Simulation Based System Level Fault Insertion Using Co-verification Tools

Bill Eklow, Cisco Systems Inc., San Jose CA, beklow@cisco.com
Anoosh Hosseini, Cisco Systems Inc., San Jose CA, anoosh@cisco.com
Chi Khuong, Cisco Systems Inc., San Jose CA, ckhuong@cisco.com
Shyam Pullela, Cisco Systems Inc., San Jose CA, spullela@cisco.com
Toai Vo, Cisco Systems Inc., San Jose CA, toavo@cisco.com
Hien Chau, Cisco Systems Inc., San Jose CA, hichau@cisco.com

Abstract

This paper presents a simulation-based, fault insertion environment, which allows faults to be "injected" into a Verilog model of the hardware. A co-verification platform is used to allow real, system level software to be executed in the simulation environment. A fault manager is used to keep track of the faults that are inserted on to the hardware and to monitor diagnostic messages to determine whether the software is able to detect, diagnose and/or cope with the injected fault. Examples will be provided to demonstrate the capabilities of this approach as well as the resource requirements (time, system, human). Other benefits and issues of this approach will also be discussed.

1 Introduction

Fault insertion continues to be an essential tool to verify the robustness and error handling capabilities of board and system level diagnostic and operating system software. For the case of diagnostics, fault insertion may be the most accurate and objective method to determine the ability of diagnostics to accurately detect and diagnose failures at a board and/or system level. Manual fault insertion (hard wiring faults to the PCA) has been a costly and time consuming task. For this reason, only a small subset of faults can be inserted on to the board, significantly reducing the benefit of fault insertion. Today as access to nets on the board is becoming increasingly difficult, manual fault insertion is becoming less and less practical. With greater frequency requirements, it may be impossible to do manual fault insertion without impacting board performance or causing additional failures, which may mask the ability of the software to detect or cope with the original fault that was injected. Automated, probe-based techniques exist; however, these techniques require sufficient mechanical access to every net in order to be effective. Again, on most high complexity boards this access is either not available or will severely impact the performance of the board (to a point where the access itself is causing failures). Another alternative uses the Boundary-scan logic [1] to insert the faults. This method has been presented by several authors [2], [3], [4], [5]. It has several advantages over manual fault insertion. Since the fault insertion is built into the hardware, no wiring has to be done. This makes the boundary-scan based approach much more practical and much less of an impact on design performance.

Hardware/software co-verification techniques have gained in popularity with the increasing use of system-on-chip (SoC). Perhaps an even more compelling rationale for co-verification is time to market requirements resulting in much smaller design cycles. Design teams can't wait until hardware is designed and debugged to start software development, and they can't afford long, iterative loops caused by spec misinterpretations. Because of design and dollar costs associated with design spins for complex ASIC's and SoC's, it is imperative that the design be done correctly the first time. Failure to identify bugs early in the design process often translates into huge repair costs and, even worse, can lead to the death of a product due to a missed market window [6]. All of the fault insertion schemes described above occur after the design has been completed and the product has been assembled. Any hardware errors which are detected by fault insertion at this time may be costly to repair, or may not be repaired at all. By performing fault insertion in a verification environment, problems can be fixed with a minimal cost or schedule impact.

 0-7803-8580-2/04 $20.00 Copyright 2004 IEEE

Co-verification can provide additional benefits to the ASIC/SoC designer as well. Software and diagnostic engineers have much earlier access to the hardware. This allows software designers to develop code and test it concurrently with hardware design and verification. Co-verification also provides additional stimulus for the ASIC/SoC design. In fact, it can provide the true stimulus that will occur in the real system. Hardware test benches typically focus on verifying external interfaces of the chip. Problems such as incorrect register mapping and bus pipelining may not be found by the hardware verification process. If these problems are not found during verification the results could be significant cost and/or schedule delay. The net result is that bugs (both software and hardware) can be found earlier in the design cycle when they are easily to identify (in the simulation environment) and fix.

