such verification is indeed an NP-complete problem [13]. On the other hand, HW-based Time-NFs often involve many clock cycles. The reason is that many Time-NFs take from microseconds to milliseconds (e.g., TCP retransmission [7], bandwidth estimation [8]), whereas a high-speed HW runs on a nanosecond-resolution clock [1], [2]. This timing discrepancy produces $10^3 \sim 10^6$ clock cycles to complete a Time-NF and raises verification complexity as a result.

The verification methods of complex HW programs can be classified into black-box testing and white-box testing. The former is simulation-based, which generates different inputs at different clock cycles either in a regular order [14] or in a random order [15]. Although they can quickly examine concrete test cases of interest, they cannot guarantee the soundness (i.e., the absence of bugs) of verification. Because the state-explosion issue produces a lot of possible test cases, it is impractical for them to exhaust them all within a reasonable amount of time. To recover the verification soundness, researchers propose model checking [16], [17], [18], [19], a white-box testing method. Model checking converts the program into a first-order logic formula and its property (i.e., expected behavior of the program) into another formula. The way it verifies the property is to try to seek mathematical relations between the program's formula and the property's formula. Because a mathematical relation usually covers multiple test cases at one time, model checking is an effective complement to simulation-based methods. However, model checking still suffers the state-explosion issue. The main bottleneck is that the size and complexity of formulas grow quickly with an increasing number of clock cycles. Large formulas could make model checking run out of memory.

Because the state-explosion issue produces a large number of scenarios to test, current methods find it difficult to balance verification efficiency and soundness. Specifically, simulation-based methods are fast to check concrete test cases, but they cannot prove the absence of bugs. Model checking can keep the verification soundness, but they do not scale with a large number of clock cycles.

Our work: In this paper, we aim to propose a method that

is more efficient than current methods and keeps verification soundness in the meantime. In our method, we use invariants as constraints and combine them with model checking. Invariants [20], [21], [22], [23], [24] are a set of relations among variables of the program under test. We leverage these invariants to detect and remove invalid test cases in the first place, thereby constraining the examination space of model checking. For example, we plan to check whether a variable $x \geq 1$ within 100 clock cycles, which can be expressed as $x[n] \geq 1, \forall n \in [0, 99]$. If an invariant claims that $x[n] \geq 0$, then the verification time can be saved by not checking invalid test cases in which $x[n] < 0$. We summarize our method in Figure 1. There are two steps: generate an invariant and validate the invariant. If the invariant is valid, then we use it as a constraint.

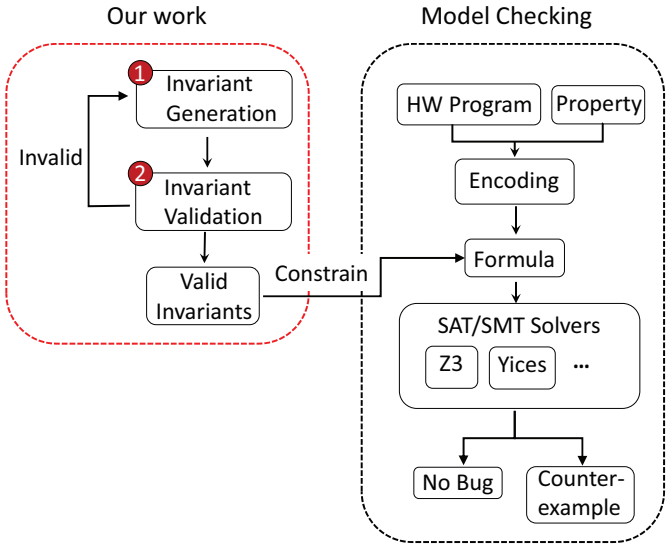

Fig. 1: Overview of Our Method: TINV

However, acquiring a proper invariant to accelerate verification is challenging since invariant validation also takes time. In terms of invariant generation, most current methods tend to produce a large number of candidate invariants but cannot guarantee their correctness, which makes it time-consuming to validate those invariants. The reason is that most methods generate invariants either from polynomial templates, such as Daikon [24] and DIG [22] or from counterexamples, such as FSIS [23] and SWISS [21]. These general templates cover a wide range of invariants. For example, the template for Paxos [21] can produce several millions of candidate invariants for just six terms. Second, because most methods cannot guarantee that invariants are inductive, the validation of invariants may take as many clock cycles as the verification of properties itself. Although there are automatic methods to make invariants inductive [23], [21], they need a long sequence of counterexamples for invariants refinement.

To address those challenges, we propose a method named invariants for Time-NFs (TINV) in this paper. To deal with the challenge of invariant generation, we try to reduce the space of candidate invariants. Specifically, we build invariants by using the domain-specific information of Time-NFs instead of using polynomial templates or counterexamples. Our method is based on an insight that even though there are various Time-NFs, most of them share a few common patterns to process timing information. We identify those implementation patterns and use them as the basis to build invariant templates. Furthermore, to deal with the long-duration challenge of invariants validation, we make our invariant templates inductive so that they can be validated quickly by mathematical induction.

We make the following contributions in this paper.

- We propose a hypothesis that most Time-NFs share three patterns to process timing information. We build three invariant templates based on these patterns, and they are linear-update, bound, and timing-classification templates. In addition, we make our invariant templates inductive so that they can be validated efficiently.
- We present TINV, a methodology using invariants as constraints to efficiently verify HW-based Time-NFs. In the meantime, our method can preserve the soundness of verification.
- We evaluate our hypothesis and our method TINV on a set of 11 real-world Time-NFs of different network layers. First, we find that all 11 Time-NFs satisfy at least one of our templates. Second, TINV is 7 times faster on average than current model checking and invariant-based methods.

The paper is organized as follows: Section II presents the background and challenges of HW-based Time-NFs verification. Section III presents the proposed method TINV in which we introduce our three invariant templates and prove that they are inductive. Section IV analyzes the soundness and efficiency of our method. Section V reports experimental results. Section VI summarizes related work, and Section VII concludes the paper.

II. Background and Verification Challenges

Time-NFs are widely used at different network layers and over a variety of time scales. For example, traffic shaping and rate limiting at network interfaces take microseconds to control transmission rate. TCP retransmission and precision time protocol (PTP) take from milliseconds to seconds to improve transmission efficiency and accuracy. However, because of the high-resolution timing and clock-driven features of HW, verification of Time-NFs is challenging. The main difficulty is that the number of test cases exponentially grows with an increasing number of clock cycles (i.e., the state-explosion issue).

