Project Title: Robust Cache Coherence Protocol Verification with Inferno

Team Name: FLY-Bee

Team Member: Zeyu Bu, Xiangfei Kong, Chenxi Lou, Yao Jiang

- **Problem to be addressed:**

Fault-tolerant architectures are emerging to guarantee reliable functionality on vulnerable silicon devices. For a fault-tolerant architecture, the RTL design is more complex than a normal one and is more likely to introduce design bugs.

- **Why does this problem matter?**

Fault-tolerant architectures are popular in current circuits design industry. However, the verification of such a complex RTL design is a time consuming and costly task that is likely to become a bottleneck in the release of new architectures. Thus, finding an efficient way for debugging would speed up the verification process.

- **Idea/Solution to be investigated by the project:**

This project will focus on the debugging process of a fault-tolerant directory-based 'MSI' protocol. We will also apply Inferno++ in our debugging process and evaluate this novel tool. To start with, the robust protocol proposed in 'A Systematic Methodology to Develop Resilient Cache Coherence Protocols' can be used to implement the resilient architecture. In the process of implementing the resilient architecture, we expect to run into several bugs. Both traditional debugging methods and Inferno++ will be used to locate bugs and their efficiency will be evaluated and compared in a systematic way. In our case, the efficiency of inferno++ will be evaluated based on its ability of locating bugs. Our analysis should provide a reference for future verification process of complex designs.

- **Progress so far:**

We finished the baseline integration before checkpoint 2 and focused on developing a robust MESI directory based coherence protocol, developing assembly tests and debugging. Also, we put Inferno into use in this stage. Following are some details about our progress.

✓ Robust MESI protocol

The idea is mainly based on the first presentation paper in EECS 578 class [1]. We implemented additional safe states and handshaking mechanism to ensure the design's ability of resending packets and tolerating duplicates is unreliable. However, there are a few custom design parts that is not mentioned or different from the paper we referenced to.

1. In our implementation, if the directory cannot process some coherent messages in a certain transient state, the directory will drop the coherent messages without sending No-ACKs back. Since our protocol is resilient, the initiator will resend requests again until the directory process it.

2. We implemented the complete state machine the paper does not describe in full detail. For example, if the directory in state M and one core send a GetS request, some transient states are added to make sure the data can be resent and the protocol is resilient.

✓ Test development & robustness check

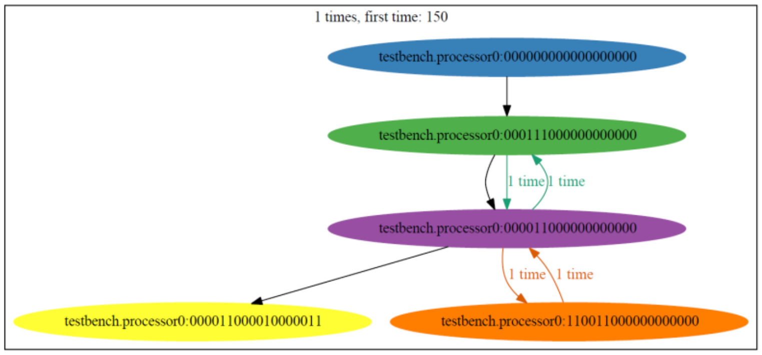
In the design we let each core has its own instruction memory and we feed different assembly programs to corresponding cores in parallel. Tests developed are divided into two categories: 1) *check the functionality of the design* 2) *check the robustness of the protocol when suffering from packet drop or data corruption*. These tests mostly involve communication between three cores and the directory and are quite complex in perspective of message passing.

As discussed in Checkpoint 2 meeting, we implemented 'drop packet' control inputs to test the design's robustness. The basic idea is to use the control signal to intentionally mask receiving packets at each node's ejection channel. We drove these control signals in the test and checked if the initiator is able to resend the lost packet after timeout and if the receiver can tolerate the duplicate if it's just a false positive.

✓ Inferno

We have put Inferno into our normal debugging process and it turned out that Inferno can give some help for detecting illegal/unexpected behavior at router-core interfaces. The way how we utilized Inferno is to monitor signals at each router-core interface and check the send/receive packets at the interface.

After some trial and error, we found that monitoring signals like the send/receive message type, destination node address, source node address (possibly some other signals) can provide the verification engineer sufficient information to understand the behavior at the interface. The following diagram is an example diagram we got when running Inferno.


1 times, first time: 150
testbench.processor0:000000000000000000
testbench.processor0:000111000000000000
1 time
1 time
testbench.processor0:000011000000000000
1 time
1 time
testbench.processor0:000011000010000011
testbench.processor0:110011000000000000

testbench.processor0:core_1_proc2router_cmd, core_1_proc2router_dest, core_1_router2proc_VC1_cmd, core_1_router2proc_VC1_req, core_1_router2proc_VC2_cmd, core_1_router2proc_VC2_src

- **Issues/Showstoppers:**

In general, Inferno is helpful when the trace signals are carefully selected and limited in a small number. Therefore, monitoring interfaces separately is a better idea. Also, for our project, we keep documenting bugs and debug efforts we put in our 'debug log'.

In next week, we will collect results from our design and analyze the debug efforts we put with and without Inferno. This will be finalized into the evaluation of the tool. We will also try to regenerate erroneous Inferno diagrams that helped us to find design bugs. The example use in our project can give inspirations to people who would like to use Inferno in future.

Reference

[1] K. Aisopos and L.-S. Peh, "A Systematic Methodology to Develop Resilient Cache Coherance Protcols," *MICRO'11,* 2011.