A Search Based Approach for Stress-Testing Integrated Circuits

Basil Eljuse and Neil Walkinshaw

University of Leicester, Leicester, UK

Abstract. In order to reduce software complexity and be power efficient, hardware platforms are increasingly incorporating functionality that was traditionally administered at a software-level (such as cache management). This functionality is often complex, incorporating multiple processors along with a multitude of design parameters. Such devices can only be reliably tested at a 'system' level, which presents various testing challenges; behaviour is often non-deterministic (from a software perspective), and finding suitable test sets to 'stress' the system adequately is often an inefficient, manual activity that yields fixed test sets that can rarely be reused. In this paper we investigate this problem with respect to ARM's Cache Coherency Interconnect (CCI) Unit. We present an automated search-based testing approach that combines a parameterised test-generation framework with the hill-climbing heuristic to find test sets that maximally 'stress' the CCI by producing much larger numbers of data stall cycles than the corresponding manual test sets.

Keywords: automated search based testing, cache coherency interconnect, system level stress testing

1 Introduction

Integrated Circuits (ICs) are commonly designed in a modular fashion. A developer will combine various subsystems into a comprehensive specification (often in the form of an RTL representation), which is then used to synthesise the hardware component. This can often require the setting of various parameters, with a view to optimising performance. When it comes to testing, this is commonly carried out at a 'system-level', i.e. by formulating and running software applications that are designed to exercise the ICs. Helper-applications can then be used to monitor the state of the underlying ICs, to ensure that its performance is as planned. Conventionally, these test suites are constructed by hand, often with the aim of 'stress-testing' various aspects of the ICs.

Hand-crafted tests are problematic for the conventional reasons that they require a significant amount of time and effort to formulate. In the context of ICs, which are often produced by combining a multitude of components, test cases can rarely be reused because, for a given system, it is often difficult to predict how different software parameters will interact and affect the underlying ICs.

Aside from these somewhat conventional software testing problems, ICs also suffer from significant testability problems. From a software standpoint, their behaviour is often highly non-deterministic, being subject to a large degree of interference from other routine operating-system processes that are difficult to control. In other words, the same test execution can lead to markedly different behaviours.

We highlight this testing challenge in the context of one of the hardware component - the ARM®CoreLink™Cache Coherent Interconnect (CCI). This is a component that seeks to maintain multi-level cache coherency support across multiple processor clusters, ensuring that each 'master' accessing the memory has the most up-to-date view of the data. The CCI provides developers with a host of parameters, and its performance and behaviour is highly affected by other processes run by an operating system (making it highly non-deterministic from a test-application standpoint).

The Search based software testing methodology have been successfully applied in testing embedded systems [1]. In this paper we describe a search-based approach to automatically produce test sets for the CCI that circumvents the above mentioned issues with non-determinism. The specific contribution are as follows:

- A brief description of our initial attempt, based upon conventional Genetic Algorithm-based test generation approaches, which used the search to directly identify potential combinations of memory configurations to stress the CCI.
- A higher-level test-input generation approach that uses hill-climbing, which was developed to circumvent the problems that had arisen with the non-determinism in our initial GA-based approach.
- An experiment that demonstrates that the hill-climbing approach manages to stress the CCI much more effectively than conventional hand-crafted test sets.

The rest of the paper is structured in the following fashion. Section II provides the essential details about one specific hardware cache coherency component, key aspects of cache operations in such a system and the configurability of this component which adds to the testing challenge, which underscores the research motivation. Section III provides the implementation details of the test framework implementing the search based algorithm and its key components. Section IV discuss the results obtained from the application of this approach to a reference platform and the conclusions. The section V outlines the related work that helped shape our approach and finally section VI outlines the future direction of this research.

2 Background

In this section we describe the underlying hardware testing problem. We introduce the typical industrial scenario, where ICs development involves the combination of multiple highly-configurable parameters, but where tests are commonly written by hand. We then cover the technical components of the test environment used in our experiments.

2.1 Motivating scenario

The CCI is a hardware component that provides cache coherency management. It provides multiple levels of configurability specifically at a) design time and b) reset time. As outlined in [2] these configuration options include but not limited to transaction tracker size, QoS (Quality of Service) settings, address width, address striping size, snoop related configurations, and many more.