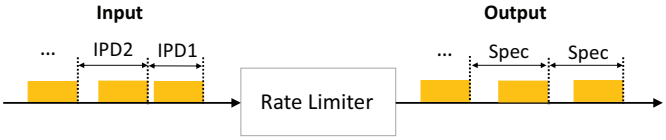

Fig. 2: A Black-box View of Rate Limiter

```
//n-1: previous clock cycle
//n: current clock cycle
always@(posedge clock) begin
    case(state[n-1])
      wait_pkt: begin
        if(pktStart[n-1]==1) begin
          state[n]<=snd_pkt;
          pktTime[n]<=1;
          startCnt[n]<=startCnt[n-1]+1;
        end
      end

      snd_pkt: begin
        if(pktEnd[n-1]==1) begin
          endCnt[n]<=endCnt[n-1]+1;
          if(pktTime[n-1]<Spec-1) begin
             delay[n]<=Spec-2-pktTime[n-1];
             state[n]<=pkt_delay;
          end
          else begin
             state[n]<=wait_pkt;
             if(pktTime[n-1]>Spec-1)
                error[n]<=error[n-1]+1;
          end
        end
        else
          pktTime[n]<=pktTime[n-1]+1;
      end

      pkt_delay: begin
        if(delay[n-1]==0)
          state[n]<=wait_pkt;
        else
          delay[n]<=delay[n-1]-1;
      end
    endcase
end
```

Fig. 3: A White-box View of Rate Limiter (i.e., a source code snippet of rate limiter written in Verilog). There are three states: *wait_pkt* denotes the state of waiting for a new packet, *snd_pkt* denotes the state of sending a packet, and *pkt_delay* denotes the state of setting the interval between the current packet and the next packet. In addition, positive inputs: *pktStart* and *pktEnd* denote the start and end of packet transmission, respectively.

A. Example: Rate Limiter

We use a common Time-NF: rate limiter [25] to detail the verification challenges. A rate limiter is invented to reduce burstiness by specifying the inter-packet delay (IPD) between any two consecutive packets in transmission. Figure 2 presents a black-box view of a rate limiter where the input is a sequence of IPDs of incoming packets, and the output is a sequence of the specified IPDs.

Figure 3 shows the source code or the white-box view of the rate limiter. If packet transmission time measured by $pktTime$ is less than the specified interval $Spec$, then the program will delay a packet for some time before sending it out. If $pktTime$ exceeds $Spec$, then the rate limiter is unable to keep IPD to $Spec$ and increments variable $error$. In addition, variables $startCnt$ ($endCnt$, resp) count the number of packet start (packet end, resp) in transmission.

Consider a case where $N+1$ packets are transmitted, and there are N IPDs. A correct rate limiter should behave in this way if $startCnt$ becomes 1 at the n^s-th clock cycle, then it will become $N+1$ after $N * Spec$ clock cycles. Formally, we express this expected behavior or property as follows (L denotes the total number of clock cycles for which we run the program. In this example, $L = n^s + N * Spec$):

$$P : \textbf{if}(startCnt[n^s] = 1 \wedge error[L] = 0)\{startCnt[L] = N + 1\}$$

B. Concepts and Notations

We define concepts and notations below to make our following discussion precise. A HW program can be expressed as a transition system $S : (\mathbf{X}, INIT, TR)$. $\mathbf{X}$ denotes the set of variables in the program. $INIT$ and TR are first-order logic formulas describing the initial conditions and allowed transitions, respectively. A state of the system under a certain clock cycle is an assignment of values to all variables $\mathbf{X}$. Specifically, an assignment at the n-th clock cycle can be denoted as $\mathbf{x}[n]$. For example, $\mathbf{X}$ and a possible assignment $\mathbf{x}[n]$ can be expressed as follows in the rate limiter program:

$$\mathbf{X} = \{state, pktStart, pktTime, startCnt, pktEnd, delay, endCnt, error\}$$
$$\mathbf{x}[n] = \{state[n] = wait\_pkt, pktStart[n] = 0, pktTime[n] = 1, pktEnd[n] = 0$$
$$startCnt[n] = 0, delay[n] = 0, endCnt[n] = 0, error[n] = 0\}$$

All possible assignments make up an assignment space (denoted as $ASpace$). If the total number of clock cycles in a program execution is L, we have $\forall n \in [0, L], \mathbf{x}[n] \in ASpace$.

A trace τ of the system S is a sequence of assignments $\{\mathbf{x}[0], \mathbf{x}[1], \mathbf{x}[2], \dots\}$. The length of a trace equals $L+1$. The trace τ either satisfies a formula F, denoted $F(\tau)$, or falsifies it, denoted $\neg F(\tau)$. In addition, F only takes the assignments of τ that it needs. For example, $INIT(\mathbf{x}[0])$ means that $\mathbf{x}[0]$ is one of the initial conditions. $TR(\mathbf{x}[n-1], \mathbf{x}[n])$ means that there is a legitimate transition from the assignment $\mathbf{x}[n-1]$ to $\mathbf{x}[n]$ in the system. A trace can be either valid or invalid. A valid trace satisfies that $INIT(\mathbf{x}[0])$ and for each adjacent pair $(\mathbf{x}[n-1], \mathbf{x}[n])$ in the trace, we have $TR(\mathbf{x}[n-1], \mathbf{x}[n])$. All valid and invalid traces make up a trace space $TraceSpace$. Specifically, the trace space can be expressed below:

$$TraceSpace[L] = \underbrace{ASpace \times \dots \times ASpace}_{\text{L+1}}$$

A system formula S_F is a conjunction of $INIT$ and TR. For example, if we run the rate limiter program with a trace $\tau = \{\mathbf{x}[0], \mathbf{x}[1], \dots, \mathbf{x}[L]\}$, the formula can be expressed as follows. If $S_F(\tau)$ is true, then τ is valid.

$$S_F(\tau) : INIT(\mathbf{x}[0]) \wedge TR(\mathbf{x}[0], \mathbf{x}[1]) \wedge \dots \wedge TR(\mathbf{x}[L-1], \mathbf{x}[L])$$

Properties P are the expected behaviors that a system should conform to. To prove that the properties of a system are valid, we need to check if the condition below is satisfied:

$$\forall \tau \in TraceSpace[L],\ S_F(\tau) \rightarrow P(\tau) \qquad (1)$$

If a trace violates the condition above, then the trace is called a counterexample, and the properties are invalid.

