FRCLe based Runtime Verification of Inner Cores for networks

Anu Chaudhary  
Department of Computer Science and Engineering,  
University of Michigan, Ann Arbor  
anuch@umich.edu

Bhavitavya Bhadviya  
Department of Computer Science and Engineering,  
University of Michigan, Ann Arbor  
bhavi@umich.edu

ABSTRACT
High throughput, low latency and high system availability are the most important criteria for an internet router. Internet Router’s complexity has been on constant rise to serve the exploding internet backbone. With rise in design complexity comes the increased possibility of bugs escaping into silicon and affecting the user at runtime. In this project we implement a verilog-based learning router and describe FRCL-based state matching system to guarantee the correct functionality of our router system in case of permanent and transient hardware faults as well as design errors. In [1] it has been shown that FRCL technique has been used to develop a fault-tolerant processor system wherein the core switches to less featured but fully functional inner core on occurrence of an error. The aim of our project is to extend this technique to a network router design. The idea is to detect an unverified state (possible source of error/fault) during runtime and switching to the inner core setup with less efficient routing mechanisms until the disruption due to the fault is corrected.

Keywords  
FRCLe, Inner cores, Runtime Verification, State Matcher.

1. INTRODUCTION
Internet has become the backbone for universal communications. It is not only limited to data sharing but hosts a variety of services like voice, video, information, instant messaging, searching, distributed computation, surveillance, commerce, control, etc. These new applications have an increasing need for performance and availability of the router hardware and software. No doubt, the router design space has become complex and the complexity is ever increasing.

Network routers in general are complex systems each having multiple routing processors, multi-direction shared bus, virtual channels, switching fabric as well as large loads of memory. Designers are well aware of this complexity and hence try to verify the router system extensively before fabrication. Random simulations, theorem proving and model checking form an integral part of the design process. However these verification techniques will suffer from lack of scalability or the non-guarantee of checking for all possible kinds of system errors.

Further, the time-to-market requirements are very competitive and provide designers a very small time window to verify all the possible states. Even a small 32 bit counter would give us $2^{32}$ states to verify and prove the correctness. Runtime verification, thus, provides a good option to fix bugs that escaped the design process.

A number of schemes are already available for runtime verification of microprocessor systems, however similar schemes in the interconnection network field are not very common. We choose to extend one of the schemes to this field. Our project extends the FRICle-based [1] run time verification to internet routers. Of the various runtime verification like Argus, SafetyNet and Casper, we felt that FRCL is the one that best suits our requirements and is low cost and easily scalable. FRCL is implemented using a state matcher - a cam storing the erroneous pattern matching vectors. A pattern match identifies that the router has transitioned into an unverified state and immediately forces it to run in its low performance inner core mode. The inner core is a simple broadcast-based forwarding decision mode which has been completely formally verified in Magellan.

Our router begins in normal mode, which involves lookup based forwarding, and the lookup tables learn with time. A pattern match detected by state matcher results in transition to inner core mode. This mode switches off lookup and involves broadcasting the packets on all ports. Once the packet is safely received on output port, the router returns back to normal mode.

We discuss the type of errors that could be a concern in the normal functioning of an internet router in the next section. In the 3rd Section we discuss the design in its degraded mode as well as in its fully –functional mode.

In the experimental section we depict how the addition of the state matcher in the router design drastically uplifts the loss of packets. More over the addition of the State matcher is insignificant as compared to the entire design. Performance effects are always positive and hence there is no loss in performance as compared to other online verification systems.

2. ROUTER ARCHITECTURE

Our design consists of four nodes, each with one input port and one output port. However, it is parameterized and thus can be extended to any number of nodes.

Each node consists of four modules, namely sonict (box 1 in the block diagram), sram (box 2), data interface (box 3), and filtering and forwarding logic (box 4). The black shaded boxes within each box represents the module-specific state matcher logic. All four nodes communicate with each other using the shared bus.