2 The Co-verification environment

Co-verification techniques have been used to test diagnostic and operating system software for several years. A typical co-verification platform is shown in Figure 1.

The co-verification platform is “transaction-based”. Instructions executed in the software environment are passed from the software environment to the hardware simulator through a PLI interface. These messages are translated into bus cycles by the hardware interface component of the co-verification environment, which includes some type of model of the on board microprocessor. Responses are captured in the hardware environment and forwarded back to the software environment where the next instruction will be fetched and executed.

The hardware simulation is expected to “host” a certain region of memory within the memory space. Any “reads” or “writes” to that region are translated into transactions to the hardware simulator. Accessing any other region in memory is treated as regular memory on the native workstation. These memory regions can be defined by an external memory map which is loaded in the software environment.

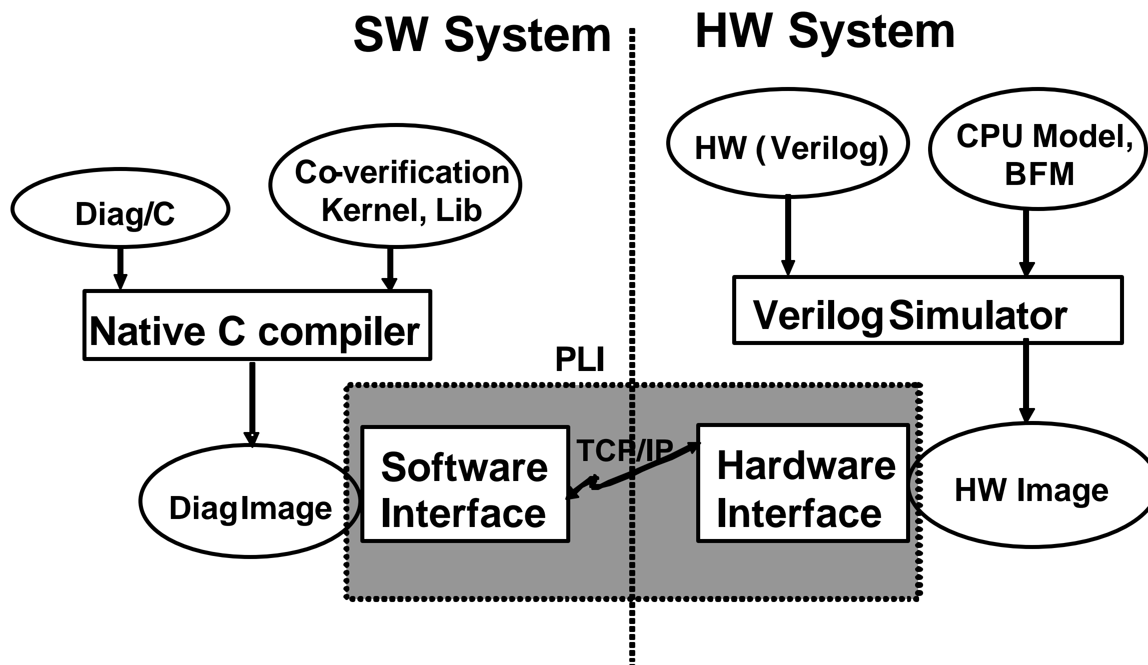

Figure 1

Consider the following code segment:

```
// SW Env. Example of code read
Uint32 *base_addr = (Uint32 *) 0xB0000000;
Uint32 data = *base_addr; // READ
```

Assuming base_addr is NOT in the software environment address space, a segment violation will be generated by the software system. The co-verification software interface will intercept the segment violation, check to see if the address is in hardware address space (as defined by the external map file), translate the address into physical hardware address, then request a READ transaction to the hardware side. The CPU model generates the read cycle in the hardware simulation environment, captures the return data and sends the data back to the software environment. The software environment then proceeds to the next instruction. A write instruction is similar to the read instruction however, the software environment will not wait for any return data from the hardware environment before proceeding to the next instruction.

