saveCHIMP: Application-aware Testbench for Chip Multi-Processors

Arjun Khurana, Dong-hyeon Park, Timothy Wong
EECS 578
University of Michigan
Ann Arbor, MI
khuranaa@umich.edu, dohypark@umich.edu, tphwong@umich.edu

***Abstract*—As computing platforms become more complex, the amount of time and effort spent on verification is increasing at a staggering rate. Despite advanced verification and validation techniques that emerged in recent years, design bugs still manage to escape into tape-out. Thus, there is heavy demand for fast, efficient validation of complex network-on-chip platforms. However, existing test generation techniques rely on random or directed test generation that blindly explore the design space, which is inefficient. To remedy this issue, we developed saveCHIMP, an application-aware testbench that target chip multi-processor platforms. Our technique seek to generate quick and efficient testvectors by incorporating information about the applications that are to be run on the system. The testvectors generated by saveCHIMP help improve the quality of the validation process by focusing on exercising the behaviors that are most important to the system when it is deployed on the field. Overall, saveCHIMP was able to provide tests that matched the target application quite closely, while keeping the test small and efficient.**

Keywords—chip-multi-processors, verification, test generation, characterization, testvectors

I. INTRODUCTION

Modern semiconductor systems are increasing in complexity at an unfathomable rate. In the past few decades, commercial processors went from single cores to chip-multiprocessors, and even network-on-chips. Intel and IBM have worked on developing processors with as many as 80 cores, and the number of cores that are expected to be packed into a single die of silicon is increasing every year. Along with the electrical challenges that come with aggressive transistor scaling, the complexity of these systems are increasing at an alarming rate, making verification of these chips more and more difficult.

Despite various state-of-the-art verification techniques that emerged in recent years, such as hardware emulation and formal verification techniques, it is impossible to eliminate all design bugs before fabrication. With the shrinking time to market, it is of the greatest importance to implement efficient and accelerated tests. Reducing errors as early as possible will reduce the cost of fixing them later. For example, Intel dealt with many repercussions because of transactional synchronization issues in the Haswell processor. When facing tight schedules and aggressive time-to-market, companies need to make a compromise on their verification plans by limiting their testing scope to focus on the areas of the system that are most likely to be exercised by the user. Unfortunately, existing test generation methods do not take into account the software side of the system, as most techniques focus on test generation based on hardware specification.

There is a need to find a way to limit the amount of testing to cases that are actually relevant to everyday operation. Designers can take advantage of common interactions that occur over chip multiprocessors. Therefore, we propose saveCHIMP, an application-aware testbench that target chip multi-processor platforms. The saveCHIMP testbench analyze the target application by extracting common patterns and transactions that are observed in the multithreaded program. Once saveCHIMP analyze the traces, and identifies the key behaviors that are present in the program, saveCHIMP generate testvectors that correspond to those hot patterns of the code. Using these characterizations, our team was able to develop concise, randomized tests that cover the common instances, which drastically reduced the number of instructions executed with similar results to the original programs.

II. RELATED WORKS

Developing more concise and superior techniques for verification of chip multiprocessor systems has been a major focus in both industry and academia. Genesys-Pro [1] is IBM's random test program generator currently used for functional verification. Rambo *et.al.* [2] generates random instruction tests to specifically address memory consistency. Inferno [3] specifically looks at characterizing correct design behavior via traces. As of now, these tests and ideas are still not efficient enough, and are a platform to build upon.