During system design a key consideration for the designer is to ensure the components are configured for optimal system behaviour. The design time parameters would have to be configured as a one-time setting. However the designer can vary the reset-time configurations during development to arrive at a configuration that produces 'optimal' behaviour – it should be as resource and time-efficient as possible. Configurations are evaluated for desired behaviour with specific test data sets.

Currently the test data design is hand-crafted. They take into account many platform attributes including but not limited to cache line size, page size, the number of memory banks etc. As these attributes do change across different configurations on a single platform (and certainly across multiple platforms), test sets cannot be readily reused.

2.2 The Cache Coherent Interconnect

The Cache Coherent Interconnect (CCI) is an infrastructure component in ARM®based systems that provides both interconnect and cache coherency functionality. It comprises a complex set of base-functionalities [2], and is delivered as a synthesisable Resistor-Transistor Logic (RTL) [3], which enables configurability of its behaviour at various phases during its design and operation.

The CCI product family includes many product versions and in our experiment we focused on CCI-400, which is a specific version of the component. It provides the interconnect and cache coherency functionality across 2 CPU clusters.

The CCI component allows various events to be monitored using its own internal counters. One such event is data stall cycle, which happens for both read and write operations, where a transaction is being stalled by CCI to cope with any issues arising. Such issues could be as a result of CCI's internal transaction tracker queues being full or any type of hazard being detected during operation. Transaction trackers can be full if there are differences in the throughput between components connected to the CCI, as typical of any producer-consumer scenario. Hazards occur in a multi-stage pipelined CPU architecture, when execution of an instruction in the pipeline could result in a wrong operation. Similar hazards are possible during CCI operations which will be handled by CCI by introducing stall cycles during its operation.

TC2 platform The CCI [4] is designed to be deployed as part of a 'platform' – a collection of hardware and software components that are designed to underpin software applications. As such, it is one of the core infrastructure components of ARM's reference TestChip2 (TC2) [5] platform.

TC2 is an ARM development board implementing a big.LITTLE™architecture. This architecture adds a degree of complexity to cache management because it consists of multiple clusters of CPUs, where the size of the L2 cache is based on the cluster composition. This variation in the L2 cache size is resulting from the heterogeneous nature of CPU Clusters - due to difference in both CPU type and in CPU topology - which make the data access operations across clusters different in nature. The TC2 platform has a big cluster made of dual-core Cortex-A15 MPCore™CPU and a little cluster made of tri-core Cortex-A7 MPCore™CPU. It has both level1 (L1) and level2 (L2) caches, with the CCI-400 providing the interconnect and hardware cache coherency across the 2 CPU clusters.

We used a full Android™[1] software stack for TC2 in our experimentation. ARM provides the firmware and full Android software stack supported on this platform via the non-profit organisation called Linaro [6]. This provides a test environment representative of a real system from both hardware and software perspective.

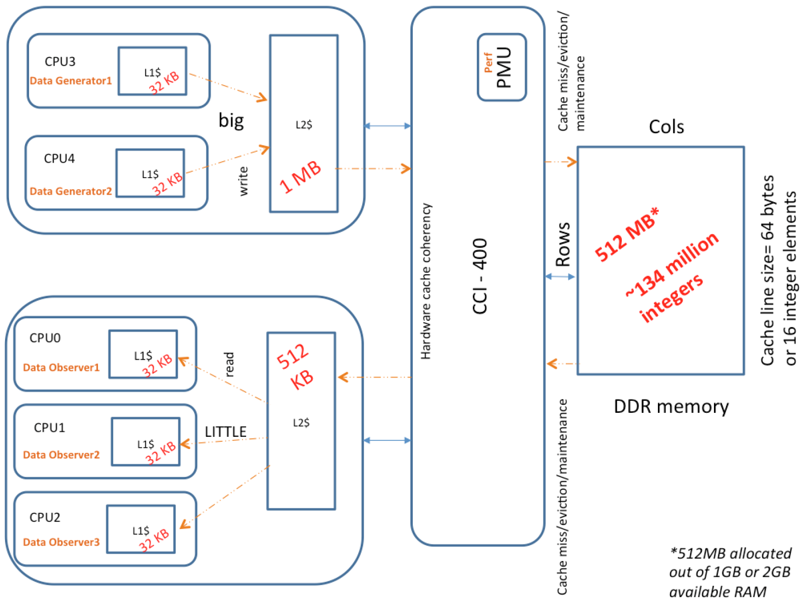

Fig. 1: TC2 platform with CCI-400. *Details the multi-level cache configuration and the various data actors.*

Figure 1 outlines the cache configuration in the TC2 platform and various software components involved during the stress test execution flow at system level. In a TC2 platform all CPUs have 32KB of L1 cache. While the little cluster has a 512KB L2 cache, the big cluster has a larger 1 MB L2 cache. More details of the various software components are explained in later sections.

