Formal Verification of Silicon for Software Defined Networking

Saurabh Shrivastava[1], Keqin Han[1], Anh Tran[1],
Chirag Agarwal[2], Ankit Saxena[2], Achin Mittal[2], Anshul Jain[2], Roger Sabbagh[3]
[1]Cavium, San Jose, USA
Oski Technology, [2]Gurgaon, IND, [3]Ottawa, CAN

***Abstract*- The configurability of multi-terabit Software Defined Networking devices ushers in a new level of challenge in the verification process. The number of combinations of configuration settings exceeds the practical limits of traditional verification methods. This paper describes how formal verification methodology was used to address this challenge on the Cavium XPliant® Ethernet Switch designs.**

I. INTRODUCTION

The rising demand for cloud computing applications and services has caused a surge of growth in Software Defined Networking (SDN) protocols. Over recent years, many protocols have emerged to define network virtualization mechanisms such Virtual Local Area Networks (VLAN) [1], Virtual eXtensible Local Area Networks (VXLAN) [2], Network Virtualization using Generic Routing Encapsulation (NVGRE) [3], and more recently, Generic Network Virtualization Encapsulation (GENEVE) [4].

The resulting challenge for design teams developing silicon for physical network devices, such as datacenter switches, is to provide the flexibility to support the wide range of existing protocols and the agility to adapt to new protocols, but without a sacrifice in performance. Traditional switching silicon with fixed architectures need to be re-engineered to support each new protocol, which results in switches that lag years behind the market requirements. In recent years, innovations in silicon design for SDN protocols has led to the development of highly configurable switching silicon that enables programming of every element of switch packet processing [5]. With these devices, a mere software update is all that is required to enable support for a new networking protocol.

The configurability of SDN silicon switches ushers in a new level of challenge in the verification process. The configurable fabric has far too many combinations of settings to practically test with traditional verification methods. The configuration space is large and difficult to bound as it is not possible to anticipate which configuration profiles are going to be used when devices are deployed in the field. Bugs that surface using an unanticipated and untested configuration profile would be catastrophic. Such bugs would render the hardware incapable of supporting an arbitrary new protocol and negate the benefit of configurability offered by these devices.

Formal verification plays a very important role in the verification process of SDN silicon. Formal verification uses mathematical techniques to efficiently explore all possible design configurations. It can provide complete coverage, equivalent to that achieved by simulating all possible profiles, thus leaving no bugs behind while getting to market on time.

II. FORMAL VERIFICATION METHODOLOGY

Formal verification methods suffer from state space complexity barriers, which are associated with exponentially hard to solve PSPACE-complete problems [5]. As a result, formal technology

must be coupled with the right methodology that enables the promised benefit of exhaustive analysis to be realized [6]. At a high level, our formal methodology consisted of the following primary components, known as the "4 Cs":

- **End-to-end Checkers.** End-to-end checkers model the end-to-end behavior of the block under test. They must be coded using special techniques that make them "formal friendly" so that they add only the minimal amount of complexity overhead.
- **Constraints.** Constraints are required to filter out illegal input space combinations, however they must be verified to ensure bugs are not missed and the design-under-test is not over-constrained.
- **Complexity Management.** Abstraction models reduce the state space depth so that formal verification can explore beyond the default complexity barriers.
- **Coverage.** Formal coverage measures both the range of stimulus that has been explored as well as the quality of checking performed by the testbench, making it a very strong sign-off metric.

Formal verification was a full-fledged part of our verification strategy, which evidenced itself in three primary ways:

1. Formal verification was incorporated into our planning process from the very beginning. As simulation verifiers developed their test plans, they worked together with the formal verifiers to divide the feature list between them. This ensured that the most efficient verification solution was applied in each case and that duplication of effort was avoided as much as possible.
2. This cooperation extended to the sign-off phase of the project as well, where the coverage achieved by both formal and simulation were considered together as part of the sign-off criteria.
3. We also integrated formal verification into our regression framework to automate checking of any RTL changes and so that deviations from the golden set of reviewed results would quickly be detected.