3 The Fault Insertion Environment

Stuck-at faults can be injected into the hardware simulation environment by accessing the appropriate net via a PLI interface and forcing it to a zero or one. The PLI interface translates commands sent from a "Fault Manager" (the Fault Manager will be discussed in the next section) into a sequence of operations which are invoked in the verilog environment. This method of forcing faults through a PLI interface is much more advantageous to fault insertion by editing and recompiling the verilog netlist. Such a method would require a lot of manual intervention, taking significantly longer than an automated method and increasing the risk for mistakes during the insertion process. By utilizing the PLI interface; the fault insertion process can be totally automated, requiring only a list of faults to be inserted into the design.

Shorts can also be inserted into the design through the PLI interface. Both "wire-or" and "wire-and" shorts can be inserted by monitoring each of the affected nets, applying the logic function (and/or) to the observed nets and then forcing both nets to the value obtained by applying the appropriate logic function to the nets. This method can also be used to inject "transient effects" which could be tied to or triggered by a single net or a combination of nets. It is the PLI interface which monitors each of the "shorted" nets for a transition and then forces both nets to the appropriate state based on the new state of the two nets.

At this point, there is no practical way to insert an open fault on a net. While it is possible to drive a net to a "Z" value through the PLI interface, the "Z" would propagate as "X's" through the design making it difficult to observe the effect of the fault. It is hoped that in most cases, open faults can be modeled by some combination of stuck-at, transient and bridging faults.

4 Integrated Environment

Co-verification has been performed for diagnostics on several products at Cisco. This section describes what was done to help build the integrated co-verification/fault insertion environment at Cisco. A specific example is described later in the paper.

The software side of the co-verification environment was created by compiling a "sanitized" version of the diagnostic code. In the "sanitized" version, all assembly level code (interrupt handlers, optimized drivers, kernel code if the diagnostics are being run under a diagnostic kernel) was either removed or translated to C. The code was compiled by a native compiler (UNIX), and linked with library elements from the co-verification tool. These library elements enabled the data transfers between the software environment and the hardware environment. A memory map is also included (as an external file) to define which accesses need to be done in the "hardware environment". It is possible to specify a logical to physical translation in the memory map as well.

The hardware environment is created by compiling the system level netlist. To do this, the system level schematic must be translated into a verilog netlist. A perl script was written to do this translation. Since most of the ASIC's (and FPGA's) on the board were still under development, a set of tools had to be written to map the logical I/O in the device into the Verilog netlist. These tools addressed a series of issues that required manual intervention and therefore were very time consuming to resolve. Among these issues were: dealing with discrete components, mapping logical signals from the ASIC to the board, resolving library naming issues, and finally integrating multiple boards into a system by creating a virtual backplane. Power