Testability Challenge The CCI has one feature (shared by most embedded systems) that is particularly challenging from a testing perspective: It is difficult to control externally. Since it is at the bottom of a relatively complex stack of hardware and software components (not least including the Android operating system) it is virtually impossible to reset to a fixed state. With the operating system continually manipulating the

[1] Android is a registered trademark of Google

memory and catering for other routine OS processes, the behaviour of the CCI becomes effectively non-deterministic. Advanced features of CCI (like speculative data fetch) only adds to this level of non-determinism.

Performance Monitoring Unit CCI includes a component called Performance Monitoring Unit (PMU) [2], which has the logic to gather various statistics about the operation of the interconnect at runtime and expose them through counters monitoring certain events. Typically there are multiple counters and different event types that could be monitored using this PMU logic. CCI-400 supports four 32-bit counters allowing one event per counter to be monitored in parallel. This means one can monitor up-to 4 events in parallel without incurring any penalty on accuracy.

The following is a non-exhaustive list of events which are of interest to this research

– Read data stall cycle
– Read request stall cycle due to transaction tracker full
– Write request stall cycle due to transaction tracker full
– Stall cycle because of an address hazard

As a general rule of thumb, the occurrence of stall cycles indicate that the CCI component is having to pause some of its operations to cope with the demand for ensuring coherency. Thus stall cycles can be used as a measure of stress in the system. Larger numbers of stall cycles observed can be an indication of stress in the platform leading to sub-optimal system performance. This can be used as an indication to conclude that the configurations evaluated by the system designer during design phase is sub-optimal.

3 Stress test data generation using Search based technique

In this paper we present a search-based technique that is designed to automatically generate the test sets for configurations of hardware components such as CCI. The aim of the technique is to ‘stress-test’ the component.

3.1 Test Case Representation

Input data Ultimately, to test the CCI, the input consists of requests to write-to and read-from regions within memory. If we adopt a simplistic view of the problem, the testing challenge is to find a set of memory-addresses to write-to and read-from in such a way that the cache is ‘stretched’ – i.e. that we arrive at a situation where requests for data frequently cannot be met by the cache.

One issue with this ‘simplistic’ view of the problem is the lack of determinism in the CCI (as discussed in Section 2.2). A single memory configuration can on one occasion trigger a large number of data stall cycles, and on another run trigger very few. In our preliminary experiments, this severely hampered any search-based approaches we attempted.

As a consequence, we re-analysed the test case representation, with the goal of producing a representation that was more robust in the face of poor testability. Our solution was to step back, and to use our domain knowledge [2] to identify the key high-level factors that could influence the performance of the CCI, and to use these to encode the test cases instead of concrete address spaces. As a result, we now represent our test cases in three dimensions: payload size, sparsity and actor profile. These factors are elaborated below.

Payload size We vary the amount of test data that is read and written. In practice, the size of the data is specified by two integers: x and y – representing the number of columns and rows required to contain the data. We limit this between 1MB and 80MB - i.e. $16 \leq x \leq 5120$ and $16 \leq y \leq 4096$. The number of columns and rows are determined based on the data size of individual data units (current experiment has this as 4 byte length).

Sparsity We need to vary the range of locations that are available to us in memory. The behaviour of the cache will differ if all of the data is to be written and read from a single, contiguous zone of memory, as opposed to a range of non-contiguous, widely dispersed regions. This is based on the principles of locality of reference based on which memory systems work efficiently. Accordingly, we allow for three categories of data 'sparsity': (1) Unconstrained – the payload can be written-to or read-from anywhere in the 512 MB of available memory, (2) relatively sparse – the operational memory is limited to half of the available memory (256 MB). (3) dense – the operational memory is limited to a tenth of the memory (51.2 MB), or (4) very dense – the operational memory is limited to a hundredth of the total memory (5.1 MB).

We chose an upper limit of 512 MB because it is substantially larger than the L2 cache size for the largest cluster (which is 1MB for the big cluster) in our reference platform, which means that with this setting there would invariably be a large number of cache misses which could lead to data stall cycles being generated. In the case where the payload size is high (i.e. 80 MB) and the sparsity is low (i.e. 5MB), this would result in the 80 MBs of data being written-to and read-from the same set of memory locations (with high probability).