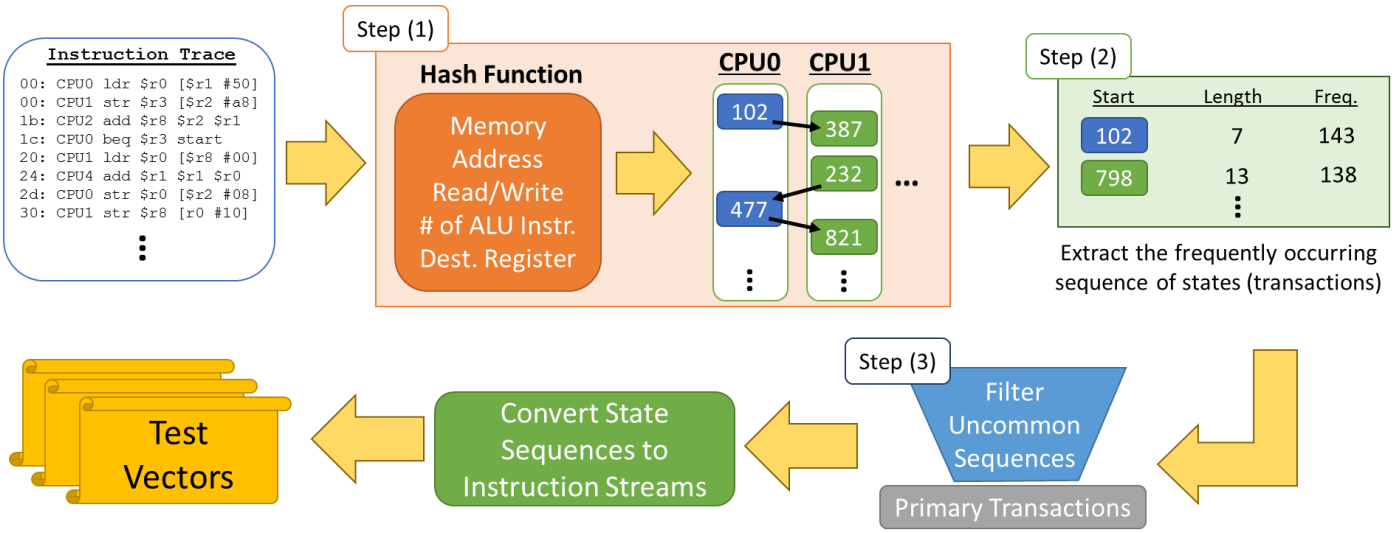

Figure 1: Pipeline of saveCHIMP technique. Step(1): Convert instruction trace into state sequences. Step(2): Breakdown state sequences into transactions. Step(3): Identify the primary transactions. Generate test vectors based on the transactions.

III. METHODOLOGY

A. Overview

The overall execution flow of saveCHIMP is as illustrated in Figure 2. The target application is first executed on a system simulator to generate the instruction traces for the particular application. The generated trace is then sent to the saveCHIMP platform to be modeled into sequences of state transitions that characterize the system behavior that were observed from the application. Next, those state sequences are broken down into subsequences, or "transactions", for identifying key set of behaviors that were observed in the program. Once all the observed transactions are extracted, saveCHIMP filters out any overlapping or non-significant transactions, to create a set of most important behaviors that we want to focus on. Lastly, these key transactions are used to generate test vectors that focus primarily on re-creating these key behaviors. The primary goal of saveCHIMP is to make sure the tests generated are a comparable and efficient representation of the application that was analyzed, and not necessary being able to detect more bugs than the original application. In fact, for our current implementation of saveCHIMP, we try to make sure that the synthesized tests will not only detect the same bugs detected by the original application, but also not trigger bugs that were not detected by the original application.

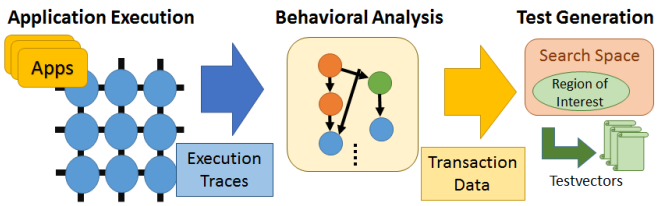

Figure 2: saveCHIMP architecture overview

The detailed procedures of saveCHIMP technique is illustrated in **Error! Reference source not found.**. In Step (1), the instruction traces are converted into states, then passed through a hash function for simplification. In Step (2), the state sequences are broken down into transactions, which are the subsequences, and the frequency of each transaction is recorded. In Step (3), the transactions are filtered out to identify the most important set of transactions that represent the key behaviors of the application. Lastly, the test vectors are generated based on these filtered transactions. The saveCHIMP testbench was developed in C++, and works on traces generated from gem5 simulation.