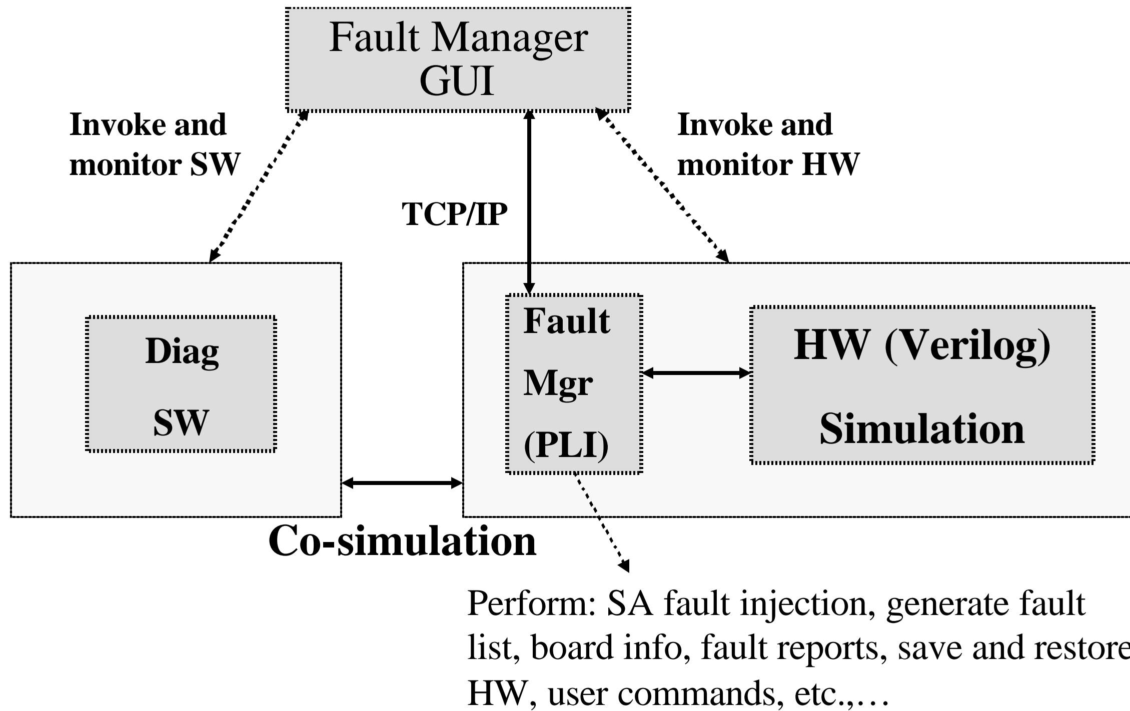

Figure 2

and ground checking is currently being worked on. These tools allowed the designers to quickly recompile the hardware environment after any changes which were made to either the ASIC/FPGA or schematic. The tools reduced the development time for the initial environment from several weeks to a few days, and reduced the time to rebuild the environment after a design change from several days to a few hours. The tools also eliminated design errors which occurred due to the manual effort required to develop and sustain the environment.

Obviously, there was a requirement to create several new component models to support each design. In most cases the models were relatively trivial. However, in some cases the development time for component models could be significant. In cases were models were provided by the vendor, costs could be significant and a significant schedule risk could be incurred if the model was not available and the vendor was required to develop the model. Simulation models for complex processors could take up to 6 to 12 months to complete.

The fault insertion environment is shown in Figure 2. A "Fault Manager" controls both the fault insertion and diagnostic execution. A user is able to build a fault list by selecting a fault type, and net name(s) or pin number. In addition, the user can select which diagnostics to apply to the given fault. The fault manager will select the next fault from the fault list, apply the fault as described above and then run the corresponding diagnostics. The fault manager communicates to the hardware and software environments through a "pseudo terminal" interface. The pseudo-terminal interface translates required actions on the fault list and diagnostic execution list into commands for the simulation environment and the diagnostic command line interface. Status and error information from the hardware and software environments are reported back to the fault manager through the pseudo-terminal interface.

A typical fault insertion action would proceed as follows:

1 The fault manager starts both the hardware and software environments. The hardware and software environment are synchronized by a power on reset associated with starting the hardware environment. The software boots to the diagnostic command line interface.
2 Once the software has booted up, the fault manager forces a fault (from the fault list) on

the appropriate net through the PLI interface.

3. The fault manager then sends a command through the pseudo-terminal interface to the diagnostic command line interface causing the first diagnostic in the diagnostic execution list to be run.
4. The fault manager captures and parses information returned from the diagnostics through the pseudo-terminal interface. Once the fault manager determines whether a diagnostic has passed or failed, it records that information on a fault reporting list. If the diagnostic passes, then the fault manager will restore the hardware environment to its initialized state through the pseudo-terminal interface, pull the next diagnostic from the diagnostic execution list and send the appropriate command through the pseudo-terminal interface to execute the diagnostic.
5. This process continues until the fault manager determines that one of the diagnostics has failed (based on the diagnostics messages returned from the software environment) or none of the diagnostics have detected the fault. The fault manager then updates the fault status (detected or not detected), restores the hardware environment to its initialized state, forces the next fault, and begins execution with the first diagnostic on the diagnostic execution list. This process then continues until all faults have been inserted or the user aborts the run. Note – since there is no way to restore the initial state of the software, diagnostics must operate independent of the state of the hardware.