Actor profile The number of actors (or processes) writing to and reading from memory can affect cache performance. The affinity of the actors of certain type (read or write) to the clusters were fixed for our experiment - read actors pinned to little cluster and write actors pinned to big cluster. However the number of actors were varied in 3 settings viz, a) single read and write actors, b) dual read and write actors and c) 3 read and 2 write actors. The number of actors performing read and write operations in parallel do affect the level of stress on CCI's operations. Especially with cache sizes being different across clusters and multiple actors operating on memory, the cache miss rates at each of the L1 and L2 caches in the system will vary quite significantly, affecting the number of data stall cycle generated.

Monitoring outputs To monitor the performance of the CCI in response to the test inputs, we use the event counters exposed by PMU (discussed above). Specifically, we use the PMU to record the number of data stall cycles for a given test. To address the problems posed by non-determinism, for each configuration of the above test case representation, we would record the average number of data stall cycle from 100 executions of a test. The decision of fixing the number of iterations to 100 was based on a very conservative approach adopted in our initial experiments.

3.2 The Search Algorithm

Our intuition was to start with a search strategy that is as simple as possible, for which the Hill-Climbing algorithm is an obvious choice. Hill-climbing algorithm starts from a random solution (in our case a random configuration of our test case representation) and navigates through the search space by continuously selecting a better solution from its 'neighbouring' solutions. The hill-climbing stops once the fitness function score is maximised - either finding a local or global maxima - or when a set amount of iterations is attempted.

The choice of hill-climbing is (at least to begin with) justified because it often performs surprisingly well for other search-based testing problems, often even outperforming more sophisticated evolutionary search techniques [7].

```
**Output**: The optimal configuration from the input set
begin
    $maximum = 0$
    $x \leftarrow$ randomInteger(5120)
    $y \leftarrow$ randomInteger(4096)
    $sparsity \leftarrow$ randomInteger(4)
    $actors \leftarrow$ randomInteger(3)
    while fitnessFunction($x, y, sparsity, actors$) $> maximum$ do
        $maximum \leftarrow$ fitnessFunction ($x,y,sparsity,actors$)
        $Neighbours \leftarrow$ generateNeighbours ($x,y,sparsity,actors$)
        forall the $(x', y', sparsity', actors') \in Neighbours$ do
            /* Evaluate fitness of child testdata */
            if fitnessFunction($x', y', sparsity', actors'$) $> maximum$ then
                $x \leftarrow x'$
                $y \leftarrow y'$
                $sparsity \leftarrow sparsity'$
                $actors \leftarrow actors'$
            end
        end
    end
    return $(x, y, sparsity, actors)$
end
```

Algorithm 1: Dynamic Test Data Generation

An overview of the approach is provided in Algorithm 1. The algorithm uses three auxiliary functions:

- $randomInteger(x)$ provides a random integer between 1 and (including) x.

- $fitnessfunction(x, y, sparsity, actors)$ evaluates the generated test cases that conform to the parameters, in terms of recorded data stall cycles. These test cases are run multiple times (we opt for 100 times) to account for any non-determinism, and the average number of data stall cycles is returned.
- $generateNeighbours(x, y, sparsity, actors)$ generates all of the possible configurations that are 'adjacent' to the given configuration. For each parameter a number of new test cases are generated. In case of payload size it generates two new test cases where we increase and decrease the size by a pre-defined step value, whilst keeping the other parameters fixed to their original values. In case of sparsity parameter we generate 3 new test cases and finally for actor profile we generate 2 new test cases. This produces, for a given test set, a total of 7 new configurations.

3.3 Implementation of the Software Test Execution Framework

One of the inherent challenges of testing ICs is that (unlike conventional software testing) there is a distance between the system being tested (in our case the CCI) and the mechanism that is carrying out the testing. Our automated software testing software is executed as a conventional software application in Android 'user-space'. In this subsection we describe how the high-level representation discussed above is mapped into concrete inputs to the CCI.

Test setup As discussed previously, the contents of the memory and cache are routinely affected by many processes within the system that are difficult to control in the context of the test-application. Nonetheless, there are some steps that we carry out at each test to reduce this potential interference:

- We stop as many Android background tasks as is possible (some cannot be stopped).
- We perform an identical memory walk-through sequence between iterations to give better chances for an equivalent initial state.
- We use data barrier instructions - an ARM architecture specific instruction [8] - to ensure all out-of-order data access is cleared before every test data is evaluated.