III. CASE STUDY: CAVIUM XPLIANT® ETHERNET SWITCH

The above mentioned formal methodology was used for the Cavium XPliant Ethernet Switch CNX880xx product family [7]. Formal verification was heavily relied upon to verify the configurability of the XPliant Packet Architecture (XPA™) control plane. The block diagram of the CNX880xx devices is shown in Fig. 1. Many of the control plane blocks were verified either partially or wholly with formal verification. This included a wide variety of designs such as traffic shapers, the programmable fabric, crossbars, packet modifiers and arbiters. The XPA Pipeline is one of the most important blocks in this architecture. It is highly configurable and has many replicated elements to provide the necessary bandwidth and throughput. This was one specific block for which formal verification was critical in achieving the coverage level required for sign-off.

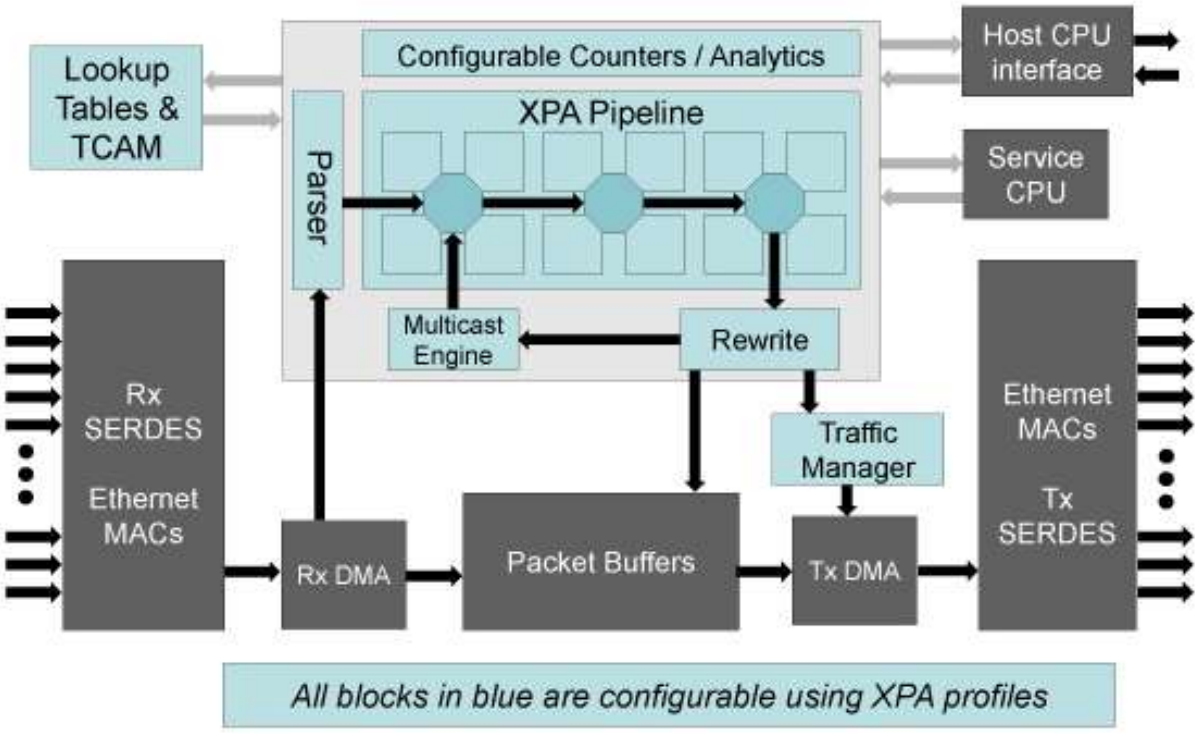

Fig. 1. CNX880xx Product Family Block Diagram

The following are examples of the designs that were formally verified:

A. Linked-List Pointer Conflict Resolution Unit

The Linked-List Pointer Conflict Resolution Unit of Fig. 2 is made up of tail pointer staging FIFOs, the tail pointer cache and tail pointer de-conflict logic. Tail pointers are allocated from the free list memory as per the incoming traffic and are sent to the tail pointer staging FIFOs. To meet the throughput needs of the device, tail pointers are broken into four banks and are allocated from each individual bank. Due to this bank architecture, there can be tail pointer bank collisions when tail pointers are recovered, as recovered pointers need to be written to the correct bank and the bank distribution of arriving recovered pointers is random based on traffic patterns. Only up to four non-colliding tail pointers can be written to memory. This can cause delays in updating the linked-list memory. To avoid flow control on the dequeue logic, a cache is maintained to hold in-flight tail pointers. For all head update requests, the cache is checked against the previous head pointer to find the current head pointer. If there is a hit, the head pointer value is provided with match indication otherwise the current head pointer will be picked from the linked-list memory.

The block cache had to be sized based on the worst case latency of the design. This block had over 14,000 legal configurations. Determining the worst-case latency is not easy in simulation because of the number of configuration combinations and the state-space complexity of the block. Formal can directly target the worst-case condition and find the combination of design configuration settings, states and events that cause the longest round-trip delay.

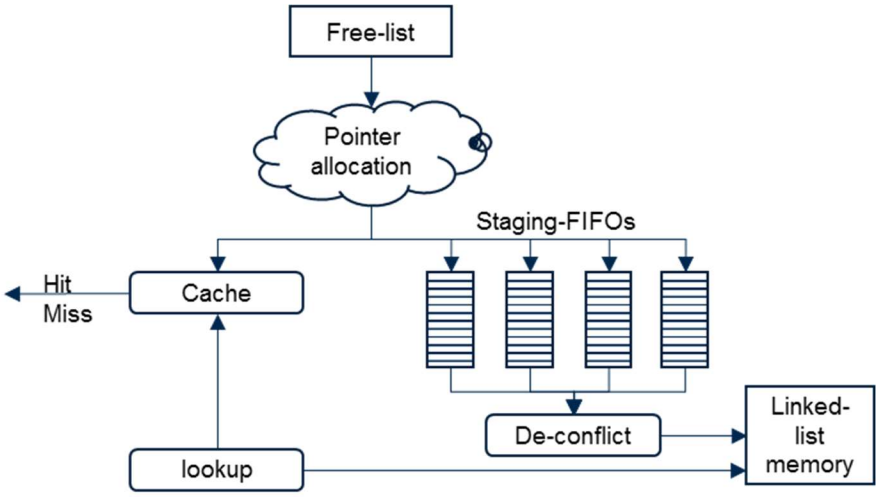

Fig. 2. Linked-List Pointer Conflict Resolution Unit

Along similar lines, the block contains logic responsible for ensuring that all used pointers go back to the free list. Formal techniques were also applied to check for bank conflicts while writing back to the free-list.

A.1 Formal Environment

Let us consider some of the key elements of the formal verification environment which was developed for the Linked-List Pointer Conflict Resolution Unit.

Checkers

1. Correctness of Head Pointer
    - While de-queueing from the linked-list, the pointers shall be traversed in the same order as they were received
2. Dropping of Pointer Data
    - When a given entry is evicted from the cache, it shall have been previously written to the linked-list memory
3. Starvation
    - If a given staging FIFO is non-empty, it shall always get an opportunity to write to linked-list memory within a known number of cycles, even if there are bank collisions
4. Bank De-conflict
    - The staging FIFOs shall not have multiple, simultaneous writes to the same memory bank

Constraints

1. Unique Pointer
    - Once allocated, a given pointer shall not be allocated again until it is freed
2. FIFO Flow-control
    - There shall be no FIFO writes when backpressure is asserted

Complexity Management

1. Reset Abstraction
    - Allow pointers to be non-deterministically allocated out of reset
2. Exploiting Symmetry
    - Use symbolic constants to non-deterministically select and track a single pointer in a single linked-list

A.2 Results

Even though simulation of the design was at a late stage and the cache seemed to be sized appropriately, through formal analysis, we found that the cache was in fact undersized in some configurations, which would have caused critical data to be dropped.

