2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA
Sept 25-26, 2014
Hotel Park Plaza
Bangalore, India

Pre-Silicon Debug Automation using Transaction Tagging and Data-Mining

Kamalesh Vikramsimhan, SenthilKumar Narayanaswamy, Deepak Sadasivam, Kaustubh Godbole
Engineering Service, Infosys Limited, Bangalore, India
kamalesh@ieee.org, {senthilkumar_n03, deepak_sadasivam, kaustubh_godbole}@infosys.com

Abstract— **Today's SoCs and ASICs are built by integrating IP's developed in-house or sourced from third party. With the increase in these IP's in system, debugging complexity also increases and also the time taken for debugging failures. Up to 30% of chip design cycle time can be consumed by top level debug and late stage bugs. Typical debug engineer had to glance through hundreds of known errors and failure signatures day after day, before hitting the important new failure signature that could potentially turn into RTL bug. Process is error prone and bugs silently remain in the RTL for many weeks because of lack of bandwidth of debug engineers, to get to those log files, which have new bugs. With many of the team globally dispersed, co-ordination and communication of issues also becomes significant overhead.**

This paper highlights attempts of improving debug productivity and effectiveness. With the objective of reducing debug fatigue in engineers, tool progressively added features for failure knowledge management, automatic triaging of known bugs, architecture based bucketization to effectively reduce workload and thus enabling through analysis of failures. An initial attempt has been made to investigate applicability of data sciences in ASIC debugging.

Keywords— debug; auto-triage; bug tracking;

I. INTRODUCTION

The integration of multicore CPUs, graphics coprocessors, modem IPs, multimedia IPs and networking components into SoC's is making verification and debug a complex task. The increased pace of development and shrinking the design to tapeout time, adds to the challenge. Considerable time is devoted by both design and validation teams to debug failures in cluster and fullchip level. Many of the larger SoC's teams deploy dedicated debug teams to debug and close issues at SoC level. Closure of bug sightings at SoC level is very crucial as this is the last step in which a bug can be caught and fixed. Bug slips at fullchip regression leads to silicon bugs that are very expensive to fix.

The bulk of the time spent by the debug engineer at SoC level is to eliminate known issues. This recurring voluminous work results in debug fatigue and hence leads to catastrophic bug slips. Hence the need was felt to automate the debug. There was also need for better communication and co-ordination between team mates spread across the multiple geographies. The regular daily pass down done in meetings was less effective resulting in non-closure of important debugs or repeated debugs of the same issues.

Considering these challenges we felt the need for automation, knowledge repository for known failures and where ever possible automatic triaging of bugs. There was also a need for better bucketizing method as the traditional approach used resulted in too many overlapping buckets.

II. PREVIOUS METHODS AND ISSUES

The fullchip debug team was responsible for regression, debug and closure of debugs at fullchip. Automation done on regression part by the team was fairly robust. Once the regression was done, the usual practice is to populate the excel sheet with simple bucketing. The buckets are then distributed to individuals and daily stand-up meeting was organized to co-ordinate and prioritize the tasks.

The following are the observations from the prior methods.

A. Debug Coordination Bottleneck

The failure debug at the top level in a chip is a very involved task as it includes system level scenarios and issues from multiple units funneling up at top level. In fullchip, a single bug can potentially take multiple avatars and appear as many different issues. In cases where debug teams are small, debug coordination can be better handled with routine sync-up between the debug coordinator and his team. In cases where the debug team is fairly large, the sheer number of simultaneous issues that needs to be tracked and closed, along with overlapping debugs between individuals becomes a challenge. In programs where the debug teams are geographically dispersed, the debug coordination is an acutely challenging task.

While the traditional system works well for a small debug teams, there was a need for tool chain that aids debug coordination, especially issue like failure knowledge management, prioritization etc.

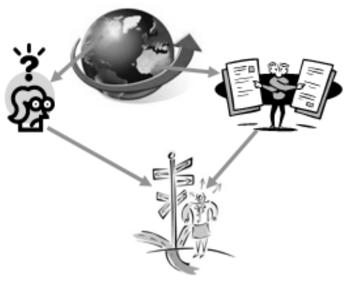

Figure 1. Practical difficulties across geographical sites

B. Bandwidth of Debug Engineers

In a large SoC, failures can arise because of many issues like testcase, test environment, configuration issues in testbench, wrong use cases and also RTL bugs. The key priority for the debug engineer, is to get to RTL bugs faster. The more the time one takes for triaging RTL bugs, more expensive it is to get the bugs fixed. Late stage RTL bugs identification results in de featuring, sub optimal fix in RTL design and in some case re-spin also. Hence there is a significant need for getting to the RTL bugs faster.