2.1 SonicT

The SonicT is a common chip for Ethernet interfaces. It includes the capabilities to receive and transmit packets via an Ethernet physical connection. It buffers the serial connection in 2 internal 32 byte FIFOs, one for transmitted packets, and one for received packets. These FIFOs are designated memory locations in the dual port sram.
Upon packet reception, preamble and SFD bytes are stripped. The destination address, source address, data fields, and CRC-32 are buffered in the Receive FIFO. Packets leaving the SonicT are taken from Transmit FIFO and prepended with preamble and SFD bytes before being transmitted out.

2.2 SRAM
It is a 32 bit dual port memory divided into 3 sections:
1. Receive buffers (FIFO) for SonicT. These buffers store packets received by SonicT at the input port.
2. Transmit buffers (FIFO) for SonicT. These buffers store packets coming from the interconnect and destined to be sent at the output port by SonicT.
3. A mailbox to facilitate communication between the SonicT and the Data Interface Module.

The base address of the Transmit Buffers is located at 0001:4000h. This 16kb of memory is divided into eight 2k buffers. These buffers are arranged in a circular queue through the use of descriptors. The base address of the Receive Buffers is at 0001:0000h and are also arranged in a circular queue.

After a frame is successfully received without any error conditions by SonicT, it writes a message to the mailbox address in the dual port SRAM. This message indicates to the Data Interface module that a frame is ready to be forwarded to the filtering and forwarding module for switching.

2.3 DATA INTERFACE
The purpose of this module to facilitate communication between the memory and the filtering and forwarding module. It performs two main activities - retrieve a packet from memory and send it to the filtering and forwarding module, and receive a packet from the filtering and forwarding module and send it to memory. It also reads and writes the mailbox in memory to indicate start and completion of this communication.

2.4 FILTERING AND FORWARDING LOGIC
This module is responsible for the filtering and forwarding decisions of the router. The decisions are facilitated by the lookup table(LUT) residing within this module. LUT is implemented as an associative memory. Each entry consists of a destination address and the corresponding port number on which a packet with this destination should be outputted. An acknowledgement packet writes to update this and all other LUTs, while a data packet initiates a lookup into it.

The filtering and forwarding logic sends a request signal to the data interface which retrieves a packet from memory if one is available. Once a packet is received from the interface, it is filtered based on the packet type. An acknowledgement packet updates the LUT and is then discarded. One of the following may happen for a data type packet:

1. If LUT lookup results in NOT found, the data packet is sent on the 16 bit wide shared bus from where it is broadcasted on all ports.
2. If the lookup results in found, and the port number given by LUT is same as this node, the data packet is discarded (assume to be consumed as it was meant for this node). It is not sent on the sharedbus.
3. If, however, the port number given by LUT is different from the current node, the packet is sent out on sharedbus from where the designated port number grabs that data and initiates the process of sending it out on its output.

2.5 STATE MATCHER LOGIC
We complement the filtering and forwarding module with a state matcher, which is a content addressable memory. This module implements the field repairable control logic which enables the runtime verification. The state matcher is empty by default. If an error is found after design tape-out, the set of states associated with the bug are encoded into a state matching pattern. This pattern is then distributed to customers, where it is loaded into the state matcher. When a buggy state is detected by the state matcher, the router switches to a degraded inner core mode which is fully verified. Thus, we can rely on this mode for the router to complete the next packet flows correctly. After one packet is safely transmitted on the output ports, normal execution mode is resumed.

3. MODES OF OPERATION
The two modes differ only in functioning of the filtering and forwarding logic of the router. While the normal high performance mode makes forwarding decisions based on lookup tables characterized by learning, the inner core mode switches off the lookup logic completely and forwards packets based on broadcast logic.

The flow chart on next page depicts the flow of packets in the router. The shaded area represents inner core mode. Router starts...
corresponding sonicT saves it to dram, from where it is read by the data interface and sent to the filtering and forwarding module.

If the packet is of data type, its destination address is searched in the LUT. If it is not found, the packet is broadcasted on all ports.