In the counter example waveform of Fig. 3, the cache had an entry flushed out (FF → 0) before it got written to linked-list memory. When the new pointer request came for pointer FF, a cache miss occurred and because the entry was never written to linked-list memory, a stale pointer was used as a next pointer. This caused corruption of the linked-list. This scenario requires a specific configuration and input pattern to create the sequence where the cache gets 32 pushes in just 10 cycles. After discussion with the system level team, it was concluded that this input scenario is perfectly legal, and the designer increased the size of the cache. In multiple iterations of such bugs, the cache size was finally increased to 96 from the original size of 32.

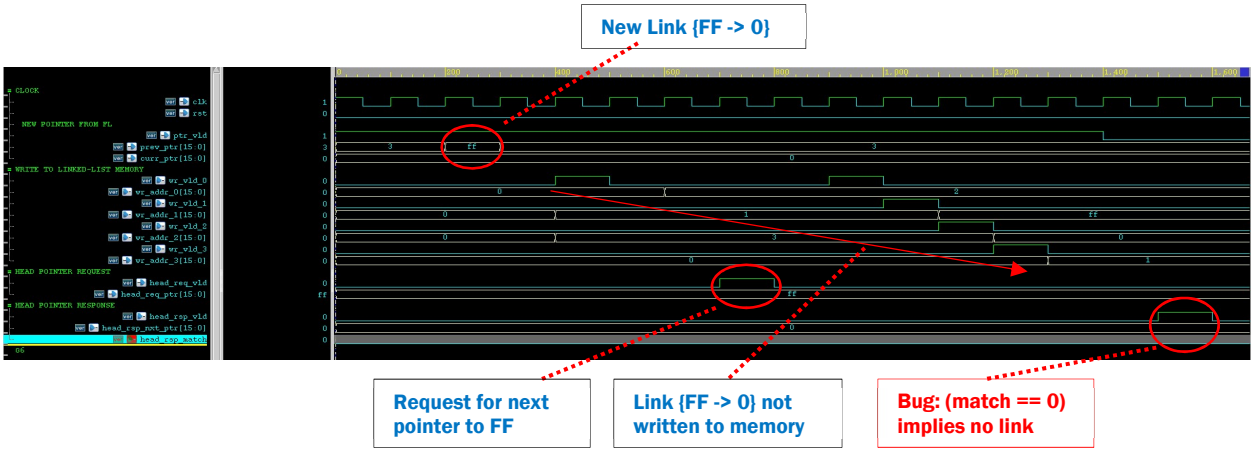

Fig. 3. Corner Case Bug Waveform For Insufficient Cache Size

B. Data Merge Block

The Data Merge Block in Fig.4 merges packet data with intermediate data coming from another block on four separate lanes. The data from each lane belongs to one of four categories. The data is then stored in one of {lane, category} lookup tables. When all four lanes are available for all four categories based on lane-mask and category-mask, then the {lane, category} data is merged with the packet data waiting in a FIFO.