The one aspect that consumes debug engineers bandwidth, preventing them to get to the potential RTL bugs is the bunch of well-known bugs that were identified in previous regression that needs to be manually sorted. These well-known bugs appear in many different avatars and debug engineer has to invest good amount of bandwidth in manually looking for markers in the log file to identify the known failures and disposition them.

C. Inadequate Bucketization

The most widely used bucketization method is to go by scoreboard checker error. This bucketization often proves to be inadequate if we consider that many events inside the DUT can lead to a similar scoreboard error. Though architecture based scoreboard is very important from verification point of view, from bug classification side, there is a need for more accurate method. One way of adding more accuracy to the bucketing, is by including micro-architectural checkers. This method gives fine grain classification of bugs, but results in way too many buckets. Every known issues could potentially be located in many buckets and it increase the workload of the debug engineers to tag and disposition numerous buckets.

There appears to be a need for bucketization, which is both accurate, and also results in fewer meaningful buckets. This alone can significantly improve the efficiency of the debug team.

D. Knowledge Sharing

The usual practice for the debug engineers to communicate identified failures is through emails or through bug tracking systems. The bugs are described by identifying the markers in the log file so that others can identify similar failures. The problem often faced by the team is that a verbally or descriptively identified failure markers are either difficult to remember or could potentially be incomplete. This leads to multiple engineers investing time on same known bugs. Such communication gaps become more significant with team not being co-located and time zone variations between the teams.

Hence there is a need to bring in tools that effectively makes the debug engineers focus on new signatures and leave the well-known bugs to be automatically dispositioned. In other words there is need for knowledge management utility that effectively captures the well-known failure signatures.

III. Automation Philosophy And Architecture

The automation philosophy is governed by following objectives,

- Capture knowledge of well-known failures in the tool
- Enable automatic triaging where ever feasible
- Make bucketing meaningful and potentially reduce the number of buckets
- Assist the debug coordinator in prioritizing the debug

With these objectives in mind, automation architecture was created which is depicted in Figure 2. The following are the components of the tool - Signature Extraction module, Error Signature Database and Bug Database, Transaction Tagging, Architecture model.