If found and the port number for the destination matches port#0, the packet is discarded (i.e. assumed to be consumed here). However, if found and port number returned by LUT is say port#2, the module starts sending data on the shared bus. Rest of the nodes (i.e. all nodes except node#0 the one that received the packet on its input port) begin to buffer this data from the shared bus. However, only port#2 will send this buffered data to its sonicT (which in turn transmits it at its output port).

If it is an acknowledgement type packet, it will update the lookup table. The destination address is the one in the packet, while the port# to be stored for this destination is the port# of this node, port#0 here. This update of LUT is global, in the sense that the LUTs of all the nodes in this router are updated simultaneously.

So, all LUTs have the same contents at all times. LRU policy is used for replacement.

State matcher is continuously monitoring the control vector. As soon as a pattern matches, router switches to inner core mode i.e. Broadcast based forwarding mode. Unlike normal mode, learning and lookup is completely switched off. Acknowledgement packets are discarded, while data packets are sent out on the shared bus. Like normal execution, rest of the nodes begin to buffer this data from the shared bus. However, unlike normal execution, all of these nodes will send this buffered data to their sonicT (which in turn would transmit it at their output ports). So, the data packet will appear at the output ports of all the nodes in the router rather than only one.

Once this data packet is safely transmitted to output ports, router returns back to normal mode.

4. INNER CORE VERIFICATION

Our inner core consists of the router without lookup logic. We formally verified the correctness of the router in this mode of execution so as to allow lossless delivery of packets even in face of faults. We undertook both verilog testbench simulation and Magellan verification. Verification of line coverage, condition coverage and state coverage goals was undertaken using in-built Magellan options, while we wrote SVA assertion properties to cover property goals. We verified properties like all output enables should eventually go high in inner core mode because we broadcast on all ports.

Another interesting property assertion we learnt was the use of Sstable. The property below checks that the current active node, out of the four nodes, stays the active one (round robin decides which one is active) until it finishes its work. Sstable can be used to write ‘hold ownership’ kind of properties.

5. TYPES OF TARGETTED BUGS

A hardware design is prone to both permanent and transient faults. We choose to introduce the following three faults in our design to experimentally evaluate the FRCL logic:

1. Permanent fault in Filtering and forwarding module

   We introduced a design error in the logic initiating lookup in filtering and forwarding module. This error results in the router going to IDLE state even before it has transferred data on the shared bus. Unless this data goes on sharedbus, it cannot be input by other ports to send out on their outputs. A router without FRCL and with this error will always cause loss of incoming of packets. But our router will be able to detect this bug (state matcher will have a pattern for this state of the system) and thus initiate inner core. Inner core by-passes all lookup related logic and hence guarantees continued flow of data.

2. Permanent fault in LUT

   We introduced this bug by permanently providing an invalid port number on the output of LUT. In non-error conditions, the output of LUT should indicate a port number between 0 and 3 inclusive. We wrote a 9 as port number for destination address 48’h787878787878. We are trying to model a situation in which either an entry in LUT gets incorrectly initialized or a situation when the output of LUT is stuck high or low. A router without FRCL when hit by this error will always cause loss of packet meant for destination address 48’h787878787878, however there is no loss in our design.
3. Transient fault in LUT

We modeled this by writing an invalid value on output of LUT for a specific number of clock cycles. We used a counter to get this effect. While our design will experience no loss of packet due to switch to inner core mode, a router without FRICLe will lose packets meant for destination 48'h787878787878 during the time duration of the error.

6. EXPERIMENTAL RESULTS

We evaluated our implementation on the basis of overall gain in the availability (reliable transmission) of the router. The results were obtained over two modifications of our router design viz. with FRICLe and without FRICLe design additions. Each of the FRICLe based design have a common 12bit wide state matcher. The required error prone vectors have been already fed into the CAM during system startup. We used NCVerilog to simulate the packet inflow(acknowledgment packet) and outflow.

Fig 2. Shows the original latency of each packet incident on the input ports for different scenarios without any errors.