Inputs

Payload: The payload holds information about the memory addresses from the defined search area, which is represented as a 2 dimensional array. This 2 dimensional array of (x,y) coordinates are populated with random set of memory locations. The payload size control the number of such memory locations that are involved during the test case execution.

Sparsity: The sparsity of the memory locations populated is controlled by limiting the range of values from which the individual (x,y) coordinates can be set. By setting the maximum range from which both the x and y coordinates can be picked during payload generation allows one to control the sparsity. If this maximum range is set as a low value then that would force the payload to be populated from memory address locations which are highly probable to be in closer proximity. On the other hand, if the maximum range is set to the highest possible value, then the payload could be populated with memory locations which are far from each other(non-contiguous).

Actors: The necessity for multiple actors to be reading from and writing to the memory at the same time means that we execute tests as a multi-threaded application. For our tests, an actor can either be a data-generator (which writes data to memory) or data-observer (which reads data from memory). Each actor is assigned (pinned) to its own CPU, where the CPU can belong to one of the big.LITTLE^{TM}clusters.

The hill climbing algorithm relies on this infrastructure while navigating the search space. Once a test data is generated, using the Message Queue IPC (Inter Process Communication) mechanism, the data operation is initiated using actors running in a multi-threaded fashion, while the Linux Perf utility will capture the monitored events as the fitness score. The hill climbing algorithm uses this fitness score associated with each of the candidate test data and arrives at the maxima as per the algorithm 1 outlined in earlier section.

Outputs The outputs (i.e. the number of data stall cycles on the CCI) are read from the CCI's PMU by using the Linux Perf tool[2]. Perf is a user space utility that provides a standard way of accessing performance measurement on supported platforms. Linux Perf can be used to capture both software and hardware events and also other kernel supported events like tracepoint events. Perf also provides the ability to perform counting of events on a per task or per CPU or system wide basis. Perf counters can be configured to operate in either counting or sampling mode. In counting mode the Perf takes the running count of the monitored event which is more accurate, while in case of sampling mode it captures the monitored events at a pre-defined sampling rate, thus providing only an approximate event count. In our experiments we use Perf utility in counting mode and configured to capture system wide events.

4 Evaluation

In order to evaluate our testing technique, we carried out an experiment that was designed to compare the effectiveness of the test-sets generated by our approach against a set of hand-crafted test sets. Specifically, the experiment aims to answer the following research question:

- RQ1 - Does the test data generated with our search-based algorithm yield better results than hand-crafted test data?

4.1 Methodology

To answer the research question, we compare the data stall cycles produced by the test set that was generated by our search-based approach against similar random test sets,

[2] `https://perf.wiki.kernel.org/index.php/Main_Page`

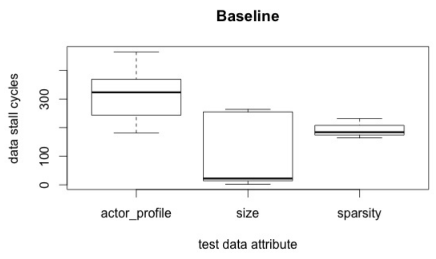

Fig. 2: Baseline scores. *Each payload attribute evaluated in isolation.*

as well as a set of established, hand-crafted test sets, that are conventionally used to test the CCI component.

For the hand-crafted test cases we identified a series of test sets used for functional validation of CCI component within ARM. Some of the existing suite of tests did perform various operations like memory walk-through operations, circular memory copy operations, simultaneous memory operations, accessing random memory locations etc. A key focus for these tests were to check for the functional correctness and that involved performing data integrity checks as part of the test execution. We chose the random memory access operations to characterise our baseline test case in our experiments.

To generate the test sets the payload was populated with random memory location coordinates similar to the hill climbing methodology. We had a fixed configuration for each of the payload attributes - payload size, actor profile and sparsity. In our baseline tests we ran tests where we fixed two of the attributes and varied the remaining final attribute alone.

In case of the payload-size attribute we fixed the actor profile and sparsity attributes and kept varying the size from a minimum (1MB) to a maximum (80MB) value with a predefined step. In case of actor profile we fixed the other two attributes and varied the actor profile against the possible variations of single, dual and multi-actor configurations. Similarly for sparsity attribute we fixed the other 2 and varied the level of sparsity - unconstrained, relative sparse, dense and very dense - by setting the maximum value the x and y coordinates for the memory location addresses being populated.