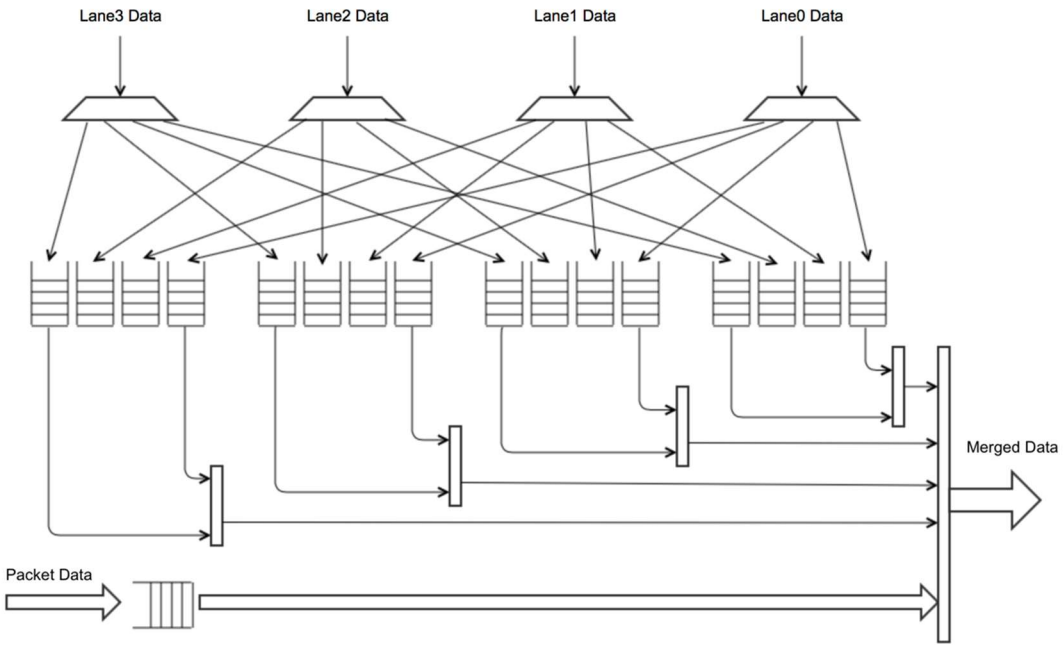

Fig. 4. Data Merge Block

Data in this block can arrive out-of-order, on different lanes and at separate times. In addition, this block is highly configurable, with over 2^{500} legal configurations. The number of conditions that must be tested for all possible lane scenarios in all possible configurations would be impractical to cover in simulations. With formal we could focus on this logic and exhaustively explore all combinations and uncover all the corner-case bugs.

B.1 Formal Environment

Some key elements of the formal verification environment which was developed for the Data Merge Block are:

Checkers

1. Data Consistency
    - Packet data shall not be modified within the merger block
2. Packet Transmit Conditions
    - Packets shall be sent only when all required data lanes are available or a timeout has occurred
3. Forward Progress
    - Once the data lanes are available and the packet is at the head of the FIFO, then it shall be transmitted within a known number of cycles
4. Bandwidth Utilization
    - With no back-pressure and with packets ready to send, there shall be no residual bubble at the output interface

Constraints

1. Unique IDs
    - Incoming back-to-back packets shall not have the same packet IDs
2. Packet ID Repetition
    - Once the packet ID is received by the data merge block, it shall not be received again before the merge is done for the corresponding packet
3. Unique Response Data
    - Once the response data for a packet ID has been received on all required lanes, response for that packet ID shall not be received again before the previous response has been consumed

Complexity Management

1. Reset Abstraction
    - The content of lookup-tables was non-deterministic at reset, which allows response data to be arbitrarily available out of reset
2. Exploiting Symmetry
    - Use symbolic constants to non-deterministically select and track a single packet ID
3. Data Coloring
    - Data consistency is checked with the Wolper method [8], which doesn't require any data to be stored in the checker logic

B.2 Results

Formal reduced the verification cycle by completely replacing block level simulation.

Multiple corner case bugs were found during the course of formal verification. These bugs were related to different configurations for lane masks. Testing huge combinations of out-of-order data arriving across 4 different lanes, where not all the lanes are always required, is impractical in simulation. Examples of the types of bugs found were:

- A packet was incorrectly waiting for lane data
- A packet merging date from the wrong lane
- The output indicating false merge done
- A packet generating false timeout

C. Clos Network

A Clos network [9] provides non-blocking connectivity in network switching applications. A parameterized Clos network was implemented for providing connectivity to NxN ports, as shown in Fig. 5. This network had $2\log_2 N-1$ stages, where each stage had N/2 2x2 crossbar switches. The total cost of such network was $M*N*(2\log_2 N-1)$, where M was datapath width for each port. Formal was extremely efficient in verifying every combination of this network before it was deployed as an interconnect across the switch.