5 Results

As was mentioned before, co-verification has been performed at Cisco to verify diagnostics prior to having real hardware. The following data was taken from the co-verification run on an OC-192 board for one of Cisco's high end routers. This card had 5 ASIC's comprising a total of more than 15 million gates, 2 complex FPGA's, a MIPS processor and several other large to medium size components. There were approximately 5000 to 7000 nets on the board. Approximately 110 diagnostics were written against the board. The co-verification environment operated at approximately 340 cycles per second. This operating frequency was obtained with the help of a hardware accelerator. It was determined that the diagnostics achieved greater than 99% net toggle coverage. Simulation time was approximately 2 days for an error free pass of the diagnostics. The co-verification platform uncovered several hardware and diagnostic issues. There were no bugs discovered during the bring-up of real hardware. However, since fault insertion was not performed on this design, a bug in the error recovery software (recovery from memory errors) was not detected in simulation and subsequently took 3 to 4 weeks to debug on real hardware.

Co-verification tools were also used to do early diagnostic and hardware verification on another high end router. Several hardware problems were discovered including: a mis-match between the pinout on the ASIC and the pinout on the board, incorrect decode logic implementation and incorrect routing of write enables. There were several diagnostic problems that were also discovered in the co-verification environment. Most significant was incorrect setup of ASIC control registers during traffic testing. This problem was identified in the co-verification environment by tracing packets through the ASIC in the verilog environment. Debugging on real silicon would have taken significantly more time. On boards which this level of diagnostic simulation was performed, diagnostic bring-up time was less than 24 hours. This was compared to over 2 weeks for boards where no simulation had been performed.

"Simulated" fault insertion was performed on a route processor board for a mid/high end router product. Cisco does "post fabrication" fault insertion on many of its router/telecom products. Between 50 and 75 faults are manually wired on to the board. Generally, it takes approximately 2 weeks to insert the faults and manually run the diagnostics. As mentioned above, it often takes several weeks to debug issues which are discovered during the manual fault insertion process.

The fault insertion was done on a portion of the board which does the packet processing. The logic consists of three ASIC's which comprise a "fast forwarding engine". Two of the ASIC's (T0, T1) consist of a multi-processor array and are used for packet processing. The third ASIC (V0) is a support ASIC and is used for providing packet information to the packet processing ASIC's and then forwarding the packet to the correct port. The ASIC's are several million gates each. Each ASIC has its own "packet buffer" memory. The fault

