Verifying A Gigabit Ethernet Switch Using SMV

Yuan Lu Mike Jorda
Enterprise Switching Division
Broadcom Corporation
{ylu,mjorda}@broadcom.com

Abstract

We use model checking techniques to verify a switching block in a new Gigabit Ethernet switch – BCM5690. Due to its dynamic nature, this block has been traditionally difficult to verify. Formal techniques are far more efficient than simulation for this particular design. Among 26 design errors discovered, 22 are found using formal methods. We then improve our model checking capability to analyze switch latency. We also use induction to avoid state explosion in the model checker.

Categories & Descriptors: B.7.2

General Terms: Verification

1 Introduction

System-On-Chip (SOC) design is becoming considerably more complicated. Traditional simulation based validation methodologies often fail to discover corner-case design errors. In this paper, we discuss how to use a symbolic model checker, Cadence SMV [CBL98], to validate the Ethernet switching logic block in a new Gigabit Ethernet networking chip, the BCM5690. This block traditionally has been difficult to verify due to its dynamic nature and great concurrency. In fact, we have seen a number of bugs missed by simulation approaches on earlier implementations.

The main contribution of this paper is to show how to apply appropriate formal techniques to industrial problems. We show the ideas behind our verification decisions and subsequent abstraction techniques. Our first and most important decision is that the primary goal is to search the design for errors instead of completely verifying it. We will justify this decision and its consequences in later sections. We will also discuss our definition of "high-quality" bugs and analyze our results with respect to that definition. In addition, we also extend the result obtained from formal analysis to identify correct performance margin. We believe that this performance analysis cannot be achieved by either formal or simulation approaches independently. Building on classic model checking, we then introduce a novel induction to avoid state explosion as the environment model becomes sophisticated. Among the total of 26 design errors uncovered, 22 were found by formal methods. As a result, the chip has been shipping for two years with no bugs in this block. We believe that our experience is applicable to many other scenarios.

Although formal verification has not yet been widely adopted by industry, a number of successful applications have been accomplished. A number of commonly used design structures such as pipelines, bus arbiters and Tomasulo's algorithm have been verified using formal techniques [BD94, M98, CCLW99]. Recently, engineers have started to apply formal techniques on a wider range of designs [A00, B01]. Specifically, Bentley [B01] shows that the Intel team found over 100 "high-quality" logic bugs in the Pentium IV. However, because networking chips lack exact specifications, this design presents a fundamentally different verification problem compared to other regular design structures. As a result, our verification effort is significantly different from previous work.

This paper is organized as follows. In Section 2, we describe the functionality and micro-architecture of the design. We discuss specific features and intuitive observations that affect our verification decisions. Section 3 sketches our basic abstraction techniques while Section 4 outlines our experimental results by introducing two metrics to analyze bug quality. We extend our model checking capability in Section 5. Finally, Section 6 draws conclusions based on our experience.

2 Design Under Verification

The BCM5690 is a single chip with 12 Gigabit Ethernet ports, one 10-Gigabit high-speed interconnect and a CPU interface. For each packet, Ethernet switching (Layer 2, or L2), IP switching (Layer 3, or L3) and higher level switching occurs in a special block called the Address Resolution Logic (ARL) by mapping addresses to physical ports in tables, both L2 and L3, stored in memory. The table data can be statically or dynamically added, deleted, or updated. This paper focuses on the L2 table.

For each packet, there are at most three possible actions that the ARL performs: source address lookup, destination address lookup, and dynamic learning. If the chip is

DAC 2004, June 7–11, 2004, San Diego, California, USA

running at 125MHz (default setting), the L2 table logic must finish these actions within 84 cycles for each Gigabit port and within 8 cycles for the 10-Gigabit port [KKC98]. The L2 table is structured as a hash table (Figure 1). The 16K logical entries are mapped to 2K buckets, with 8 entries per bucket, using a hash function. There is no ordering between the buckets. Similarly, within each bucket, there is no ordering among the entries. Dynamic modifications are decoupled read-modify-write operations which require coherency. Aging (dynamic deletion) has lower priority than lookups. Hence, the design must account for starvation cases.