B. Step (1): State generation

Given a trace of instructions executed by each core of the system, saveCHIMP converts the trace to a stream of states for each core. A state is defined as a read or write memory instruction and the number of ALU instructions that were executed in between the memory operations. Each state consists of: the type of memory operation, the memory address, the data read or written, the primary register address, and the number of ALU instructions that were executed prior to the memory operation.

For efficient indexing of the states and modeling of the execution space of an application, we constructed a custom hash function to convert the state into a single value. Our function simply maps each bits of the value to correspond to each component of the state as shown below:

$$Idx = [MemType]_{9}[ALUCNT]_{8:7}[ADDR]_{6:2}[REG]_{1:0}$$

The hashing allow for quick referencing of the states and simplify our analysis in subsequent steps. We focus our analysis on the memory instructions, and abstract away the ALU operations since majority of bugs in multiprocessors occur due to the different interleaving of memory operations between cores.

C. Step (2): Identification of characteristic behaviors

In Step (2) the state sequences from Step (1) are broken down into subsequences that we define as transactions. First, state sequences of the entire applications are partitioned into windows

of fixed length. This allows our characterization to focus on fixed size states and allow local behaviors to be identified more easily. The most frequently occurring state of each program window are chosen as the boundary states of the transactions. As we go through the state sequence of each window, we separate the sequences anytime the boundary state is observed. To simplify the behavioral model, we only focused on recording the initial state of the transaction, and length of that transaction. Thus, any two transaction that had the same initial state and same length are considered identical transactions. This abstraction allows our model to focus only on the high-level behaviors, and abstract away extraneous paths of a transaction. Every time a new state transition occurs, it is put into the corresponding transaction table, and a counter for that transition is incremented. Subsequent occurrences of the same transition within the same CPU were not put into the table. This will result in a set of unique state transitions, each paired with its number of occurrences. Hence the characteristic patterns were identified (Figure 3**Error! Reference source not found.**).

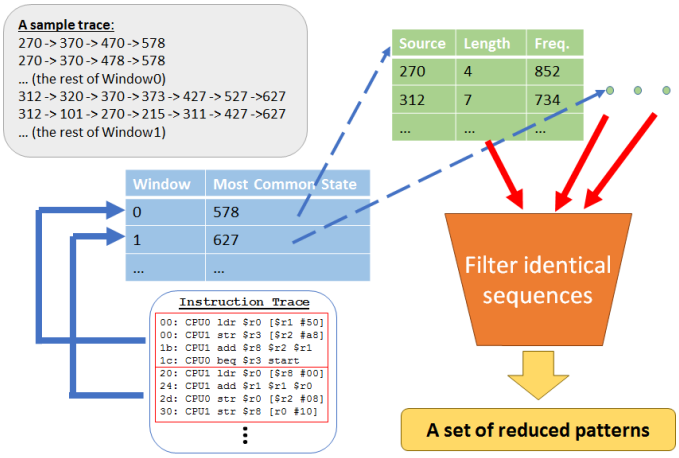

Figure 3: Detailed saveCHIMP technique (Step 3)

It is important to note that the transaction tables established in Step (2) only contained unique state transactions. Subsequent occurrences of the same transactions on the same core were not put in the table. This allowed a minimal set of transactions to be created, without incurring redundancy. We argue that this method would not leave out important state transitions by omitting repeated states, because all transition patterns would at least contain the transition between two states as an atomic unit, so if we could accommodate all the unique states in a set, we had essentially taken into account all the possible transactions on a core.

D. Step (3): Filtering of transactions

In each window of operations within a given CPU, the most common state is determined from the frequency of its occurrence. The source states and lengths of the transitions to the most common state were recorded as separate entries within the same window. Identical patterns in subsequent windows were omitted, hence saving resources used in generating tests on patterns that were already covered in some previous window. Depending on the test plan and extent of testing, the user could specify a fraction of the total number of transitions to test on. This approach helped the test generation method become more streamlined, while giving the user a certain degree of flexibility to decide to what extent the verification was carried out. This method resulted in a reduced set of operations on which tests were generated.