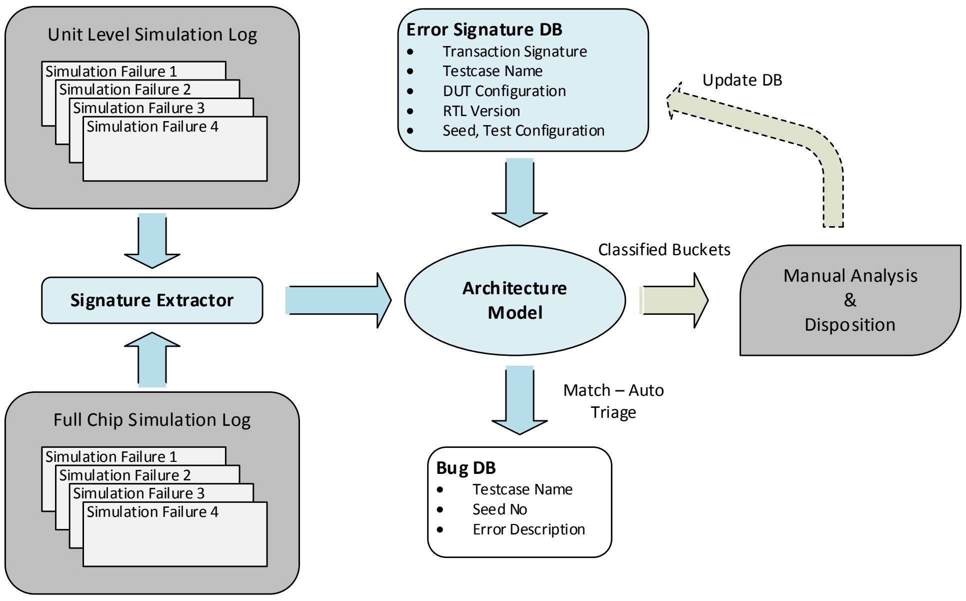

Figure 2. Architecture of the Tool

A. Signature Extractor

Signature extractor is the parsing module that extracts meta-data from the log files. The primary source of data is full-chip log files. Unit level log files can also fed into the system with extra tags which the architecture model can use for correlation. The key fields to be extracted for failures are the ones like testcase names, test group, DUT configuration settings, version number of RTL and testbench components, transaction trace, and the phase of simulation where error has occurred. The signature extractor collates the data, sequences it with time and feeds it into the architecture model, which attempts the correlation with known failure signatures.

B. Error Signature Database and Bug Database

The error signature database contains table, with entries populated from "Signature Extractor". In addition to signature, database also has handle to the bug id that the failure signature belongs to. In case of unit level signature, special tags are maintained so that architecture model can use those tags in addition to signature for correlation and eventual triaging.

Bug database contains the table entries with bug's description, state of bug, location of bug etc. indexed by the bug ID.

C. Transaction Tagging

One of the key enablers for this level of automation is transaction tagging. The key concept here is to abstract the interface level signals of the DUT into transactions and also ability to sequence those transactions in time. Such sequenced transactions give us ability to observe functioning of DUT and serves as marker for describing a failure. There are multiple methods to tag a transaction in DUT. In this paper we want to focus on using transaction tagging to automate triaging. The figure 3 describes how a tagged transaction appears inside a DUT.

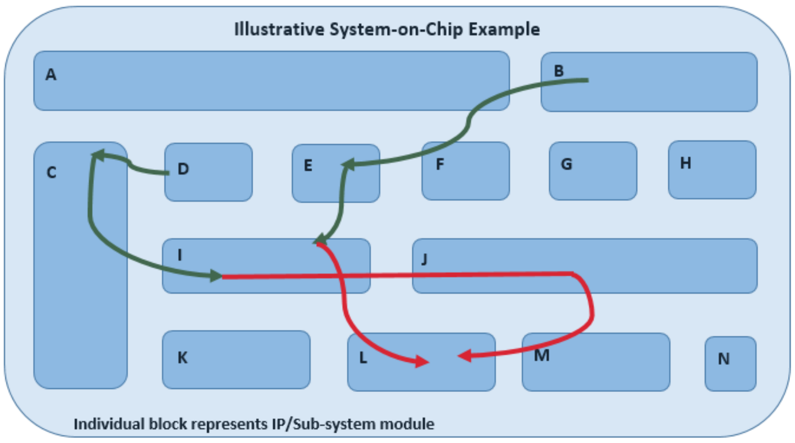
Illustrative System-on-Chip Example
A
B
C
D
E
F
G
H
I
J
K
L
M
N
Individual block represents IP/Sub-system module

Figure 3. Shows different error paths from unit/module across System-on-chip

The indicated SoC has IPs/Modules names between A to N inside a DUT. Let us assume two different transactions hit an error in module "I" and error propagated to subsequent downstream modules. The "Signature Extractor" has ability to extract the transaction sequence or paths taken by the transaction along with fault location. Transaction in DUT, validation environment and error logs are extracted and grouped. In the specific case shown in the Figure 3, rules in the architecture model can be coded such that, both the error transaction shown using arrows, falls in same bucket. If the bucket is well known failure, then the failures can be automatically triaged.

Transaction tagging allows the debugger to form buckets based on various parameters. In certain cases, bucket can be defined based on position of divergence point between DUT and validation environment. In some other cases buckets can be defined based on DUT configuration and Test Group. Such bucketing that are based on architectural details of the DUT, transaction, DUT configuration, test group are much more well defined and trustable way of bucketing than traditional plain checker based classification. In our own example such signature based bucketization reduced the number of buckets by up to 90 percent in certain regression runs. Reduced buckets alone yielded the significant benefit of reduced workload for the debug team.

D. Architecture Model

Architecture model is the heart of the tool that does the search, match to correlate the known failures with the error log files. The first set of search is for direct match to well-known failures which were debugged. The match is done with the signature extracted from log file like test group, test name, transaction trace etc. If there is a match with the rule base, then automatic triaging is completed for those failures.