If a formula F is inductive, it should satisfy the induction step below. If F is proved valid by mathematical

induction, it should satisfy both conditions below.

$$InitialStep: \forall \mathbf{x}[0] \in ASpace, INIT(\mathbf{x}[0]) \rightarrow F(\mathbf{x}[0])$$
$$InductionStep: \forall n \in [1, L], \forall (\mathbf{x}[n-1] \times \mathbf{x}[n]) \in ASpace \times ASpace,$$
$$F(\mathbf{x}[n-1]) \wedge TR(\mathbf{x}[n-1], \mathbf{x}[n]) \rightarrow F(\mathbf{x}[n])$$

C. Verification Challenges

We apply black-box and white-box verification methods to the rate limiter and explain the verification challenges.

1) Simulation-based Methods: The simulation procedure treats the system under test as a **black box** and stimulates different input signals to check the property. As shown in Figure 2, different input signals are different sequences of IPDs. If there are N incoming IPDs, and each IPD is in the range $[Low, High]$ $(Low < High < Spec)$, the number of possible scenarios to check is $(High - Low)^N$. For example, consider a typical bandwidth estimation scenario where the bandwidth is between 10Mbps and 1Gbps, the default packet size is 100 bytes, and N is 100 [8]. Then the range of each IPD is between $0.8\mu s$ and $80\mu s$. If the HW clock period is $10ns$, $High - Low$ is around 8000 ($(80\mu s - 0.8\mu s)/10ns \approx 8000$). Thus, the number of all possible test cases can be as large as 8000^{100}. However, current studies show that checking 1 test case takes 1 micro-second on average in modern simulators [26]. Therefore, covering all cases of the rate limiter can take years to complete.

2) Model Checking: Model checking treats the system under test as a **white box** and analyzes its source code. This method aims to find mathematical relations or contradictions between the program and its expected behaviors. As shown in Formula (1), model checking aims to find a counterexample among all possible traces. If no such trace exists, then the property P holds. However, it still suffers the state-explosion issue. The size of $TraceSpace$ grows exponentially with the number of clock cycles. Unfortunately, Time-NFs often involve a large number of clock cycles. Follow the bandwidth example above, even if each IPD takes the smallest value: $0.8\mu s$, the total number of clock cycles is 8000 ($100\ IPDs \times 0.8\mu s/10ns = 8000$), which is larger than most hardware verification benchmarks [19].

D. Use Invariants as Constraints

Although model checking has the state-explosion issue, it is generally faster than simulation-based methods due to highly efficient SMT solvers [27], [28], [29], and it can keep verification soundness. Because of those advantages, we aim to improve model checking by applying invariants.

The idea is to use invariants as constraints. Specifically, we generate a set of invariants I from the program under test. These invariants can differentiate between valid traces and invalid traces. Those invalid traces are removed in advance so that verification speed can be improved:

$$\forall \tau \in I \cap TraceSpace[L],\ S_F(\tau) \rightarrow P(\tau)$$

However, it is time-consuming for current methods to generate invariants as constraints. There are two reasons. First, the space of candidate invariants is large, and the candidates' correctness is not guaranteed. Second, most methods cannot guarantee that invariants are inductive, so validating invariants themselves could be as difficult as verifying properties. As an example, we apply a common invariant-based method: FSIS algorithm [23] to the rate limiter program. An invariant of FSIS is a claim that an assignment is unreachable for all valid traces, such as I_1 shown below:

$$I_1:\ \forall n \in [0, L], \neg(delay[n] = Spec \wedge, \ldots, \wedge error[n] = 0)$$

Because any assignment could be set to unreachable, the size of the invariant candidate space is $size(ASpace) = \prod_{v \in \mathbf{X}} |valueRange(v)|$, and $valueRange(v)$ is the set of all possible values that v can get. In addition, I_1 is non-inductive, so it requires as many clock cycles as the verification of properties:

$$\forall \tau \in TraceSpace[L],\ S_F(\tau) \rightarrow I_1(\tau)$$

Therefore, in this paper, we aim to address the difficulties of invariant-based methods and make them more efficient and applicable to the verification of Time-NFs.

III. The Proposed Verification Method

Because current methods are hard to balance verification efficiency and soundness in the verification of Time-NFs, we aim to propose a method named TINV to achieve both, and this section presents the details.

A. Overview

The main idea of our method is to use invariants to constrain the trace space by removing invalid traces so that model checking can take less time to complete. Invariants are a set of relations among variables of programs under test. The traces against those invariants are regarded as invalid traces, and our method can remove them first before verifying properties.

To deal with the difficulties of current invariant-based methods discussed in Section II-D, we make two improvements in our method. First, we explore a smaller invariant space for Time-NFs verification. Specifically, we exploit the domain-specific information of Time-NFs. The key insight in our method is that although there are lots of Time-NFs in different layers of network stacks, most of them share a few implementation patterns to process numerical timing information. We convert those patterns into several invariant templates and make them become a subset of templates generated by current methods such as DIG [22], or FSIS [23], so our space of candidate invariants is smaller. Second, we make our invariant templates inductive under the condition that those implementation patterns are met so that the invariants generated by our templates can be validated quickly by mathematical induction. The definition of an inductive formula is presented in Section II-B. Below, we first define basic notions and then elaborate on our method.

Definition III.1. Timing Variable: Timing variables cover three types of variables. First, a variable counts the number

of occurrences of key timepoints. Second, a variable records a duration. Third, a variable counts the number of occurrences that variables of the first two types meet a certain condition. We use T to denote the set of timing variables of a program.

Example: As shown in Figure 3, $endCnt$ is the first type, $pktTime$ is the second type, and $error$ is the third type.

Definition III.2. Timing-Dependent Variable: A variable whose value can be impacted by a timing variable is called this timing variable's dependent variable. In addition, a timing variable itself is one of its timing-dependent variables.

Example: As shown in Figure 3, $error$ is a timing-dependent variable of $pktTime$.

Our method TINV is presented in Algorithm 1. First, TINV collects all timing-dependent variables of program S and stores them in HashMap T_{map}. The keys of T_{map} are timing variables T provided by a user, and the values of each key are the key's set of timing-dependent variables. Users providing variables is a common step in invariant-based methods. We use $Slice$ function [30] to generate T_{map}. Next, if any variable of properties P can be impacted by any timing variable of T, TINV stores such timing variable in array T_p. Last, TINV uses $InvGen$ function to generate invariants and $InvValid$ function to validate those invariants. If those invariants are correct, then TINV uses them as constraints to accelerate model checking. Otherwise, $InvGen$ function will try another set of timing variables to generate invariants.