As explained in earlier section, one of the measures we employed to overcome the issue with non determinism was to repeat each test for 100 iterations. This was consistently applied during the evaluation of baseline test and hill climbing methodology.

Finally, in order to provide a further comparable baseline the same experiment, we produced random a test set by choosing random configurations of the parameters. Here we repeated the same number of iterations as that of the hill climbing experiments but randomly populating all the attributes across generations.

4.2 Results

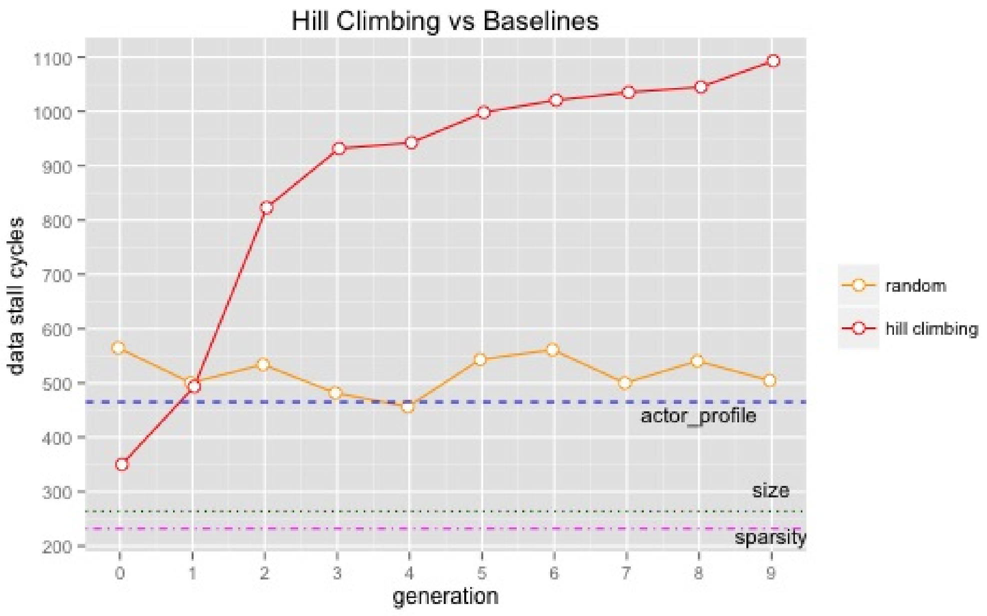

Fig. 3: Hill Climbing scores. *Final scores generated from hill climbing better than any of the baseline scores. Hill climbing scores also performed better than equivalent random test generation experiment*

Baseline test data Figure 2 shows the distributions of scores for our manual baseline test data, considering each attribute in isolation. The plot shows how the three different attributes have varying effects on the number of data stall cycles. No baseline test data generated a score beyond 500 stall cycles. Comparing these we can observe that the actor profile seems to yield the best scores among the baseline candidates. However there is an overlap between the scores generated by each of these factors, which indicates that all of these factors are relevant to be considered in the search based test implementation.

Figure 3 shows the mean scores for our hill-climbing approach, compared with an equivalent random approach, and also alongside the manually generated test scores. The results indicate that the hill-climbing approach rapidly outperforms the other approaches, finishing off at a mean score of over 1000 stall cycles, whereas other approaches fail to improve beyond 600.

4.3 Discussion

The results from hill climbing algorithm captured in Figure 3 show the typical plot where the later generations yielding better results than the initial ones. It starts with scores at a similar level as that of the baseline test data in the experiment and keeps improving over subsequent generations. In the final set of generations the results approach a maximum.

Comparing the scores achieved in hill climbing approach against the baseline it is clear that there is over 2 times improvement in the generated scores. In this experiment hill climbing yields almost 1100 data stall cycles. This is against the maximum value of up to 500 data stall cycles from baseline test data evaluation. The random search approach did not yield any scores better than 600 data stall cycles which is way below from what was achieved using the hill climbing approach.

The results from our experiments clearly shows the suitability of search based software testing techniques in stress testing of platforms with hardware coherency support. By exploiting the PMU counters as a fitness function, a rather straight forward application of search based technique is possible in this testing domain. Additional meta-heuristics in defining next generation nodes and fitness function evaluation could only improve the application of SBST methodology in this area.

