Regression Automation For Architectural Checker

Siva Kumar Kotamraju

Abstract: The paper deals with the design and development of Regression automation for Architectural Checker. The Automation Script developed automates the running of Chekhov (AV Tool Frame Work) for all Processors and collects the files in desired Paths.

Keywords: Regression Automation, Checker.

I. INTRODUCTION

Pre Silicon Validation Starts before the development and after finishing of the Exploration and Planning of Silicon. Architectural Validation is ensuring that the Programmer visible behavior is Correct and IA-32 Compatible. In Architectural Validation Tool Flow, The feed (A Script) takes a regression list and passes each entry on the list to Exec (A Script). The Exec parsed the entry to create a work area, build and execute a single test. Tests named with the .max extension are assumed to contain macros that need to be expanded before assembling. The expanded assembly file is assembled into object code, typically as a .32.obj file. The object file is passed to RTL and to Archsim as the test input, and each model executes. The Checker is used to compare the States of the two Systems (RTL-Untested & Archsim-baseline) against each other. The Checker State Comparison includes register values and memory values. Chekhov is the next generation Micro Processor or Multi Processor Pre-Silicon Validation Checker that enables Architectural State Comparison of an MP (Multi Processor) or MT (Multi Threaded) RTL simulation with a functional simulator. There is no Single Script which does automation of running the Chekhov for all Processors. Developed such a Script which automates the running of regression test directories for all Processors.

II. WHAT IS CHEKHOV AV?

The Chekhov AV tool is a validation framework for use in validating the operation of Intel processors at the RTL (pre-silicon) stage of development. It can be used to validate processor operation for single threaded, multi-threaded (MT), and multi-processor (MP), and multi-core configurations. Chekhov AV runs on a Linux-based platform and validates processor operation by performing a variety of checks on the outputs of an RTL Simulator and of an Architectural Simulator (such as Archsim or Sphinx). The checks that Chekhov AV performs include the following:

- *Department of CSE, Vignan's Nirula Institute of Technology & Science for Women, Pedapalakaluru,Guntur-522009*
- *siva.kotamraju@gmail.com*

- System Bus Protocol Checking:-Chekhov extracts individual bus transactions from an FSB trace that is derived from the RTL simulation. It then checks these transactions for system bus protocol violations.
- Cache Coherency checking:-Using the bus transaction information, Chekhov creates cache line

ownership data, which it in turn uses to check for MESI or other cache-coherency protocol violations..

- Architectural State Checking:-The architectural state generated by the RTL Simulator and the Architectural Simulator are compared at instruction retirement boundaries to verify correct functional operation.
- Inter-processor Data Consistency Checking:-By correlating architectural instruction ordering with global load and store visibility points, Chekhov determines if inter-processor load and store values are correct.
- Memory Ordering Checking:-By correlating RTL instruction retirement data with global visibility points, Chekhov checks for memory ordering rules violations.

III. CHEKHOV AV OPERATION

The following illustration shows the basic components and checking modules that make up the current version of Chekhov AV. It also shows the flow of data through the tool and the output files that provide the validation data from the various checkers.

Operating modes

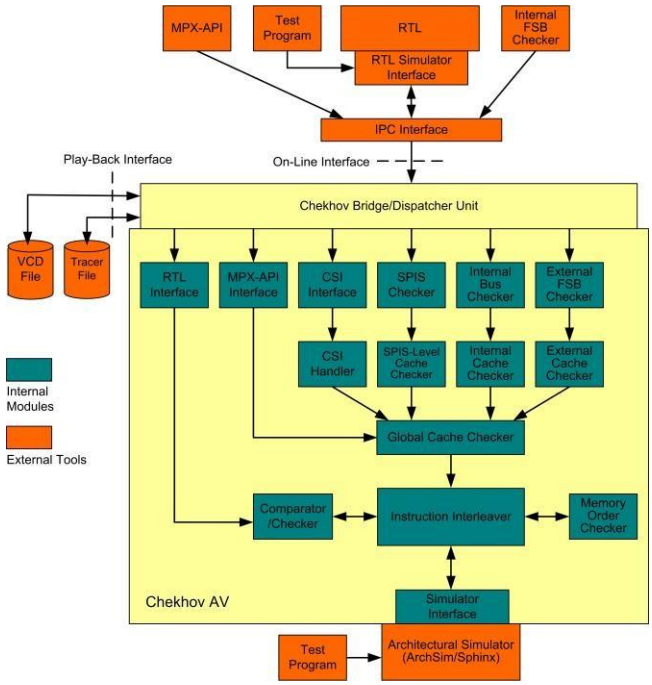
Chekhov AV has two operating modes: on-line and play-back.