E. Generating tests from the transactions

The test generation process is illustrated in Figure 4. Because the transactions only contain the initial state and the length, we need to convert the transactions back into state sequences. This is done by using state transition table of the application. This is a table that lists the most likely common state transition from one state to another, based on the state sequence of the application. From the state transition table, we can derive the most likely state sequence of a transaction, given the initial state and the length. Once the transaction is converted into state sequences, each individual states are now used to generate assembly instructions. First, the ALU count of the state is used to generate the corresponding number ALU instructions, with randomly chosen source and destination registers. Then, the memory operation is generated to match the type, address and source or destination register of the state. In order to ensure that the address references a data region safe for user access while maintaining the data consistencies, we pass the memory address to a built-in hash function of C++.

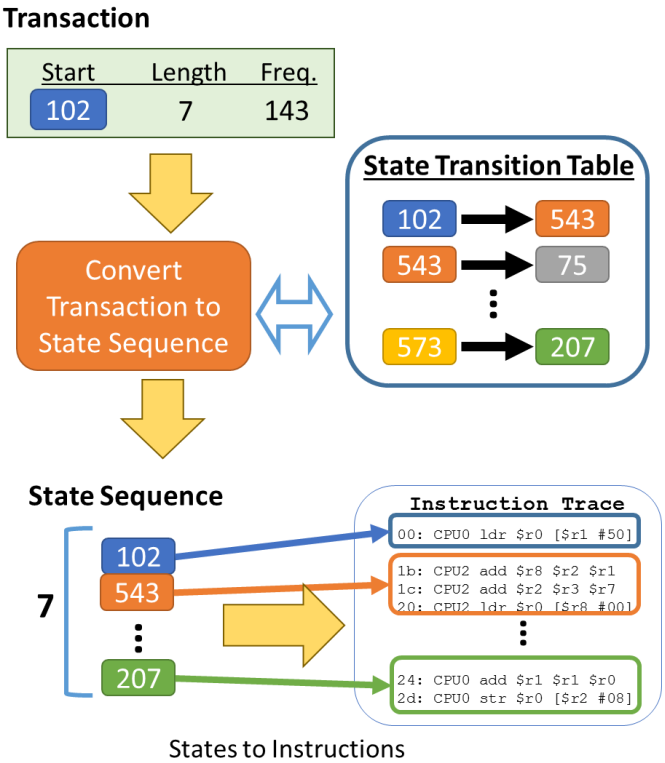

Figure 4: Test Generation Flow: Transaction is converted to state sequence, then the individual states in the sequence is converted into assembly instructions.

IV. Experimental Setup

A. Gem5 Simulation

To analyze the effectiveness of saveCHIMP framework, we applied the technique on a 4x4 mesh network in gem5. The system consists of 16 ARM cores with directory-based MOESI cache coherency. The network interconnect was simulated by the Garnet network simulator, using dimension order routing on a fixed 5-stage virtual channel router. The detailed specification of the system are listed in Table 1. The streams of ARM assembly instructions that were generated by saveCHIMP were injected to each core of the system by having each thread of the testbench program call the assembly function that corresponds to its core ID.

CPU	ARM, TimingSimple
Topology	4x4 Mesh
Coherency Protocol	MOESI CMP Directory
L1 Instruction Cache	4-way 32 KB
L1 Data Cache	4-way 32 KB
L2 Cache	8-way 1024 KB
DRAM	DDR3 1600MHz 4GB
Routing Function	XY-routing
Router Pipeline	Fixed 5-stage

Table 1: Gem5 Simulation System Configuration

B. Testvectors

The baseline program used in our analysis was PARSEC benchmark's Blackscholes application. The original application was executed in a normal gem5 system to collect the instruction traces for saveCHIMP. The application was also executed on a buggy system, and it recorded the bugs that were triggered by the application, along with the simulation time for when the bug was exercised. This bug data was then compared with the bug data collected from executing each of the following four testvectors:

- Random: randomly generated streams of instructions
- Step (1): tests generated from hashing step.
- Step (1) + (2): tests generated from hashing and characterization steps.
- Step (1) + (2) + (3): tests generated from full saveCHIMP procedure.