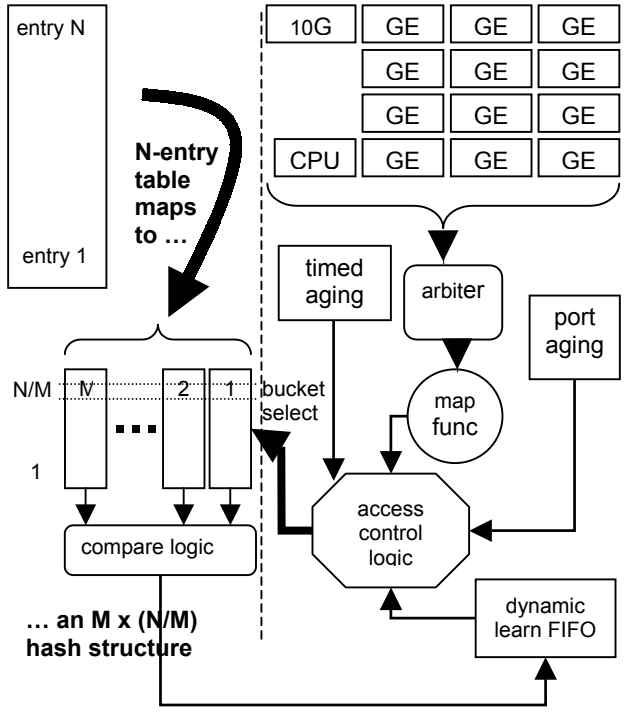

Figure 1 Micro-architecture for L2 logic

The L2 table logic is traditionally difficult to verify due to its dynamic nature. For example, learns can be cancelled for many reasons. Without knowing the exact state of the logic, it is almost impossible to predict when learns happen. In Section 4, we will further discuss this observation. Second, performance issues are difficult to identify in simulation. Multiple operations associated with a single packet can be decoupled over many cycles. Erroneous behavior may thus result in an observable effect thousands of cycles after the fact, or not at all, depending on the test or application. Third, writing a checker for this block is not appropriate. The checker can be easily written similarly to the RTL. It is then difficult to guarantee its correctness.

In this work, we decide to apply formal techniques. Since it is difficult to come up with a complete specification, we choose not to fully verify this block. Instead, we focus on finding as many bugs as possible. Our decision has been justified based on its effectiveness. The chip has been shipping for two years with no bugs in this block.

3 Model Reduction

In order to discover design errors efficiently, we have made two verification decisions. First, we write as many properties as possible because there is no complete specification available. Second, we apply abstraction to reduce the design size. The L2 table logic includes around 3000 registers and a 16K-entry (1Mbit) lookup table. To our knowledge, state-of-the-art model checkers cannot verify such large-scale designs with complicated properties. Therefore, we apply aggressive abstraction even when some features are not completely verified. In Section 5, we extend our effort to verify some uncovered features.

Assume that a Kripke structure M [CGP99] models the L2 table logic. Also let $M \downarrow t$ be a short-hand notation for M where a function t is disabled. For example, $M \downarrow aging$ represents the reduced L2 table model where aging is disabled. Then we find that the buckets in the L2 table are fully symmetric on $M \downarrow aging$. Instead of modeling all 2048 buckets, we only need to model two buckets. With aging enabled, we still use the reduced L2 table because the original table is too large. This may introduce both false positive and false negative behaviors into the abstracted model. With careful debugging, we have been able to avoid unwanted behaviors. Similar symmetry exists within buckets, i.e., any two entries within a bucket are symmetric. Instead of modeling eight entries per bucket, we only model two entries.

Besides reducing the table size, we also apply two abstractions to further reduce the complexity. First, the MAC addresses are represented by two bits instead of 48 bits. This technique is widely used in formal verification. Second, the behaviors of the 12 Gigabit ports are independent, though they are not symmetric. Let $M \downarrow port^i$ denote the L2 table model without sending requests from port i, if we disprove a universal property on the reduced model $M \downarrow port^i$, that property will be false in the original model M. Hence, we only keep at most two Gigabit ports instead of twelve. Note that the 10-Gigabit port and the CPU interface are modeled accurately. By applying these abstractions, we reduce the size of the L2 table model to fewer than 200 registers (Table 1).

	OriginalModel	AbstractModel
#registers	2878	< 200
#buckets	2048	2
#entries/bucket	8	2
#bits/mac addr	48	2
#Gig ports	12	2

Table 1 Abstraction effect on model size

In addition to abstractions, we also simplify the environment model. A request from a Gigabit port to the

L2 table arrives at least every 84 cycles. It is difficult for SMV to complete a fix-point computation if we model that exactly. We also observe that a single request from multiple ports is more interesting than multiple requests from a single port. Therefore, we only model one request per port, significantly reducing the computation time. Most bugs are found with this simplified environment. To improve coverage, we show how to debug the scenario with multiple requests in Section 5.2.

These abstractions are accomplished by recoding the RTL using ePerl [OSSP98]. The rewritten code is configurable. For example, if we want to verify the logic with one Gigabit port, one 10-Gigabit port, and aging, ePerl will generate the model and its related properties automatically.