In on-line mode, the RTL simulator executes a test program and produces trace data. This trace data is then transmitted directly to Chekhov AV allowing it to perform its validation checks on-the-fly. In play-back mode, Chekhov AV is used as a post processor. Here, the trace data from the RTL simulator is saved in files. This trace data can then be run through Chekhov AV off-line to perform the validation checks. For this task , considering Play-Back mode only.

Test Program

To generate test data for analysis by Chekhov AV, a test program must be executed by the RTL simulator. In on-line mode, this program is launched with the Chekhov invocation command and executed by the RTL simulator. The output of the RTL simulator is then transmitted to the Chekhov Bridge/Dispatcher Unit for analysis. As part of the analysis process, the test program is also stepped through the Architectural Simulator (Archsim), so that the output of the Architectural Simulator can be compared with the architectural state information generated by the RTL

Simulator. In play-back mode, the test program is executed by the RTL simulator and the resulting trace data is saved. At a later time, this trace data can then be run through to Chekhov AV for analysis. During this play-back run, the tests program is stepped through the Architectural Simulator



RTL Simulator

The RTL Simulator allows a test program to be run on the RTL model of a processor under development. The simulator can be configured for single processor, multi-threaded (MT), or multi-processor (MP) execution. For MT and MP simulations, the test program dispatches processes or threads to run on each processor (each bus agent) that is being simulated.

The RTL simulator produces three types of output signals:

- FSB signals—The Intel processor front-side bus signals. Chekhov AV uses these signals to perform system bus protocol and cache coherency checking. These signals also enable Chekhov AV to perform fuzzy checking of memory transactions before the MPX-API is available.
- RTL instruction retirement signals—Signals from the RTL simulator that indicate when instructions have been retired and the architectural state of the processor (register and memory states) at the retirement boundaries. These signals are used for memory order, inter-processor data consistency, and architectural state checking.
- MPX Signals—MPX-API signals that define when loads and stores became globally visible (become GO). The load visibility points (LVs) and store visibility points (SVs) provided by MPX-API allow Chekhov to perform exact memory order and inter-processor data consistency checking. The MPX-API must be available to capture these signals.

These signals are transmitted from the RTL Simulator Interface to the Chekhov Bridge/Dispatcher Unit through an IPC (Inter-processor communications) Interface

Chekhov Bridge/Dispatcher Unit

The Chekhov Bridge/Dispatcher Unit is an interface and translator for the Chekhov AV input signals that it receives from the RTL Simulator Interface or from play-back files. It translates the FSB signals into a VCD-style trace, which it then transmits to the Bus Checker. It translates the RTL instruction retirement signals and MPX signals (if available) into a proprietary Intel data format, which it then transmits to the RTL Interface and the MPX-API Interface, respectively.

The Chekhov Bridge can optionally store the FSB trace data, RTL instruction retirement data, and MPX data off-line in play-back files. In play-back mode, these play-back files can then be read back into Chekhov AV for analysis.

RTL Interface

The RTL Interface takes the RTL instruction retirement data and converts it into a proprietary data format that is used by the Comparator/Checker and the Instruction Interleaver. The data from the RTL Interface is sent to the Comparator/Checker first, and from there it is sent to the Instruction Interleaver

MPX-API Interface

The MPX-API Interface takes the global load and store visibility data (LV and SV) it receives from the Chekhov Bridge/Dispatcher Unit and converts it into a format that can be used by the Instruction Interleaver.

Instruction Interleaver

The Instruction Interleaver forms the heart of Chekhov AV. Its major function is to reorder the RTL instruction retirement data from multiple processors or multiple threads so that the output of the RTL Simulator can be accurately compared with the output of the Architectural Simulator. This reordering of instruction retirement data is performed with two operations:

- Inter-processor instruction interleaving.
- Store synchronization.

When these two operations have been carried out for a group of RTL instructions retirement states, the Instruction Interleaver steps the Architectural Simulator. The resulting instruction retirement data from the Architectural Simulator and the interleaved and synchronized instruction retirement data from the RTL Simulator are then send to the Comparator/Checker for checking. Independently from this RTL vs. functional checking operation, the Instruction Interleaver also checks the RTL instruction retirement data for inter-processor data consistency.