The "Random" test serve as the baseline in assessing how effective saveCHIMP is in comparison to the traditional technique of random test generation.

Bug Injection

C.

We quantified the performance of our solution through injection of network traffic bugs into the system. In our bug model, we trigger that a bug has been detected if the network interconnect of gem5 observes a certain sequence of traffic being injected into the network (**Figure 5: Bug Injection Example –** Flags a bug when gem5 detects a specific sequence of states. We focus on bugs in the high-level network behavior, because those are the type of bugs that are most difficult to target in validating multi-core systems. In our analysis, injected 10 randomly generated bugs of length three, so that the system will trigger a bug whenever it observes one of the ten sequence of source-destination traffic. Because the goal of saveCHIMP is to be able to recreate the behavior of the target program, the evaluation focused on how closely the detected and undetected bugs from the synthesized tests matched those of the baseline program.

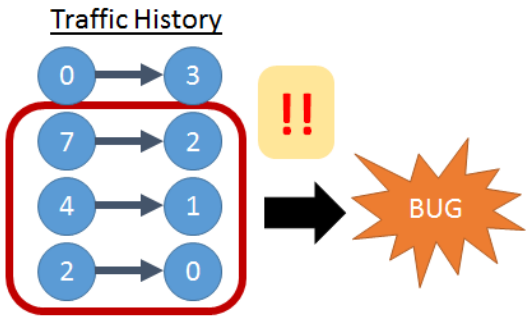

***Figure 5: Bug Injection Example** – Flags a bug when gem5 detects a specific sequence of states*

V. Results

A. Evaluation Metrics

We used the following four statistics to analyze how closely a testvector match the behavior of the baseline application:

$$\text{Sensitivity:} \quad \frac{True\ Positive}{True\ Positive + False\ Negative}$$

$$\text{Specificity:} \quad \frac{True\ Negative}{True\ Negative + False\ Positive}$$

$$\text{Precision:} \quad \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$\text{Accuracy:} \quad \frac{True\ Positive + True\ Negative}{Total\ Samples}$$

The sensitivity is the true positive rate, which quantifies the likelihood of the bug detected by the test program to have also manifested in the baseline application. The specificity is the true negative rate, which measures the likelihood of a bug that was not detected by the test program to have not been triggered by the original program. The precision is the proportion of the bugs that was triggered by the original program that gets identified by the synthetic test. The accuracy is how closely the result of the testvector matches the result from the original application, in both bugs triggered and bugs not triggered.

B. Modeling Results

The sensitivity, specificity, precision, and accuracy of the four different synthetic test generation methods are shown in Figure 1.Figure 6. The test generated from Step (1) that only applied the state generation through hashing had the best sensitivity with 52%, thanks to its large code size detecting a large number of bugs. However, it had lot of false positives which caused other metrics to be low. Each additional step of saveCHIMP was able to improve the accuracy of mirroring the original application and performed better than the randomly generated tests, but at the cost of sensitivity. While the tests were able to

fit the behavior of the baseline program, at the cost of narrowing down the scope of its models. The data shows that the transaction characterization and filtering steps were effective in simplifying the program behavior to few key features, but may have truncated the application too much.