4 Design debugging

We have written over 200 properties in ePerl. For different configurations, the number of properties varies from 150 to 300. The runtime for a property ranges from 15 seconds to 2 hours. We have verified 35 configurations. It took four man months to complete the formal verification process. Parallel with this work, a traditional simulator is developed at the ARL level. The simulator has found 4 bugs in the L2 table logic while SMV has found 22 bugs. One reasonable explanation of the difference is that the simulator not only needs to *stimulate* the error behavior in the L2 table logic, it has to make them *visible* at the ARL interface. However, a simple comparison of these two numbers is meaningless because the simulator would find more bugs if our formal verification were not performed. In order to understand the quality of these bugs, we introduce two metrics, **error stimulus** and **error visibility**.

The first metric is the error stimulus function, or ES. This is simply the probability of the input stimuli required to discover the bug starting with the initial states. For example, assume that a valid MAC address *A* is stored in a particular bucket of the L2 table. ES for a lookup of MAC address *B* with *h(B)=h(A)* is 1/2048. In computing ES, we only consider the best possible known case to stimulate the corresponding bug. This provides a conservative approximation of ES.

The error visibility, or EV, of a bug is a function of time and input stimuli. EV_{time} is the number of cycles required for erroneous behavior to propagate to the test bench. EV_{stim} measures the probability of the input stimuli required to propagate the bug from cause to externally visible symptom. Then $EV = EV_{stim}/EV_{time}$ provides a means of comparing the relative visibility of bugs. Note that EV_{stim} is calculated similarly as ES, based on a conservative approximation. As an example, let us assume that the minimum number of cycles to propagate a bug to the ARL interface is 84 cycles. Also assume that the required stimuli include a sequence of events:

- Two consecutive 10Gigabit port lookup requests on bucket *i*,
- A specific Gigabit port lookup request also on bucket *i*,

then

$EV_{stim} = (1/13)^2 * 1/2048 * 1/13 * 1/2048 = 2.5e\text{-}7$

and

$EV = EV_{stim}/EV_{time} = 3.0e\text{-}9$.

Note that EV is evaluated conservatively because EV_{time} is approximated using the shortest trace and EV_{stim} is the upper-bound probability for stimuli required to propagate the bug.

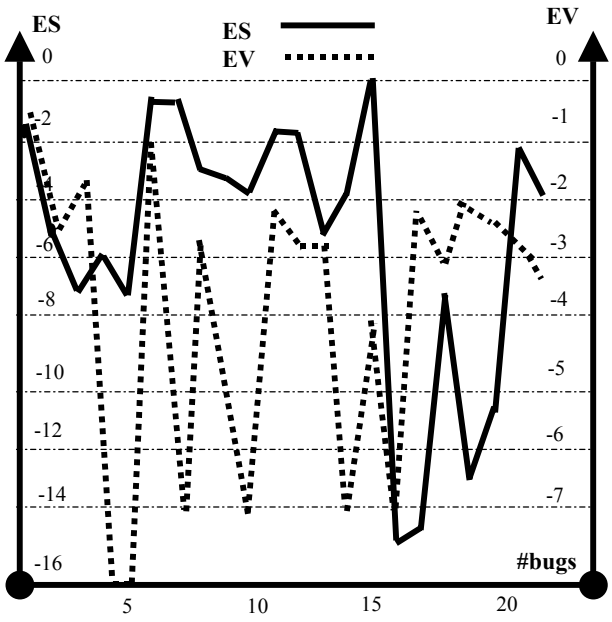

Figure 2 ES/EV for every bug found by SMV

The error stimulus ES and error visibility EV are reported in Figures 2. The X axis corresponds to the cumulative number of bugs. The Y axis corresponds to ES and EV separately. Note that the Y axis is in logarithm scale (-6 means that 10^{-6}). The lower the value, the less likely that the bug is found by the simulator. For example, it is much more difficult to stimulate bug No.16 than bug No.15. We believe that the simulator can easily catch bug No.15. In contrast, without formal approaches, it is extremely difficult for random simulation to catch bug No.16. We do not compute the EVs for bug No.4 and No.5 because we do not know how to propagate them to the ARL interface directly. However, they degrade the system performance. Accumulated inefficiencies eventually lead to a visible error. The required number of simulation cycles can be prohibitively large.

5 Improving Model Checking

5.1 Rigorous performance analysis

Due to aggressive abstraction, certain important performance issues are not addressed. For example, all Gigabit port lookup requests and learns must be served within 84 cycles. The initial analysis shows that the design

has 47% and 21% margin for lookups and learns respectively.

Given our abstracted model, we are not able to verify these performance requirements. However, in an abstracted configuration, SMV finds that a learn is served far later than expected when the logic transitions from the aging mode to the normal switching mode (Figure 3). If a lookup request comes just before that transition, it is possible that its learn happens much later. By manual construction of a similar scenario for all ports, we find that the performance requirement for learns is only marginally satisfied. The margin for learns reduces to 3% from 21%. This raises potential problems due to the assumptions made in the design. It is extremely difficult for simulation to detect this trace because it requires that over twenty different behaviors occur in a specific order and time. Starting with an SMV trace, we construct valuable analysis that cannot be accomplished by either simulation or formal verification independently.