Algorithm 1: TINV Function

```
// L:total number of clock cycles, P:property
// S:program under test, T:timing variables
Function TINV(L, S, P, T):
    Tmap ⇐ Slice(S, T)
    T_p ⇐ [ ]
    foreach v ∈ getVar(P) do
        foreach t ∈ Tmap.keys() do
            if v ∈ Tmap[t] ∧ t ∉ T_p then
                T_p.append(t)
    I ⇐ [ ]
    while I == [ ] do
        // step1: invariant generation
        InvGen(T_p, S, I)
        if I == [ ] then
            break
        // step2: invariant validation
        InvValid(L, S, I)
    I_merger ⇐ True
    foreach Inv ∈ I do
        I_merger ⇐ I_merger ∧ Inv
    // apply model checking
    S_F ⇐ get the formula of S for L cycles
    return result ⇐
      ∀τ ∈ TraceSpace[L], S_F(τ) ∧ I_merger(τ) → P(τ)
```

B. Background of Hardware Transitions

This part presents implementation details of TR in HW, which are the basis for our three implementation patterns and invariant templates in the next section.

If $size(\mathbf{X})$ denotes the total number of variables in program S, we can expand TR as a conjunction of each variable's transition as follows ($\forall n \in [1, L], \forall i \in [1, size(\mathbf{X})], v_i \in \mathbf{X}$):

$$TR(\mathbf{x}[n-1], \mathbf{x}[n]) = tr(v_1[n-1], v_1[n]) \wedge \cdots \wedge tr(v_{size(\mathbf{X})}[n-1], v_{size(\mathbf{X})}[n])$$

The implementation of v_i's transition: $tr(v_i[n-1], v_i[n])$ or tr_{v_i} for short is shown below. It is a set of conditional functions (denoted as $func$) within a clock-driven loop [11]. We use $size(tr_{v_i})$ to denote the number of functions in tr_{v_i}. These functions are used to update the value of v_i during the transition once their associated conditions (denoted as $cond$) are met. We assume that each $func$ does not equal others, and we use $cond(func)$ to get the condition of $func$ in our following discussion.

always@($clock\ edge$) *begin*
cond$_1$: $v_i[n] \Leftarrow$ **func$_1$**$(v_1[n-1], \ldots, v_{size(\mathbf{X})}[n-1])$
...
cond$_{\mathbf{size(tr_{v_i})}}$: $v_i[n] \Leftarrow$ **func$_{\mathbf{size(tr_{v_i})}}$**$(v_1[n-1], \ldots, v_{size(\mathbf{X})}[n-1])$
end

tr_{v_i} has two features in a HW program. First, there exists one and only one condition within $\{cond_1, \ldots, cond_{size(tr_{v_i})}\}$ that becomes true at a clock cycle. Second, if there is no change, which is $cond_z : v_i[n] \Leftarrow v_i[n-1] + 0$, we have $cond_z = \bigwedge_{k=1(k\neq z)}^{size(tr_{v_i})} \neg cond_k$.

C. Invariant Generation

This section explains the function for invariant generation $InvGen$. This function is based on our observation that most Time-NFs follow three implementation patterns to process timing information, and we derive three invariant templates from these patterns. We use "$pattern$(template)" to denote the pattern that deducts the template. As shown in Algorithm 2, $InvGen$ leverages three templates (LT, BT, and CT) to generate invariants. Specifically, users select a collection of timing variables from T_p for a certain implementation pattern and put them into the pattern's corresponding template. Then, the template will generate an invariant. In the following discussion, we first show an implementation pattern and then present its associated invariant template. Lastly, we prove that these templates are inductive if their associated patterns exist in program S. For easy reading, the patterns and the templates share symbols, and we omit the $always$ loop sign in tr_{v_i}.

Definition III.3. Linear-Update Pattern ($pattern$(LT)): In this pattern, a timing variable is updated periodically by a linear combination of another set of timing variables. Specifically, for $t_y, t_1, \ldots, t_k \in T$, $\forall n \in [1, L]$, we

Algorithm 2: InvGen Function

```
// tplt stands for template
Function InvGen(T_p, S, I):
    foreach tplt ∈ [LT, BT, CT] do
        // If the subset has been used for tplt
           previously, select another one
        varSet ⇐ Users select a subset of T_p for
          Pattern(tplt) in S
        if varSet ≠ ∅ then
            // replace template variables with
               varSet
            Inv ⇐ tplt(varSet)
            I.append(Inv)
```

express the pattern below ($c_1, \ldots, c_k, b$ are parameters):

$$tr_{t_y} = \begin{cases} \forall i \in [1, size(tr_{t_y})](i \neq y), (b_i \leq 0), \\ cond_i: \ t_y[n] \Leftarrow t_y[n-1] + b_i \\ cond_y: \ t_y[n] \Leftarrow c_1 \cdot t_1[n-1] + \cdots + c_k \cdot t_k[n-1] + b \end{cases}$$

Definition III.4. Linear-Update Invariant Template (LT):

$$t_y[n] \leq max(c_1 \cdot t_1[n-1] + \cdots + c_k \cdot t_k[n-1] + b, t_y[n-1])$$

Example: As shown in Figure 3, $delay$ is updated by $pktTime$ in a negative relation. We put those two variables in the template, and we can get

$$delay[n] \leq max(-pktTime[n-1] + Spec - 2, delay[n-1])$$

Definition III.5. Bound Pattern ($pattern$(BT)): In this pattern, there are two linear combinations of timing variables. And the first one's change is a bound of the second one's change. Specifically, for $t_1, \ldots, t_k$, $t'_1, \ldots, t'_j \in T$, $\forall n \in [1, L]$, we express the pattern below ($c_1, \ldots, c_k, b, c'_1, \ldots, c'_j, b'$ are non-negative parameters):

$$\forall ch_t \in \{func_t - t[n-1] | func_t \in tr_t\}, t \in [t_1, \ldots, t_k]$$
$$\forall ch_{t'} \in \{func_{t'} - t'[n-1] | func_{t'} \in tr_{t'}\}, t' \in [t'_1, \ldots, t'_j]$$
$$If(cond(func_{t_1}) \wedge \cdots \wedge cond(func_{t_k}) \wedge cond(func_{t'_1}) \wedge \cdots \wedge cond(func_{t'_j}) = True):$$
$$c_1 \cdot ch_{t_1} + \cdots + c_k \cdot ch_{t_k} + b \geq c'_1 \cdot ch_{t'_1} + \cdots + c'_j \cdot ch_{t'_j} + b'$$

Definition III.6. Bound Invariant Template (BT): The left-hand combination is an upper bound of the right-hand combination.

$$c_1 \cdot t_1[n] + \cdots + c_k \cdot t_k[n] + b \geq c'_1 \cdot t'_1[n] + \cdots + c'_j \cdot t'_j[n] + b'$$

Regarding parameters instantiation, users can choose either to directly assign values to parameters or to solve a system of equations: $\{c_1 \cdot t_1[n] + \cdots + c_k \cdot t_k[n] + b = c'_1 \cdot t'_1[n] + \cdots + c'_j \cdot t'_j[n] + b'\}$. Specifically, we select $k+j+2$ pairs of clock cycles (denoted as $(n_i - 1, n_i), i \in [1, k+j+2]$) when we run the program. At any of these pairs, at least one timing variable should change its value. We sample the values of timing variables at these $k+j+2$ pairs of clock cycles to instantiate the parameters. In addition, the selection of clock cycles should make a unique solution exist in the system of equations.