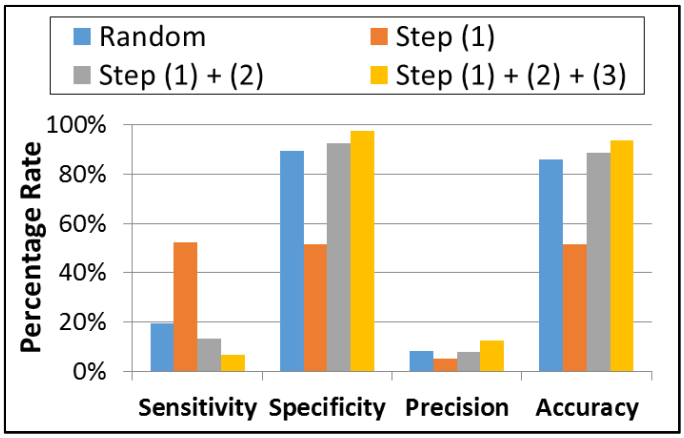

Figure 6: Modeling Results for each Technique. The tests generated at Step (1) has the best sensitivity, but suffers from high false positive rate, which cause the accuracy to be low. While the final testvector of saveCHIMP have low sensitivity, it manages to maintain high true negative rate which contributes to its high accuracy.

C. Overhead Results

The speedup of the bug detection latency from the original application to each of the synthesized tests is shown in 7.Figure 7. The hashing step of saveCHIMP greatly increased the detection latency, but each subsequent steps of saveCHIMP succeeds in achieving the main goals of the characterization and filtering by significantly reducing the latency. We observe a similar trend in the instruction footprint of each technique, as shown in Figure 8.Figure 8. The hashing step dramatically increases the instruction size as expected, but saveCHIMP manages to reduce the size of the testvectors through the characterization and filtering procedures. Overall, the speedup and instruction footprint results show that saveCHIMP is quite efficient in modeling the original application behavior, and performs better than random test generation.

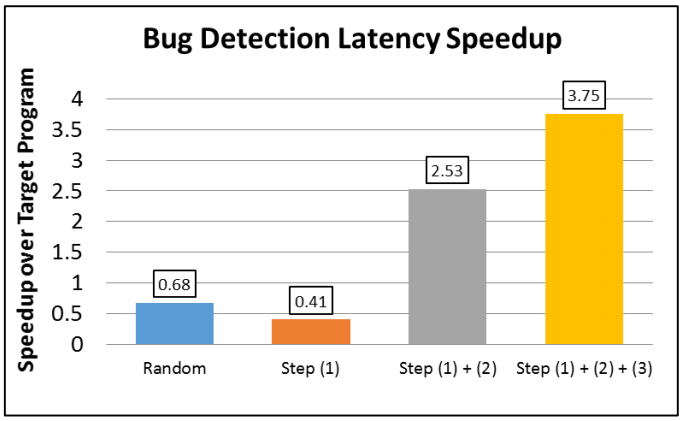

Figure 7: Speedup of Bug Detection Latency for each Technique. The tests from Step (1) has the largest bug detection latency, and performs worse than the original. The final test vector of saveCHIMP is significantly faster at detecting bugs.

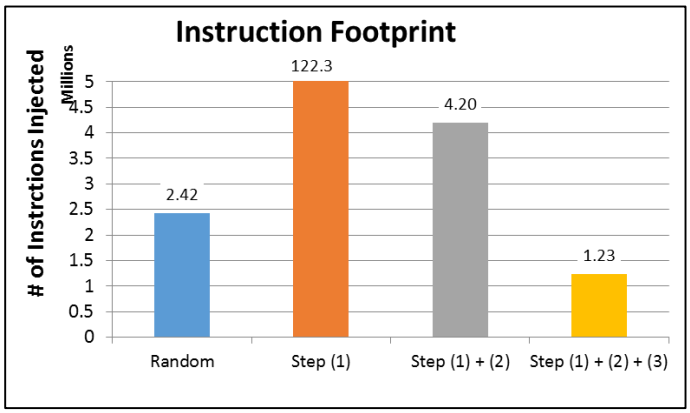

Figure 8: Instruction Footprint of Each Technique. The instruction footprint for Step (1) is the largest with 122.3 million instructions (outside the bound of Y-axis). The final testvector of saveCHIMP manage to reduce the instruction size significantly.