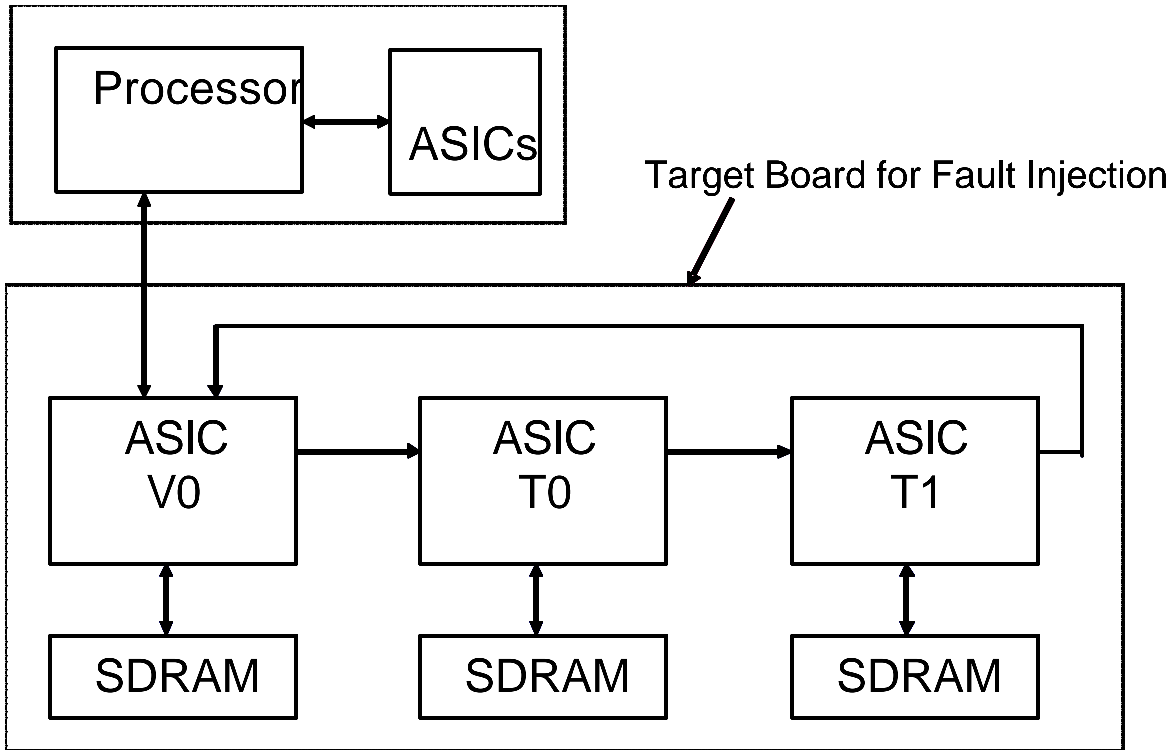

Figure 3

Insertion environment is shown in Figure 3. The on-board processor and other support ASIC's are not part of the fault insertion environment, but are included in the co-verification environment.

Simulation time averaged approximately 1 hour per fault. A hardware accelerator was not used as part of this project. It is estimated that the hardware accelerator could reduce simulation time to as little as 6 – 12 minutes per fault. There were 2569 possible faults which could be generated in the circuit. Of these faults approximately 100 faults were simulated. All busses had only a small portion of the faults simulated. Diagnostic ordering was optimized for each fault to try to detect the fault with the minimal number of diagnostics. In fact, almost all faults were detected after running only 3 diagnostics (there were 52 diagnostic tests written for the board). Based on the simulation results, it was projected that the diagnostics could detect over 99% of the 2569 faults.

6 Conclusion

Co-verification tools can provide significant benefit in the "pre-silicon" debug of both hardware and software. For diagnostics, co-verification tools can help to assess the effectiveness, as well as the robustness of the diagnostics. Fault insertion is another means to determine the effectiveness (and robustness) of the diagnostics. Generally, fault insertion is done "post-silicon", and as such loses some of its benefits. This paper presents a technique for performing fault insertion in the co-verification environment where the maximum benefit of both processes can be derived. There is a significant amount of work required to create this fault insertion environment. Faults are, however, easier to detect and fix in the co-verification environment. More importantly, bugs are fixed prior to silicon tape out or board fabrication, where the design or software changes are much more expensive.

7 References

1. IEEE Standard Test Access Port and Boundary-Scan Architecture, 2001.
2. Savio Chau, “Fault Injection Boundary-Scan Design for Verification of Fault Tolerant Systems”, International Test Conference, 1994.
3. Richard Sedmak, “Boundary-Scan: Beyond Production Test”, 12th VLSI Test Symposium, 1994.
4. Phil Wilcox, Jim Hjartarson, Robert Hum, “Software Verification by Fault Insertion”, U.S. Patent # 5,130,988, 1992.
5. Benoit Nadeau-Dostie, Harry Hulvershorn, Saman M. I. Adnan, “A New Hardware Fault Insertion Scheme for System Diagnostics Verification”, International Test Conference, 1995.
6. Cheryl Ajluni, “HW/SW Codesign and Coverification Come of Age, Electronic Design, June, 1992.