The class of errors we considered are:
1) Permanent electric Failures
2) Transient electric errors
3) Design Errors

Using permanent failure experiment we have successfully modeled errors caused in the router due to physical failure of transistors or the interconnect. As depicted in Fig 3 we can see that there has been considerable loss in packets at all ports in design without our proposed implementation. Transients faults also behave in similar ways as we can see in Fig 4.

More interesting experiment of the proposed design is its ability to keep the router functioning even in case of errors caused due to the designers themselves. As you can see in Fig 5 that the router with built in state matcher can successfully transmit all the packets even though the normal router completely fails.

Apart from the above performance analysis we also synthesized our designed using Synopsys’s DC compiler and found out that the CAM state matcher overhead is merely 0.12 percent.

7. IMPLEMENTATION INSIGHTS
The project was a good learning experience for us, especially since we came across a number of unforeseen issues. This was the first time we were dealing with a dual clock design. Sonit module interacts with outside world at 1.25 MHz and with rest of the components of the router at 20 MHz clock. Dual clock design posed two challenges. One was that it complicates writing the vera code for packet generation. It was interesting to explore the way to do it for two clocks. We are now being able to generate random destination and source addresses, variable packet length packets, and single or burst packet inputs.

The other challenge led to an interesting discovery that magellan does not start verification on a non-synthesizable design. The original sonit module was written as a behavioral module involving two clocks, and hence was not synthesizable. We restructured the module to produce a synthesizable version. We came across an unusual error while magellan verification. Our design consists of a def file which consists of all the `define we have used in our design. We were not able to add this file to our magellan project properly. Though we fell back upon the option of including the `define in the specific verilog files themselves, but we would want to figure out the reason for this unusual case.

8. FUTURE WORK
The sharedbus routing architecture used in our implementation is very simple and definitely does not give us full freedom to explore more benefits from FRICLe. It was difficult to introduce the error as there was very . Extension to this project could be to add more functionality to the advanced features of the already existing router. In future we would definitely like to explore more advanced routing architectures and compare their performance as well as reliability increment from the use of state matcher based run time verification. Another aspect of exploration could to extend this work over to a network of routers

9. CONCLUSION
FRCL (field repairable control logic) is a runtime verification technique introduced by [1] for a processor system. In our project, we attempted to extend FRCL to a network router design.

Starting with a lookup based router with two input/output ports, we extended it to a four port learning router with an ability to operate in two modes. It starts in normal mode where the learning lookup based forwarding decision logic provides high performance. The FRCL state matcher logic can detect when the router enters a bug state and switch to a low complexity broadcast based inner core mode until the bug is bypassed. This mode guarantees delivery of all incoming packets.

We analyzed our design on three types of errors - permanent design error in forwarding logic, permanent fault in lookup table and transient fault in lookup table. In all cases, our design guarantees lossless delivery of packets whereas the design without FRCL results in loss of packets. Performance, in terms of throughout, increases. Latency suffers as compared to normal mode case when no contention for the output ports exists. Inner core fairs same on latency as the normal mode case when all packets are directed towards same port.

Overall, our FRCL based router design provides low overhead in the case of fault-free execution, and guarantees lossless delivery of packets in case of faults.

10. GROUP DYNAMICS
Our project broadly involved developing three aspects - building upon the base design, magellan verification of inner core and developing the state matcher. Anu was mainly involved with enhancing the base design by adding broadcasting for inner core and learning lookup for high performance mode, while Bhavi concentrated on developing the state matcher. Packet generation using vera, and modification of base design towards making it synthesizable was also worked upon by Bhavi. We both worked together while verifying the design in verilog and magellan.

11. ACKNOWLEDGEMENTS
Our sincere thanks to Prof. Bertacco for providing the base code for a 2 port router/switch. Her inputs during the checkpoint meetings were also highly valuable.

12. REFERENCES
[3] SystemVerilog Assertions Design Tricks and SVA Bind Files”; Clifford E. Cummings