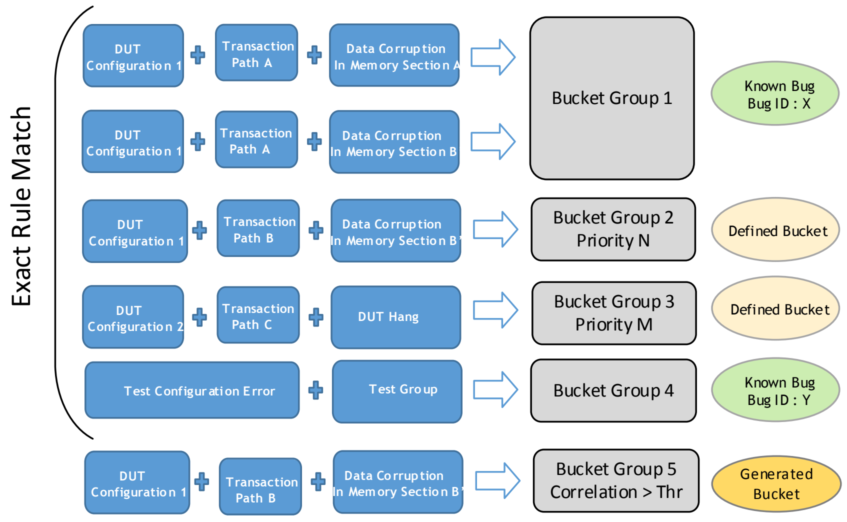

Figure 4. Architecture Model Based Rule Database

The Figure 4 mentioned above shows the rule written by the debug engineers for automatic triaging of failures. More than one rule entry may point to the same bug or bucket.

In addition to well-known failures, debuggers can also write rules for classification of potentially unknown failures. Such bucket classification can be based on DUT configuration, test group type, type of failure etc.

If the coded rule doesn't match and if the threshold is higher than a set number, then model correlates signature with pre assigned weightages to different parameters in signature, comparing with the available rules and groups failures. If the threshold is less than a set number, grouping is based on default parameters or it remains ungrouped. Such ungrouped failures need to be debugged manually and new set of rules is updated in the database and re-bucketed. Figure 5 explains the search, match and update cycle.

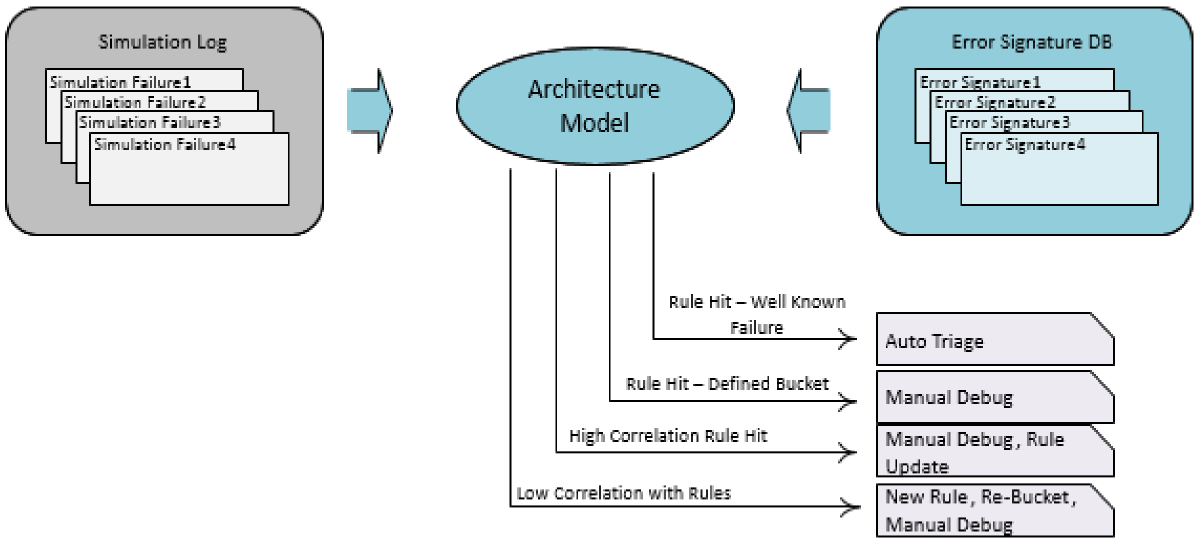

Figure 5. Search, Match and Update Cycle

Architecture model with the rule base acts as the repository of known failures – in other words knowledge repository. This is a well-documented way of knowledge management than the tradition descriptive documentation which teams, especially geographically dispersed teams finds very difficult to corroborate to reach a conclusion. Also the rules are in the database, hence a repeating failure gets highlighted automatically. Rules that are not relevant needs to be pruned by the debug team from time to time.