Example: As shown in Figure 3, $Spec-2$ is an upper bound of $delay$, so an invariant can be $Spec - 2 \geq delay[n]$.

Definition III.7. Classification Pattern ($pattern$(CT)): In this pattern, the value of a timing variable is sampled periodically, and these samples are classified into different groups. A set of timing variables are used to count the number of samples of each group. Specifically, for $cntY, t_1, \ldots, t_k \in T$, the variable $cntY$ is used to count the total number of samples, and $t_i, (i \in [1, k])$ is used to count the number of samples belonging to the i-th group. $\forall n \in [1, L]$, we express the pattern below:

$$tr_{cntY} = \begin{cases} cond_y: \ cntY[n] \Leftarrow cntY[n-1] + 1 \\ \neg cond_y: \ cntY[n] \Leftarrow cntY[n-1] \end{cases}$$

$$tr_{t_i} = \begin{cases} cond_i: \ t_i[n] \Leftarrow t_i[n-1] + 1 \\ \neg cond_i: \ t_i[n] \Leftarrow t_i[n-1] \end{cases}$$

$$\forall i \in [1, k], \ cond_i \rightarrow cond_y$$

Definition III.8. Classification Invariant Template (CT):

$$\forall i \in [1, k], t_i[n] \leq cntY[n]$$

Example: As shown in Figure 3, $endCnt$ counts the total number of finished transmission, and $error$ counts the number of certain transmission in which transmission time is larger than or equal to $Spec$, so an invariant can be $error[n] \leq endCnt[n]$.

Next, we prove that our three templates are inductive.

Theorem III.1. If $pattern$(LT) exists, LT is inductive.

Proof. First, if $cond_y$ = $True$, then we have $t_y[n] = c_1 \cdot t_1[n-1] + \cdots + c_k \cdot t_k[n-1] + b \leq max(c_1 \cdot t_1[n-1] + \cdots + c_k \cdot t_k[n-1] + b, t_y[n-1])$, so LT is true. Second, if $cond_y$ = $False$, then $\exists i \in [1, size(tr_{t_y})](i \neq y), cond_i = True$. Thus, $t_y[n] = t_y[n-1] + b_i$. Because $t_y[n-1] + b_i \leq t_y[n-1] \leq max(c_1 \cdot t_1[n-1] + \cdots + c_k \cdot t_k[n-1] + b, t_y[n-1])$, LT is true. □

Theorem III.2. If $pattern$(BT) exists, BT is inductive:

Proof. Assume that $c_1 \cdot t_1[n-1] + \cdots + c_k \cdot t_k[n-1] + b \geq c'_1 \cdot t'_1[n-1] + \cdots + c'_j \cdot t'_j[n-1] + b'$. Because $pattern$(BT) exists, the following inequality is true under all valid conditions: $c_1 \cdot (func_{t_1} - t_1[n-1]) + \cdots + c_k \cdot (func_{t_k} - t_k[n-1]) + b \geq c'_1 \cdot (func_{t'_1} - t'_1[n-1]) + \cdots + c'_j \cdot (func_{t'_j} - t'_j[n-1]) + b'$. The parameter-wise sum of these two inequalities is BT, so BT is true. □

Theorem III.3. If $pattern$(CT) exists, CT is inductive:

Proof. Assume that $\forall i \in [1, k], t_i[n-1] \leq cntY[n-1]$. If $cond_x = True$, then $t_i[n] = t_i[n-1] + 1$, and $cntY[n] = cntY[n-1] + 1$, so $t_i[n] \leq cntY[n]$. If $cond_x = False$, then $t_i[n] = t_i[n-1]$, and $cntY[n]$ could be either $cntY[n-1]$ or $cntY[n-1] + 1$. In both cases, $t_i[n] \leq cntY[n]$. □

Algorithm 3: InvValid Function

```
Function InvValid(L, S, I):
    I_return ⇐ [ ]
    foreach Inv ∈ I do
        error ⇐ Check(L, S, Inv)
        if error == 0 then
            I_return.append(Inv)
    I ⇐ I_return
SubFunction Check(L, S, Inv):
    [X, INIT, TR] ⇐ S
    InitialStep ⇐
    ∀x[0] ∈ ASpace, INIT(x[0]) → Inv(x[0])
    InductionStep ⇐
    ∀n ∈ [1, L], ∀(x[n − 1], x[n]) ∈ ASpace × ASpace,
    Inv(x[n − 1]) ∧ TR(x[n − 1], x[n]) → Inv(x[n])
    if InitialStep ∧ InductionStep then
        return 0
    else
        return 1
```

D. Invariant Validation

To efficiently validate the correctness of invariants, we use a method named mathematical induction instead of examining all clock cycles. As shown in Section II-B, a typical HW program has two parts: initial condition ($INIT$) and transition condition (TR). The initial condition describes the program state before the first clock cycle, and the transition condition describes state transitions between two consecutive clock cycles. Valid invariants should satisfy two conditions in mathematical induction: 1. Invariants are correct in the initial condition. 2. If invariants are correct at the previous clock cycle, then they should still be correct at the current clock cycle. These conditions are presented in $Check$ subfunction of Algorithm 3. The advantage of this induction method is that it only needs to examine two conditions instead of traversing all clock cycles, which can avoid the state-explosion issue. However, only inductive invariants can be validated by this method, but many invariants generated by current methods are non-inductive. In contrast, TINV can generate inductive invariants if those three patterns exist in the program.

IV. Analysis of Our Method

In this section, we first prove the soundness of our method, then we compare the efficiency of our method with pure model checking and related invariant-based methods.

A. Soundness Analysis

We prove the theorem below that if our method TINV reports that a property P holds in a system formula S_F, then P actually holds in S_F.

Theorem IV.1. If $\forall \tau \in TraceSpace[L], S_F(\tau) \rightarrow I(\tau)$ and $I(\tau) \wedge S_F(\tau) \rightarrow P(\tau)$, we have $S_F(\tau) \rightarrow P(\tau)$.