We have successfully validated the research question by comparing the results from hill climbing implementation against the baseline test data, which shows a clear benefit with the search based test approach.

4.4 Threats to Validity

Hardware cache coherency components do provide additional hardware mechanisms like PMU that can be exploited by this methodology. In the absence of such hardware support, relevant meta-heuristics need to be defined to ensure the applicability of the proposed methodology. Although here we leveraged the hardware mechanisms to apply this technique, in situations where such mechanisms are not present can limit the easy application of this methodology. A large variation in the measure of data stall cycles could pose a threat to the validity of the results, unless controlled. We observed in our experimentation a large extent of variation in the PMU counter values even when same test data is executed repeatedly. The potential for additional variability can be introduced by the fact that the test framework is also running on the target. In absence of effective measures to reduce the variability of the measured scores, we could get unreliable results which may not necessarily lead to the global maxima. There is a possibility that more needs to be done beyond current measures to reduce the impact of variability in the scores.

We adopted hill climbing as it was a simpler form of search based algorithm to implement and to apply on the target domain. With the known limitations of hill climbing, we could be stuck at a local maxima and miss achieving the global maxima. This could reduce the effectiveness of the approach.

5 Related work

The Search Based testing techniques are used for automated test data generation in various domains including black box testing [9]. The surveys as in [10] indicates that test data generation for non-functional testing of software using SBST techniques were initially focussed on execution time analysis. More work in [11] shows the use of SBST for worst case execution time analysis using multi-objective criteria. The possibility of automated test generation for testing non functional attributes of embedded systems is proven in [12], even though these don't seem to rely on search based techniques.

The application of SBST for non functional testing of systems recently have been wide and varied including performance analysis as explained in [13]. Automated test generation using search based testing techniques are described by [14] for testing worst case interrupt latencies in embedded systems. Further surveys in [15] confirms its application being extended to other non functional system attributes like safety [16], usability, quality of service [17] and security [18]. It has been successfully used in stress testing of real-time systems too as explained in [19].

While most of the above referenced work are focusing on testing software components, application of SBST to hardware component testing can also be seen. We found that the research into the application of SBST techniques with focus on hardware components were mostly in the context of hardware-in-the-loop systems as captured in [20] and [21].

All the prevalent research into cache testing are mostly focused on hardware self testing as explained in [22] and [23]. There are established methodologies for targeted testing of cache memory at very low level as found in [24]. More standard techniques are outlined in the work published in [25]. Successful application of search based software testing methodology in the area of memory system validation can be seen in the work using genetic algorithms with memory consistency model (MCM) verification as per [26].

Similar to the general approach to cache testing some of the L2 cache testing had been also using on-line testing, but looking from a functional testing perspective mostly, as outlined in [27]. Further we could see that in the area of testing cache coherency management, again the focus had been mostly on functional testing as explained in [28]. The work on improving coverage for cache coherency protocol verification can be seen in [29].

However Search Based software testing methodology being adopted for stress testing of L2 cache coherency component from a system perspective is not seen adopted so far. Our research establishes the successful extension of the use of search based software testing techniques for stress testing systems with hardware cache coherency support. Given the fact that this specific application of the technique in this area don't seemed to be attempted earlier, there are no direct comparisons available. Nonetheless this study establishes the benefits of using this technique and provides wider possibilities to make stress testing of systems with complex hardware components more efficient in future with SBST.

6 Conclusions and Future Work

In this study we have investigated the possibility of applying search based software testing techniques in stress testing systems with hardware cache coherency support. Based on the results we have from the selected hardware platform it is evident that the proposed methodology is applicable and more over provides benefits over the traditional approach. We are confident this will improve the adoption of SBST techniques in the related field.

In the current evaluation we focused on using a single objective fitness function targeting a single PMU event. Since the hardware we used supports multiple counters and can monitor up to 4 events in parallel, there is a possibility of extending the methodology to be used in a multi-objective manner. This is an area to be evaluated in future.

Suitability of advanced search based algorithms needs to be evaluated which could potentially avoid the issue of being stuck at a local maxima. However this is a wider problem with any search based algorithm implementation and various strategies are devised in the academic research arena which could be found useful in this specific testing domain. This is another area that needs to be looked into in future.

References

[1] A Arcuri, M Z Iqbal, L Briand, Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing, Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems, ICTSS 2010.