The bucket prioritization can also be set based on features enabled in the testcase and other meta-data including how long a certain bucket is pending for debug. Such prioritization helps the debug coordinator focus on most important failures first and hence unearth potential RTL bugs much earlier.

IV. Benefits

This tool is an attempt to accomplish two divergent goals. One is to reduce the workload of the debug team. Other is to significantly reduce the time to catch new bugs at the fullchip level. The set of techniques deployed in debugging includes deriving the signature from regression failures, creating rules on these signatures and classifying & automatic triaging based on these signatures has helped in significantly containing the debug workload. This tool has also helped the team to focus on the most important and yet to be debugged issues. Such focused debugging has helped the team to triage a completely new failures in regression within a week's time.

The proposed methodology can be used in both unit level as well as fullchip level. Its application is valued much higher as debug co-ordination and order of complexity is pretty high at the fullchip.

V. CONCLUSION

The set of techniques employed viz., transaction tagging, signature extraction from log files, coding rules based on signatures, correlation of new failures based on old signatures; has helped in reducing debug fatigues significantly for the debug engineers. Reduced intervention in dispositioning of the significant portion of known issues has helped debug engineers to focus on new issues in the regression and potential RTL bugs. Rule sets in the database for classifying failures, served as knowledge repository and helped in leveraging each other's work. Signature database and bug database provides the critical information required for correlating known failures with the debug team.

Some of the key observations prior to deploying the tool and post deployment are captured in the Table I.

TABLE I. CAPTURES SIGNIFICANCE OF SIGNATURE BASED CLASSIFICATION

Regression/Log Reports	Scoreboard & Checker Error Based Classification	Signature Based Classification
Number of Buckets in Typical Weekly Regression	300 to 500	30 to 70
Number of known buckets auto triaged	10 to 15 (approx. 3 %)	10 to 20 (approx. 30%)
Number of buckets debugged every week	150 to 200 (50% to 70%)	Usually 100 %
Life of bug in Full chip	Average of 4 to 5 weeks (at times found on emulation) before getting recorded	Typically 1 week to record new signatures.
Cross GEO co-ordination	Every day stand-up meeting in morning , followed by late night sync-up in 1:1 with counterparts on debug progress	Thrice a week call, less frequent 1:1 sync-up on debug progress

VI. FUTURE WORK

The tool was an attempt to investigate the usefulness of architecture based classification and also a preliminary effort in applying data mining techniques in the fullchip debug. With the success this far, there is a fair degree of confidence that algorithms like Naïve Bayesian Classifiers, decision tree etc. can be applied very effectively for classifying the buckets based on past known failures and derived new group of failures automatically. We hope to continue the work in similar direction and make debugging a pleasurable experience for the ASIC community.

REFERENCES

[1] Debug Limited No More: The Case for Debug Automation Author: Andreas Veneris
http://vote.dvcon.com/sites/default/files/venssa.pdf

[2] Efficient Failure Triage with Automated Debug: a Case Study Author: Sean Safarpour, Evean Qin, and Mustafa Abbas
http://verificationhorizons.verificationacademy.com/volume-7_issue-2/articles/stream/efficient-failure-triage-with-automated-debug-a-case-study_vh-v7-i2.pdf

[3] Advanced Techniques for RTL Debugging Author: Yu-Chin Hsu , Bassam Tabbara , Yirng-An Chen , Furshing Tsai
http://www.cs.cmu.edu/afs/cs/user/yachen/www/papers/verdi-dac03.pdf

[4] From RTL to Silicon: The Case for Automated Debug Author: Andreas Veneris, Brian Keng, Sean Safarpour
http://www.eecg.utoronto.ca/~veneris/10aspdac2.pdf