Proof. We apply logic conversion below:

$$
\begin{aligned}
&set\ \mathbf{Pre} = (S_F(\tau) \rightarrow I(\tau)) \wedge (I(\tau) \wedge S_F(\tau) \rightarrow P(\tau)) \\
&\because S_F(\tau) \rightarrow I(\tau) = I(\tau) \vee \neg S_F(\tau) \\
&\because I(\tau) \wedge S_F(\tau) \rightarrow P(\tau) = \neg I(\tau) \vee (\neg S_F(\tau) \vee P(\tau)) \\
&\therefore \mathbf{Pre} = (I(\tau) \vee \neg S_F(\tau)) \wedge (\neg I(\tau) \vee (\neg S_F(\tau) \vee P(\tau))) \\
&= (\neg S_F(\tau) \vee P(\tau)) \vee (\neg I(\tau) \wedge \neg S_F(\tau)) \\
&set\ \mathbf{Post} = S_F(\tau) \rightarrow P(\tau) = \neg S_F(\tau) \vee P(\tau) \\
&\textbf{if } I(\tau) = True : \mathbf{Pre} = (\neg S_F(\tau) \vee P(\tau)) \rightarrow \mathbf{Post} \\
&\textbf{else} : \mathbf{Pre} = (\neg S_F(\tau) \vee P(\tau)) \rightarrow \mathbf{Post}
\end{aligned}
$$

□

B. Efficiency Analysis

We first compare our invariant-based method TINV with non-constraint model checking, then we compare TINV with related invariant-based methods.

Theorem IV.2. Using invariants as constraints is more efficient than non-constraint model checking.

Proof. Both methods aim to prove the absence of bugs, which means that they need to examine all traces. Because using invariants can reduce the trace space, it is more efficient. Specifically, if we run a system S for L clock cycles, the trace space of non-constraint model checking is $TraceSpace[L]$. If we use a set of invariants I, the trace space is $I \cap TraceSpace[L] \subseteq TraceSpace[L]$. □

We compare our method TINV with the polynomial-template method [22] and the counterexample-template method [23]. Our method is more efficient from two aspects. First, our invariant space is smaller, which is better in the worst case that all candidate invariants are examined before a valid one is found. Second, our method can make invariants inductive, which makes invariant validation scalable with an increasing number of clock cycles. We present the first aspect here, and the second aspect is detailed in Section III-C.

Definition IV.1. Invariant Space of Polynomial Template: We use Te to denote the set of terms over $\mathbf{X}$ that can appear in polynomials with maximum degree d ($d \geq 1$). Because variables can appear either in the form of the previous clock cycle: $\mathbf{x}[n-1]$ or the form of the current clock cycle: $\mathbf{x}[n]$, the total number of terms in Te, denoted as $size(Te)$, is $\binom{2 \cdot size(\mathbf{X})+d}{d}$ [22]. The expression of the space is shown below:

$$\forall i \in [1, size(Te)], te_i \in Te, c_1 \cdot te_1 + \cdots + c_{size(Te)} \cdot te_{size(Te)} \geq 0$$

Theorem IV.3. The invariant space of TINV is smaller than that of polynomial template.

Proof. TINV reduces the invariant space of polynomial template from three aspects. First, TINV constructs invariants from T, and $T \subseteq \mathbf{X}$. Second, TINV sets the degree $d = 1$. Third, each variable in TINV only appears in the form of one clock cycle. Therefore, the term space of TINV is $\binom{1 \cdot size(\mathbf{T})+1}{1}$, which is smaller than $\binom{2 \cdot size(\mathbf{X})+d}{d}$. □

Definition IV.2. Invariant Space of Counterexample Template: A candidate invariant of the counterexample template is a claim that an assignment is unreachable for all valid traces. Because any assignment could be set to unreachable, the size of the invariant space is $size(ASpace) = \prod_{v \in \mathbf{X}} |valueRange(v)|$. We use $valueRange(v)$ to denote the set of all possible values that can be assigned to the variable v.

Theorem IV.4. The invariant space of TINV is not larger than that of counterexample template.

Proof. TINV only uses T to construct invariants, so the size of the invariant space of TINV is $\prod_{t \in \mathbf{T}} |valueRange(t)|$. Because $T \subseteq \mathbf{X}$, the size of the invariant space of TINV is not larger than $\prod_{v \in \mathbf{X}} |valueRange(v)|$. □

V. Evaluation

In this section, we aim to compare TINV to current verification methods and examine whether real-world Time-NFs match our templates.

A. Experimental Setup

Test Suite. To evaluate TINV, we apply it to a test suite of HW-based Time-NFs from four sources: our implementation of a bandwidth estimation function (bwe) based on Pathload [31], TCP/IP Stack developed by Sidler et al. [3], Limago 100-GbE network stack [25], and a comprehensive collection of multi-gigabit NFs organized by Forencich et al. [32]. The reason to choose these four sources is that they can cover different layers of TCP/IP stack. Specifically, the first three sources cover the application and transport layers, and the last one covers the Ethernet and physical layers. The source code of TINV is also in [31].

Setup. All experiments are conducted on a 16-core machine with Intel Xeon-E5 CPU processors at 3.70GHz, with 64GB of memory, and running Ubuntu 2020.4 LTS with the Linux kernel 5.4. For simulation, we use ModelSim version 21.1.1 [33]. For model checking, we use Yosys [34] to compile programs into SMT formulas, and we use a mainstream solver: Yices version 2.2.1 [28] to process those formulas. We compare TINV with a counterexample-based method: FSIS [23] and a polynomial-based method: DIG [22] in terms of performance. TINV and DIG require a user to provide a group of variables, and FSIS selects all variables by default.

B. Comparison with Prior Work

Table I summarizes the results of running TINV on the test suite and its performance comparison with current verification methods. The invariant-based methods (FSIS, DIG, and TINV) perform verification in three steps: invariant generation (step 1), invariant validation (step 2), and model checking with invariants (step 3). The sum of the time of these three steps is the total verification time of invariant-based methods, and a good invariant-based method should make the sum less than the time of model checking without invariants (baseline). In our experiments, step 1 of these three methods takes less than five seconds, which is negligible compared to the other two, so we omit step 1 and only record the time of the other two steps in the format "step 3|step 2". Furthermore, our important findings are presented below.