[2] ARM CoreLink CCI-400 Cache Coherent Interconnect - Technical Reference Manual : `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0470k/DDI0470K_cci400_r1p5_trm.pdf`

[3] IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1 - Verilog Register Transfer Level Synthesis.

[4] White paper on big.LITTLE^{TM}processing Technology : `http://www.arm.com/files/pdf/CacheCoherencyWhitepaper_6June2011.pdf`

[5] TestChip2 - part of ARM Versatile Express product family `http://www.arm.com/products/tools/development-boards/versatile-express/index.php`

[6] Linaro - A non-profit organisation working on open source software for ARM based platforms `http://www.linaro.org`

[7] Harman M, McMinn P. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation, International symposium on Software testing and analysis, 2007.

[8] When to use Barrier instructions? `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka14041.html`

[9] M Fischer, R Tonjes, Generating test data for black-box testing using genetic algorithms, IEEE 17th International Conference on Emerging Technologies and Factory Automation, 2012.

[10] P McMinn, Search Based Software Test Data Generation: A Survey, Software Testing Verification and Reliability, Wiley 2004.

[11] U Khan, I Bate, WCET Analysis of Modern Processors Using Multi-Criteria Optimisation, First Symposium on Search Based Software Engineering, 2009.

[12] S Chattopadhyay, P Eles, Z Peng, Automated software testing of memory performance in embedded GPUs, International Conference on Embedded Software, 2014

[13] D Shen, Q Luo, D Poshyvanyk, M Grechanik, Automating performance bottleneck detection using search-based application profiling, International Symposium on Software Testing and Analysis, 2015.

[14] T Yu, W Srisa-an, M Cohen, G Rothermel, SimLatte: A Framework to Support Testing for Worst-Case Interrupt Latencies in Embedded Software, International Conference on Software Testing, Verification and Validation, 2014.

[15] W Afzal, R Torkar, R Feldt, A systematic review of search-based testing for non-functional system properties, Information and Software Technology, 2009.

[16] A Baresel, H Pohlheim, S Sadeghipour, Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms, Genetic and Evolutionary Computation Conference, 2003.

[17] G Canfora, M D Penta, R Esposito, M L Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms, Conference on Genetic and Evolutionary Computation, 2005.

[18] C Grosso, G Antoniol, M D Penta, P Galinier, E Merlo, Improving Network Applications Security: a New Heuristic to Generate Stress Testing Data, Annual Conference on Genetic and Evolutionary Computation, 2005.

[19] L C Briand, Y Labiche, M Shousha, Stress testing real-time systems with genetic algorithms, 7th annual conference on Genetic and evolutionary computation, 2005.

[20] J Wegener, P M Kruse, Search-Based Testing with in-the-loop Systems, First International Symposium on Search Based Software Engineering, 2009

[21] F Lindlar, A Windisch, A Search-Based Approach to Functional Hardware-in-the-Loop Testing, Second International Symposium on Search Based Software Engineering, 2010.

[22] G Theodorou, N Kranitis , A Paschalis, D Gizopoulos, Software-Based Self Test Methodology for On-Line Testing of L1 Caches in Multithreaded Multicore Architectures, IEEE Transactions on Very Large Scale Integration Systems (VLSI), 2013.

[23] G Theodorou, N Kranitis , A Paschalis, D Gizopoulos, Software-Based Self-Test for Small Caches in Microprocessors,IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2014.

[24] Z. Al-Ars, S Hamdioui, G Gaydadjiev, S Vassiliadis, Test Set Development for Cache Memory in Modern Microprocessors, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2008.

[25] S Di Carlo, P Prinetto, A Savino, Software-Based Self-Test of Set-Associative Cache Memories, IEEE Transactions on Computers, 2010.

[26] M Elver, V Nagarajan, McVerSi: A Test Generation Framework for Fast Memory Consistency Verification in Simulation, The 22nd Symposium on High Performance Computer Architecture, 2016.

[27] M Riga, E Sanchez, M S Reorda, On the functional test of L2 caches, IEEE 18th International On-Line Testing Symposium, 2012.

[28] J P Acle, R Cantoro, E Sanchez, M S Reorda, On the functional test of the cache coherency logic in multi-core systems, 6th Latin American Symposium on Circuits and Systems, 2015.

[29] X Qin, P Mishra, Automated Generation of Directed Tests for Transition Coverage in Cache Coherence Protocols, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012.