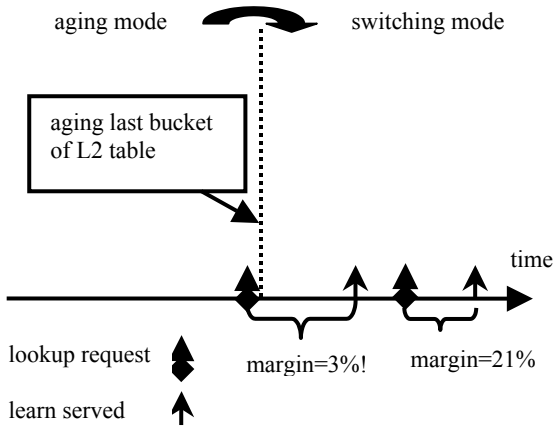

Figure 3 the performance analysis for learns

5.2 Verifying multiple requests scenario

In Section 3, we make an assumption that only one request comes from each port in our abstracted model. The interaction between consecutive requests from the same port is neither modeled nor verified. Ideally, if a request is served, the related state registers return to the reset values. In reality, this is not true because the residual state of the previous request may not be cleared. This may introduce unexpected behavior for subsequent requests. We solve this problem by modeling consecutive requests using a set Q of initial states larger than the original set I of initial states, i.e., $I \subset Q$, and $Q - I$ models the set of residual states. Note that the L2 table is already modeled non-deterministically. Only the initial values for the registers outside the L2 table need to be considered.

Let $M \downarrow req$ be the L2 table model with only one request for each port and p be a universal property on $M \downarrow req$. The key to extend our result is to find the reachable set of states Q. It is a difficult task. Our idea is that, instead of searching for the exact Q, we predict its projection on subsets of registers. Assume that V is the set of registers on $M \downarrow req$. We partition V into m equivalence classes V_i, i.e., $V = \bigcup_i^m V_i$ and $V_i \bigcap V_j = \Phi$. Let Q_i denote the projection of Q on V_i. Then we try to predict Q_i using the model checker. If we fail to predict Q_i, or Q_i is too large, then we use the projection of I on V_i, denoted as I_i. Eventually, we use $Q' = Q_{i1} \times ... \times Q_{ij} \times I_{k1} \times ... \times I_{kp}$ as the set of initial states for model checking. Apparently, we are not able to fully verify our intended property on M with this approximation. However, this extension does increase our verification confidence. With the approximation Q', we uncover a new bug in which the residual state from a corrupted packet's learn request interferes with a subsequent packet.

6 Conclusion

In this paper, we use formal techniques to verify a complicated Ethernet switching table. We start by applying aggressive abstraction to debug the basic logic. Its effectiveness is demonstrated by a number of "high-quality" RTL bugs. Then, based on classic model checking, we extend our effort formally and informally to analyze performance. In contrast to ad hoc performance simulation, such rigorous analysis is beyond the capability of either simulation or formal techniques alone. We also extend classic model checking using a novel induction to avoid state explosion as the environment model becomes sophisticated. Our experience can be applied to many table driven designs without much difficulty.

References

[A00] M. Aagaard ed. *Formal Verification of Iterative Algorithm in Microprocessors*. In Design Automation Conference, pages 201-206, 2000.

[B01] B. Bentley. *Validating the Intel Pentium 4 microprocessor*. In Design Automation Conference, pages 244-248, 2001.

[BD94] J. Burch and D. Dill. *Automatic verification of pipelined microprocessor control*. In Computer Aided Verification, pages 68-80, June 1994.

[CBL98] Cadence Berkeley Lab. *Cadence SMV*. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv, 1998.

[CCLW99] P. Chauhan, E. M. Clarke, Y. Lu, and D. Wang. *Verifying IP-core based system-on-chip designs*. In IEEE ASIC Conference, 1999.

[CGP99] E. M. Clarke, O. Grumberg and D. Peled. *Model Checking*. MIT Press, 1999.

[KKC98] J. Kadambi, M. Kalkunte, I. Crayford. Gigabit Ethernet: Migrating to High Bandwidth LANs. Prentice Hall, 1998.

[M98] K. L. McMillan. *Verification of an implementation of Tomasulo's algorithm by compositional model checking*. In Computer Aided Verification (CAV98), 1998.

[OSSP98] Open Source Software Project. *Embedded Perl Language*. In http://www.ossp.org/pkg/tool/eperl, 1998.