Invariants as Constraints. First, using invariants as constraints can effectively reduce the verification complexity. In the experiments, we use the number of boolean propagations ("Propa") of solvers to indicate the verification complexity, since the number has a positive correlation with the number of traces in the verification. The "Propa" entries shown in three invariant-based methods (i.e., FSIS, DIG, and TINV) represent the number of boolean propagations of step 3. By comparing the "Propa" entries between model checking without invariants and the other three invariant-based methods, we find that the latter ones have fewer propagations and less verification time. For bwe as an example, TINV reduces "Propa" by 30 times and makes verification 60 times faster.

Inductive Invariants. Second, one of the main advantages of TINV is that it can generate inductive invariants so that the validation of these invariants is fast. This advantage is useful when TINV's invariants are less efficient than that of other methods. For example, in the experiments of close-timer, although DIG spends less time on step 3 than TINV, DIG needs much longer time to validate its invariant (step 2) because the invariant is not inductive. FSIS has a refinement procedure to make its invariants inductive, but it needs long time to analyze a sequence of counterexamples. Therefore, this advantage makes TINV stand out in the competition of the total time.

Efficiency. Third, although TINV can improve the verification efficiency of model checking, the variance of the improvement is large. For example, TINV makes the verification of bwe 60 times faster compared to model checking without invariants, and TINV only improves the verification efficiency of close-timer by 3 times. The efficiency improvement may depend on many factors, such as the relations between properties and invariants, and the program size. Further research is needed.

C. Distribution of Templates

We check how many programs in the test suite match our templates. The distribution of our three templates among these 11 programs is shown in Figure 4. We find that all programs satisfy at least one of our templates. We also find that BT spreads over Time-NFs of all different layers, LT often appears in Time-NFs that focus on packet transmission, and CT appears in Time-NFs that classify different actions under different conditions. For the details of invariants and properties, please refer to [31].

VI. Related Work

For network verification, there is a rich body of work for verifying stateless and stateful networks. Veriflow [35] and header space analysis [36] verify forwarding behaviors of stateless networks. More recent work such as NetSMC [37] and VMN [38] target the verification of stateful NFs. Because

Source	Program	Sim	Model Check (Baseline)		FSIS [23]		DIG [22]		TINV	
		Time(s)	Time(s)	Propa	Time(s)	Propa	Time(s)	Propa	Time(s)	Propa
[31]	bwe	⟳	7493	$2.88 \cdot 10^{10}$	5692|6358	$2.14 \cdot 10^{10}$	121 | 12	$8.38 \cdot 10^{8}$	125 | <1	$9.82 \cdot 10^{8}$
[3]	iperf-udp	⟳	5965	$2.18 \cdot 10^{10}$	5153|5364	$2.11 \cdot 10^{10}$	2316|5074	$1.12 \cdot 10^{10}$	1307 | <1	$9.2 \cdot 10^{9}$
	retran-timer	⟳	4412	$2.07 \cdot 10^{10}$	4182|4220	$2.04 \cdot 10^{10}$	2106|4212	$1.05 \cdot 10^{10}$	882 | <1	$6.4 \cdot 10^{9}$
	probe-timer	⟳	4097	$2.03 \cdot 10^{10}$	3552|3871	$1.74 \cdot 10^{10}$	2102|3856	$1.05 \cdot 10^{10}$	543 | <1	$3.4 \cdot 10^{9}$
	close-timer	⟳	4932	$2.1 \cdot 10^{10}$	4890|4902	$2.1 \cdot 10^{10}$	1457|4287	$9.33 \cdot 10^{9}$	1644 | <1	$9.7 \cdot 10^{9}$
[25]	rate-limiter	⟳	4837	$2.09 \cdot 10^{10}$	4621|4922	$2.08 \cdot 10^{10}$	3003|4765	$1.61 \cdot 10^{10}$	819 | <1	$6.21 \cdot 10^{9}$
	bw-debug	⟳	2071	$1.04 \cdot 10^{10}$	1784|1921	$9.89 \cdot 10^{9}$	230| <1	$1.57 \cdot 10^{9}$	230| <1	$1.57 \cdot 10^{9}$
[32]	xgmii-tx	⟳	2466	$1.22 \cdot 10^{10}$	2075|2187	$1.04 \cdot 10^{10}$	730|2261	$5.20 \cdot 10^{9}$	817 | <1	$6.21 \cdot 10^{9}$
	rgmii-phy	⟳	4261	$2.05 \cdot 10^{10}$	3919|4066	$1.99 \cdot 10^{10}$	2132|4171	$1.09 \cdot 10^{10}$	1014 | <1	$6.72 \cdot 10^{9}$
	ptp-perout	⟳	⟳	/	⟳ | ⟳	/	4107 | ⟳	$2.03 \cdot 10^{10}$	3832 | <1	$1.89 \cdot 10^{10}$
	ptp-clock	⟳	10422	$8.16 \cdot 10^{10}$	9380|10131	$7.91 \cdot 10^{10}$	2541|9782	$1.27 \cdot 10^{10}$	2176 | <1	$1.09 \cdot 10^{10}$

Note: The verification time of FSIS, DIG, and TINV are expressed in the format 'step 3|step 2' in their time entries, and "Propa" stands for boolean propagation of step 3. The meaning of step 2 and step 3 are explained in Section V-B. FSIS and DIG finish running after finding the first valid invariant, and TINV finishes running after finding a valid invariant generated by any of these three templates: LT, BT, and CT. The symbol ⟳ indicates that the method did not finish in 4 hours. We set the number of clock cycles in model checking to 1000 for all experiments.

TABLE I: Performance Comparison of Different Methods

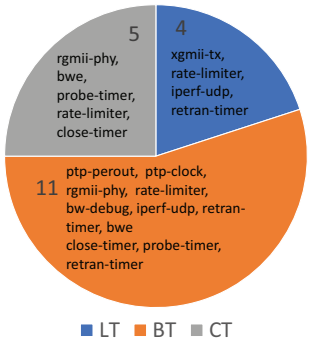

Fig. 4: Distribution of Templates

such verification is undecidable, NetSMC considers only one packet at every state, and VMN abstracts away the order of packets to recover the decidability. In addition, the work [36], [39] target the safety properties of networks, such as reachability, and loop-freedom, and some other work [40], [41] target the liveness properties, such as whitelists of firewalls. However, most recent work focus on the verification of high-level network designs, while our work focuses on the low-level hardware implementation of Time-NFs.

For hardware verification, simulation-based methods [14], [15] can quickly examine concrete cases but cannot keep soundness, so recent efforts advance the development of model checking [16], [17], [18], [19] to fix the unsoundness issue. Besides the work listed in Section I, there are optimization techniques trying to improve model checking. One of the major optimization types is abstraction [16], [42], which converts many low-level states into a few high-level abstract states. However, abstracting numerical values is not proper for Time-NFs, since the verification of Time-NFs involves numerical timing values, and abstraction can easily cause a false alarm. Another optimization technique is to detect variables that stay constant for most of the time and replace them with constant values. This technique can reduce examination space [43], [44]. In addition, the work [11] applies software verification methods on hardware.