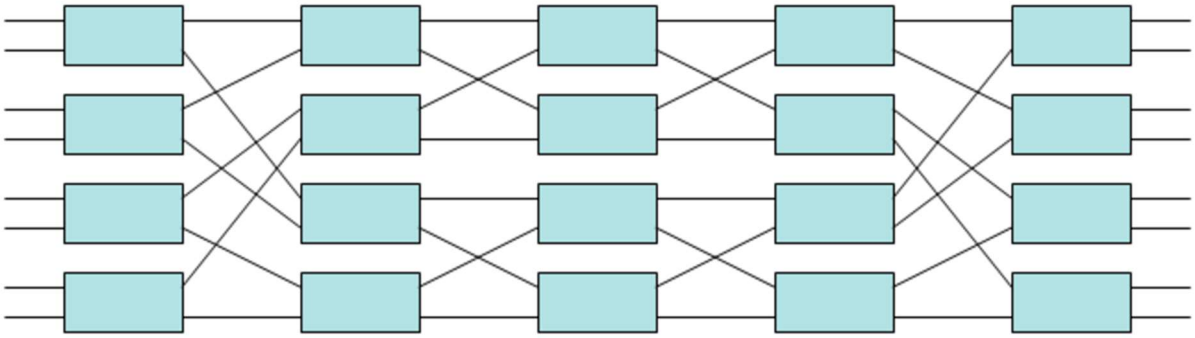

Fig. 5. Clos Network

C.1 Formal Environment

The key aspects of the formal environment for the Clos network were as follows:

Checkers

1. Data Integrity
    - Output data shall match expected data
2. Bandwidth Utilization
    - When there is no back-pressure, then data shall be sent without any bubbles
3. Forward Progress
    - When there is no contention, a transaction destined to a given port shall come out in a fixed number of cycles

Complexity Management

1. Exploiting Symmetry
    - Only data from one port is tracked; selected port ID is constant, but otherwise unconstrained
    - A single bit of data is tracked; selected bit ID is constant, but otherwise unconstrained

C.2 Results

This network block was used in multiple places across the switch, hence the bugs found during formal verification had a greater impact. There were multiple bugs found related to data integrity and performance.

IV. RESULTS

We found over 100 bugs in 40 design blocks and we achieved a level of coverage that wouldn't have been possible any other way. A few of corner case bugs would have been very difficult to find in unit or cluster simulations. A subset of bugs would have been ultimately found in simulations, albeit at a higher debug effort and schedule cost. Some of the select corner-case bugs that were found are listed in section III along with the description of three design blocks.

In each case, we used formal coverage metrics as part of the sign-off criteria. Formal coverage helped to grade the other three aspects of the methodology: checkers, constraints and complexity management. Reachability coverage helped to identify over-constraints and to set the

minimum bounded-proof depths. Formal observability coverage reported any design flops that were not covered by the checkers and helped to ensure checker completeness.

We subsequently simulated all the formally verified blocks at higher levels of design hierarchy and used in-circuit emulation to enable software integration and co-verification at the full-chip level. During these subsequent verification steps, we did not find any bugs in the areas that were targeted with formal verification.

V. CONCLUSIONS

Formal verification is a key enabler for the efficient verification of SDN silicon designs. It can explore all possible design configurations to ensure that no bugs are left behind which could otherwise surface in the field when new networking protocols are deployed.

Formal verification methodology has evolved to the point where it is practical to apply this technology to many types of RTL design blocks. The formal flow is now an essential component of the silicon development sign-off process for the XPliant ethernet switches.

ACKNOWLEDGMENT

The authors are grateful for the support of Frank Wang and Guy Hutchison.

REFERENCES

[1] IEEE 802.1Q, http://standards.ieee.org/about/get/802/802.1.html

[2] Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, https://tools.ietf.org/html/rfc7348

[3] NVGRE: Network Virtualization Using Generic Routing Encapsulation, https://tools.ietf.org/html/rfc7637

[4] Geneve: Generic Network Virtualization Encapsulation, https://tools.ietf.org/html/draft-gross-geneve-00

[5] A. Aziz, V. Singhal, and R. Brayton. “Verifying Interacting Finite State Machines: Complexity Issues.” Technical Report UCB/ERL M93/52, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1993.

[6] I. Tripathi, A. Saxena, A. Verma, P. Aggarwal. "The Process and Proof for Formal Sign-off: A Live Case Study.", DVCon 2016.

[7] XPliant® Ethernet Switch Product Family, http://cavium.com/XPliant-Ethernet-Switch-Product-Family.html

[8] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In Proc. POPL ’86, pp. 184–193, 1986.

[9] Clos Network: https://en.wikipedia.org/wiki/Clos_network