D. Impact of increasing the Filtering Window

The impact of varying the filtering window in Step (3) was explored by comparing the results of changing the threshold from 10% of the window size to 80%, with an increment of 10%. We observed that while there were some minor deviations in instruction footprint generated by the testbench, there were no measurable difference in the performance of the testvectors at different filtering levels. This may be due to the transactions beyond 10% not having significant activity to be triggered by our bug models.

VI. LIMITATIONS

While the results obtained from saveCHIMP showed promising results for a proof-of-concept prototype, there are several limitations to the current algorithm and the experimental evaluation that needs to be address.

First, the implementation does not take advantage of the potential of the hashing algorithm in mapping the states to a bounded search space. The current technique merely treats the hashed values as simple indices for looking up the state from the table. However, the hashing function could be used more effectively by identifying similarities between states at a high level, if the function manages to maintain some locality of its content. This is retained by the bit-partitioning nature of the current function, but the function could be developed further for better mapping of the states to the search space.

The processes of converting the state sequences to transactions, and then from transactions to state sequences discarded many details of the state transition that may have been crucial to the behavior of the program. In addition, because there was no way of maintaining ordering across the cores within the generated assembly files, the instructions most likely would not have been executed in the order that was originally intended. For more accurate assessment of the generated tests, it will be highly

desirable to have a better testing platform where we can control the flow of the assembly instructions that gets fetched by the core.

VII. Future Works

There are several improvement that can be made to the current implementation of saveCHIMP. First, the transaction characterization and filtering algorithms should be enhanced so they can retain the sensitivity level that is observed at the hashing stage of the algorithm. This could be done through fine tuning the filtering parameters, more complex algorithm for extracting transactions, or incorporating larger pool of applications. Second, saveCHIMP could be extended to incorporate multiple application data, so that it will be able to identify overlapping behaviors and testvectors that can cover the space exercised by multiple programs. Third, the current techniques of saveCHIMP only take into account instruction stream of the program. We can incorporate network traces into the extraction and characterization of the behaviors, so memory dependencies can be better modeled by the platform. Lastly, we can incorporate saveCHIMP to a heterogeneous system to further expand the outreach of our solution.

VIII. Conclusion

With the advent of Network-on-Chips and System-on-Chips, the complex computing platforms are growing both in demand and complexity. It is important for companies to release these highly complex products in a timely manner to be competitive on the market, but it is also crucial to ensure that correctness of the design is not compromised. We propose saveCHIMP, a test generation technique that characterizes relevant behavior of a system based on its applications, and generates testvectors that target those behaviors.

The saveCHIMP technique identifies the characteristic execution patterns, focusing on memory operations in particular. The framework takes in instruction traces, extracts the dominant transactions from the traces, and generates a condensed testvector based on those key transactions.

We show that the tests generated by saveCHIMP had significantly less bug detection latency and code footprint compared to traditional randomly generated testvectors. The technique was reasonably effective in efficiently recreating the behaviors of the target system. Some future improvements on saveCHIMP would be to increase sensitivity and precision, incorporate a wider variety of applications, include network traffic in the behavior characterization, and further generalize the technique to accommodate heterogeneous systems.

References

[1] Adir, Allon, et al. "Genesys-pro: Innovations in test program generation for functional processor verification." Design & Test of Computers, IEEE 21.2 (2004): 84-93.

[2] E. Rambo, O. Henschel, and L. dos Santos, “Automatic generation of memory consistency tests for chip multiprocessing,” in Proc. ICECS, 2011.

[3] DeOrio, Andrew, et al. "Inferno: streamlining verification with inferred semantics." Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 28.5 (2009): 728-741.

[4] Foster, Harry D. "Trends in functional verification: a 2014 industry study."*Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015.

[5] Reinders, James. Kinghts Corner: Your Path to Knights Landing [PDF document]. Retrieved from Intel Corporation Website: https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf

[6] Binkert, Nathan, et al. "The gem5 simulator." *ACM SIGARCH Computer Architecture News* 39.2 (2011): 1-7.

[7] Bienia, Christian, and Kai Li. *Benchmarking modern multiprocessors*. USA: Princeton University, 2011.