For invariant generation, Daikon [24] uses preset mathematical templates to generate invariants, and DIG [22] combines dynamic running traces. One of the advantages of DIG is that it can generate polynomial invariants and array invariants. FSIS [23] uses counterexamples to infer inductive invariants for hardware verification. The work [45] applies interval counterexamples to automatically infer loop invariants. In the field of distributed systems, SWISS [21] automatically searches for inductive invariants to prove safety properties, and it uses counterexamples to speed up invariant searching. Ivy [46] uses partial invariants provided by users as a starting point. I4 [20] analyzes invariants on a small number of instances and generalizes those invariants to an arbitrary number of instances.

VII. Conclusion

This paper describes a method called TINV that uses invariants as constraints to accelerate model checking. TINV leverages three timing templates: LT, BT, and CT to build easy-to-validate invariants. According to our experiments on real-world Time-NFs, our method can accelerate the verification by seven times on average and keep the verification soundness in the meantime. The authors have provided public access to their code and/or data at [31].

Acknowledgement

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

References

[1] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling programmable transport protocols in high-speed NICs," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 93–109.

[2] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014.

[3] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 36–43.

[4] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 51–66.

[5] R. Skhiri, V. Fresse, J. P. Jamont, B. Suffran, and J. Malek, "From FPGA to support cloud to cloud of FPGA: State of the art," *International Journal of Reconfigurable Computing*, 2019.

[6] V. Arun, M. T. Arashloo, A. Saeed, M. Alizadeh, and H. Balakrishnan, "Toward Formally Verifying Congestion Control Behavior," in *Proceedings of ACM SIGCOMM Conference*. Association for Computing Machinery, 2021, p. 1–16.

[7] A. Kesselman and Y. Mansour, "Optimizing TCP Retransmission Timeout," in *Networking - ICN*, P. Lorenz and P. Dini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 133–140.

[8] M. Jain and C. Dovrolis, "End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP Throughput," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4, pp. 295–308, 2002.

[9] P. Ha, E. Zhang, W. Sun, F. Cui, and L. Xu, "A novel timestamping mechanism for clouds and its application on available bandwidth estimation," in *International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 12–22.

[10] H. Wang, K. S. Lee, E. Li, C. L. Lim, A. Tang, and H. Weatherspoon, "Timing is everything: Accurate, minimum overhead, available bandwidth estimation in high-speed wired networks," in *Internet Measurement Conference (IMC)*, 2014, pp. 407–420.

[11] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *Computer Society Annual Symposium on VLSI*. IEEE, 2015, pp. 7–12.

[12] T. Fang, L. Xu, and W. Srisa-an, "Automated Field-based Decomposition to Accelerate Model Checking FPGA-based TCP/IP," in *IEEE International Conference on Communications (ICC)*, 2020, pp. 1–7.

[13] S. Buss and J. Nordström, "Proof complexity and SAT solving," *Handbook of Satisfiability*, vol. 336, pp. 233–350, 2021.

[14] B. Wile, J. Goss, and W. Roesner, *Comprehensive functional verification: The complete industry cycle*. Morgan Kaufmann, 2005.

[15] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, "An empirical comparison of combinatorial testing, random testing and adaptive random testing," *IEEE Transactions on Software Engineering*, vol. 46, no. 3, pp. 302–320, 2018.

[16] A. Goel and K. Sakallah, "Model checking of verilog RTL using IC3 with syntax-guided abstraction," in *NASA Formal Methods Symposium*. Springer, 2019, pp. 166–185.

[17] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking." *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.

[18] Y.-S. Ho, P. Chauhan, P. Roy, A. Mishchenko, and R. Brayton, "Efficient uninterpreted function abstraction and refinement for word-level model checking," in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016, pp. 65–72.

[19] N. Amla, R. Kurshan, K. L. McMillan, and R. Medel, "Experimental analysis of different techniques for bounded model checking," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 34–48.

[20] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, "I4: incremental inference of inductive invariants for verification of distributed protocols," in *ACM Symposium on Operating Systems Principles*, 2019.

[21] T. Hance, M. Heule, R. Martins, and B. Parno, "Finding Invariants of Distributed Systems: It's a Small (Enough) World After All," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021, pp. 115–131.

[22] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "DIG: a dynamic invariant generator for polynomial and array invariants," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–30, 2014.

[23] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2007, pp. 173–180.

[24] M. D. Ernst, "Dynamically detecting likely program invariants," *PhD Dissertation, University of Washington*, 2000.

[25] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo, "Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack," in *29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 286–292.

[26] H. Qian and Y. Deng, "Accelerating RTL simulation with GPUs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011.

[27] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[28] B. Dutertre, "Yices 2.2," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 737–744.

[29] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

[30] J.-C. Ou, "Hardware description language program slicing and way to reduce bounded model checking search overhead," PhD dissertation, 2007.

[31] (2022, December) TimingFunctionVerification. [Online]. Available: https://github.com/ftqtfff/TimingFunctionVerification

[32] (2022, July) Verilog Ethernet Components. [Online]. Available: https://github.com/alexforencich/verilog-ethernet/

[33] (2022, July) ModelSim-Siemens. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/

[34] (2019, September) Yosys Open Synthesis Suite. [Online]. Available: http://www.clifford.at/yosys/

[35] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 15–27.

[36] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 99–111.

[37] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "NetSMC: A custom symbolic model checker for stateful network verification," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 181–200.

[38] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *USENIX symposium on networked systems design and implementation (NSDI)*, 2017, pp. 699–718.

[39] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the Conference of the ACM Special Interest Group on data communication*, 2018, pp. 490–503.

[40] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 59–72.

[41] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani Khaledi, and A. Akella, "Liveness Verification of Stateful Network Functions," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2020, p. 257–272.

[42] Y.-S. H. A. M. Robert and B. N. Een, "Enhancing PDR/IC3 with Localization Abstraction," 2017.

[43] M. L. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced Verification by Temporal Decomposition," in *Formal Methods in Computer-Aided Design*. IEEE, 2009, pp. 17–24.

[44] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich, "rtlv: push-button verification of software on hardware," in *CARRV*, 2021.

[45] R. Xu, F. He, and B.-Y. Wang, "Interval counterexamples for loop invariant learning," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 111–122.

[46] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 614–630.