Comparator/Checker

The primary function of the Comparator/Checker is to compare the results of an RTL simulation with an architectural simulation. The checker compares the interleaved RTL instruction retirement data with the Architectural Simulator results to check that RTL results are functionally (architecturally) correct

Inter-Processor Data-Consistency Checking

One of the auxiliary functions of the Instruction Interleaver is to perform inter-processor data-consistency checking on the RTL instruction retirement data. This checking is done by comparing the load, store, and snooping activities of the processors being simulated against a set of data-consistency rules. Specifically, the Instruction Interleaver checks that data from a snooped architectural store goes out on the bus correctly, and it checks that an architectural load is synchronized correctly with a prior architectural

store. If cache ownership windows are being used to determine load and store visibility, fuzzy checking occurs. Here, a load can take data from another processor's store whose visibility window overlaps or comes before the load's visibility window. If discrete global visibility points are being used, a load can take data from a store on another processor only if the LV point comes after the SV point.

Memory Order Checker

The Memory Order Checker (MOC) uses the interleaved RTL instruction retirement stream to check intra-processor memory ordering. Specifically, it checks that all loads and stores from the same processor become visible on the system bus in the correct order. The memory ordering rules used by this checker are those that have been established for the specific processor being simulated.

IV. PROPOSED WORK

There is no Script in Chekhov group which do regression run of test directories of all Processors. The Script does Automation of the Chekhov run for different processors and generating a report. The report generated by the Automation Script gives the description of the work done till end of execution. This report can be taken as basis to eliminate the errors generated for each Processor after Chekhov run, to know the stage at which the script was failed and the stages it was Completed Successfully. This Script also does Clean-up of files and directories on a regular basis. The Procedure for Automation is as Follows

2.3.1 Define Config file

Create a Config file which contains the hard coded build/regress path $path and retrieve the path for future reference. Create a LOG file which is used to log the status of each step either Successful or Unsuccessful. If Unsuccessful, Exit from the Script & Log Contents are generated in the form of mail to user or if Successful, then it logs the message.

2.3.2 Chekhov Cleanup

Check whether any Previous Chekhov Sub-directory is there. If so, the script removes it in-order to get the fresh Chekhov Sub-directory.

2.3.3 Checkout sources

Checkout the Sources of Chekhov. To Checkout the code base, the Procedure is as follows:

1) Setenv CVSROOT /mpg/s1394/chekhov

2) Cvs co Chekhov.

If there are any errors while Checkout Sources due to lack of group permissions… etc, the Control automatically exits from the Program, logged into the Log file the respective messages and Email is generated to the User. The error if any is redirected to the error_chekout.log.

2.3.4 Build for Merom Executable

The next step is the build to get the merom Chekhov Executable. The Procedure to build the Chekhov Executable as Follows:

1) Change the Path to the make directory.

2) Specify the project (processor) name in the Makefile.project file

At present, the following processors are supported. BONNELL, NEHALEM, TEJAS, YONAH, MEROM

Run Do Build Wrapper and the executable and static/shared library of merom Processor will

Created under linux2* directory.

If there is any error while build due to errors in Source repository etc.., simply exit from the Program and mail the log contents to the respective user. The error if any is redirected to error_build.log file. If there are no Errors, simply append the Successful message to the log file and Proceed to next step.

2.3.5 Make file Update

The next Step is Update the make file in make directory to get the Nehalem Executable ready after Rebuild using search and replace technique. Search for a word merom and replace that word with Nehalem.

2.3.6 Rebuild for Nehalem Executable

Run Do Build Script again and the Nehalem executable and static/Shared libraries of Nehalem Processor will create under Linux* directory. If there is any error while build due to errors in Source repository etc.., simply exit from the Program and mail the log contents to the respective user. The error if any is redirected to error_build.log file. If there are no Errors, simply append the Successful message to the log file and Proceed to next step.

2.3.7 Difflog Cleanup

Clearing of Diff logs generated by the Script on a regular basis. For implementing this function the Perl package used is Date::Calc to create today's directory and clear the output directory generated before.

2.3.8 Tar files Cleanup

Clearing of Output tar files of run directories of respective Processors. For implementing this function the Perl package used is Date::Calc to create today's directory and clear the output directory generated before.

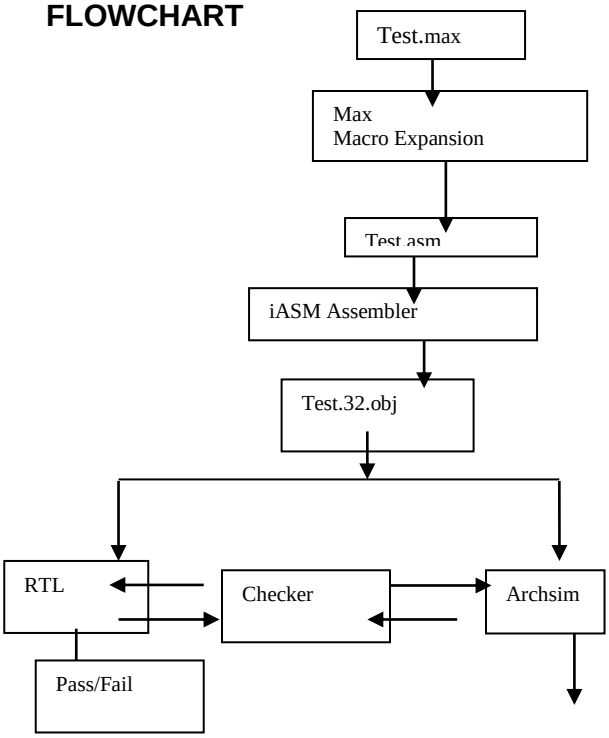
2.3.9 Run Chekhov

Take a loop and run the run-chekhov.pl Script for all Processors in-order to collect the output tar files and difflogs for all test directories and for all processors.

2.3.10 Email generation

Email the Log file Contents to the user using UNIX mutt command.

FLOWCHART



V..RESULT S

YONAH:

Diff Log of Chekhov regressions
==========

WHO : skotamra
WHEN : Wed Dec 6 01:40:45 PST 2006
CURRENT OUTPUT : output_120601_sk_ynh.tar.gz
OLD OUTPUT : output_082110_zh_ynh.tar.gz
FILES CHECKED : run_log.out, chekhov_log.out

MEROM:

No Failed Tests in
/mpg/s1396/CHEKHOV_REGRESSION_SCRIPTS/DIFFLOGS/overall_diff_log_120600_sk_mrm.

Diff Log of Chekhov regressions
==========

WHO : skotamra
WHEN : Wed Dec 6 00:38:01 PST 2006
CURRENT OUTPUT : output_120600_sk_mrm.tar.gz
OLD OUTPUT : output_111515_bj_mrm.tar.gz
FILES CHECKED : run_log.out, chekhov_log.out

TEJAS:

No Failed tests I observerd in
/mpg/s1396/CHEKHOV_REGRESSION_SCRIPTS/DIFFLOGS/overall_diff_log_120600_sk.

Diff Log of Chekhov regressions
==========

WHO : skotamra
WHEN : Wed Dec 6 00:41:01 PST 2006
CURRENT OUTPUT : output_120600_sk.tar.gz
OLD OUTPUT : output_111515_bj.tar.gz
FILES CHECKED : run_log.out, chekhov_log.out

BONNELL:

No Failed tests in
/mpg/s1396/CHEKHOV_REGRESSION_SCRIPTS/DIFFLOGS.

Diff Log of Chekhov regressions
==========

WHO : skotamra
WHEN : Wed Dec 6 20:32:52 PST 2006
CURRENT OUTPUT : output_120620_sk.tar.gz
OLD OUTPUT : output_082111_zh.tar.gz
FILES CHECKED : run_log.out, chekhov_log.out
==========

VI.CONCLUSIONS

Regression Automation, Error Criteria need to be investigated further. The selection of diffed output tar file is considered for comparison of earlier results. In future, the Script is to be modified to include Skip check in the Command line to eliminate user interaction for each Processor regression and the second improvement is required to have different regression Output directory names for bonnell and tejas.

VII. REFERENCES

[1] "When Perl is not quite Fast enough" by Nicholas Clark.
Link: http://www.ccl4.org/~nick/P/Fast_Enough/
[2] "Optimize Perl" by Martin Brown Dated-19oct 2004.
[3] Link::http://www-128.ibm.com/developerworks/library/l-optperl.html?ca=dgr-lnxw01OptPerl
[4] Data Structures and Algorithms with examples in Perl by Jon Jacky.
Link: http://staff.washington.edu/jon/dsa-perl/dsa-perl.html
[5] IA-32 Intel Architecture Software Developers Manual –Volumes 1,2 and 3
[6] Learning Perl 4th Edition O'REILLY, by Randal L Schwartz, Tom Phoenix and brain d foy
[7] Programming Perl 3rd Edition O'REILLY, by Larry Wall, Tom Christiansen, Jon Orwant

GLOSSARY

- AV – Architectural Validation
- DUT –Device under Test
- RTL – Register Transfer Logic
- MESI - Modified/Exclusive/Shared/Invalid Protocol
- IA –Intel Architecture
- CE – Constraint Engine
- API –Application Programming Interface
- XML –Extensible Markup Language
- MUT – Model under Test
- FSB – Front Side Bus
- LV –Load Visibility
- SV–StoreVisibility