AGARSoC: Automated Test and Coverage-Model Generation for Verification of Accelerator-Rich SoCs

Biruk Mammo Doowon Lee Harrison Davis Yijun Hou Valeria Bertacco
Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48105
e-mails: {birukw,doowon,hardavis,yijunhou,valeria}@umich.edu

Abstract— SoC design trends show increasing integration of special-purpose, third-party hardware blocks to accelerate diverse types of computation. These accelerator blocks interact with each other in unexpected ways when integrated into a complex, accelerator-rich SoC. In this work we propose a novel solution that guides verification engineers to the high-priority accelerator interaction scenarios during RTL verification. We observe that interaction scenarios frequently exercised by software for the SoC, which is typically developed alongside the RTL, should be the highest priority targets for verification. To this end we analyze the behavior of software executed on high-level simulation models to identify commonly occurring accelerator interaction scenarios. We encapsulate scenarios observed from diverse software executions into an abstract representation that can then be used to extract coverage models and generate test programs. Our experiments show that our solution is able to identify frequently exercised scenarios, extract coverage models, and generate compact, high-quality tests for two completely different SoC designs.

I. Introduction

Driven by stagnating performance gains and energy-efficiency constraints, recent trends in processor design show an increasing adoption of systems-on-chip (SoCs) that integrate several, highly-specialized hardware blocks that accelerate certain types of computation [1, 2, 13]. While there is a significant body of research into designing accelerator-rich SoCs [5, 14], there are only a handful of works that attempt to address the impending verification challenges [4, 7, 12]. Even though the accelerators that go into these complex SoCs are independently verified by their designers, they can still harbor bugs that only manifest when interacting with other accelerators. These bugs escape independent verification because they arise due to incompatible interpretations of specifications and assumptions baked into the different accelerators. Each accelerator can be configured in multiple ways, leading to several different modes of operation, which, coupled with the large number of components in the system, result in a copious amount of interaction scenarios to be considered for verification.

We recognize a need for tools to manage the complexity of verifying the large space of accelerator interactions in accelerator-rich SoCs. We observe that even though the space of possible interactions in an accelerator-rich SoC is large, the actual interactions that are exercised by software are much more limited. It is of paramount importance to get these likely-exercised interactions correct as they affect customers the most.

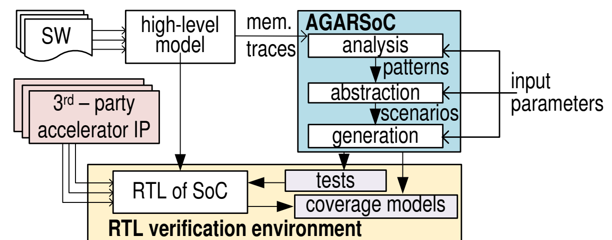

Fig. 1. AGARSoC overview. We analyze the execution of software on a high-level model of an accelerator-rich SoC to identify high-priority accelerator interaction scenarios for verification. We then generate corresponding coverage models and compact test programs, which are used to guide the verification of the RTL for the SoC. AGARSoC is mostly automated and can be flexibly controlled by parameters supplied at runtime.

By analyzing software execution traces from high-level models, we propose to identify the accelerator interaction scenarios that are likely to be executed by software and thus are of highest priority for verification. We then generate coverage models that capture these scenarios and test programs that exercise them. In doing so, we minimize the guess-work and randomness in verification planning, coverage model development, and test generation. Our proposed solution, illustrated in Figure 1, leverages HW/SW co-verification environments where software to be shipped with the SoC is developed and tested on high-level simulation models, concurrently to RTL development. By using our generated test programs, the integration team can significantly cut down the amount of effort and simulation cycles spent on RTL verification. In addition, our automated tools simplify adjusting verification goals as new software becomes available during the verification lifetime. Our solution is designed to guide the verification of accelerator interactions and hence is complementary to approaches that target other goals.

Contributions: In this work, we propose **AGARSoC**: **A**utomated Test and Coverage-Model **G**eneration for Verification of **A**ccelerator-**R**ich **S**ystems-**o**n-**C**hip. AGARSoC is a methodology and set of tools that guide the verification of accelerator-rich SoCs towards high-priority accelerator interaction scenarios. These scenarios are automatically identified by analyzing software executions on high-level models of the SoC under verification. AGARSoC extracts and generates coverage models and high-quality test programs that hit the coverage goals. The generated programs are much more compact than the original software suite they are derived from and hence require fewer simulation cycles. In addition, AGARSoC is designed to be highly-versatile for analyzing test runs, adapting verification goals, and generating test programs throughout the verification lifetime, with only minimal engineer input.

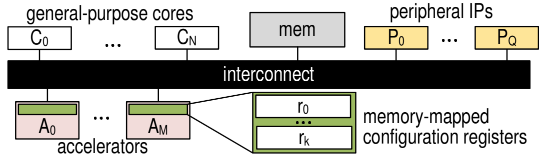

Fig. 2. Accelerator-rich SoC architecture. Multiple cores, accelerators, memories, and peripheral components are integrated in a chip through an interconnect. These components interact via memory operations, which are routed by the interconnect to the proper destinations. Memory-mapped registers in the accelerators are used to configure accelerators for execution.

II. Capturing software behavior

Accelerator-rich SoC architectures integrate several accelerators, general-purpose cores, and peripheral components as illustrated in Figure 2. Each accelerator in the system is designed to execute a specialized task quickly and efficiently. Accelerators for cryptographic hashing, video encoding/decoding, and image processing are already prevalent in current SoCs. Software executing on the general-purpose cores launch tasks on the accelerators by writing task parameters to the memory-mapped accelerator configuration registers. A launched accelerator performs its task, possibly operating on a chunk of data in memory. Upon completion of its task, an accelerator notifies the software, often by writing its status to a memory location specified during configuration.

To capture software behavior, AGARSoC analyzes memory operation traces collected from software running on a high-level model. These traces include regular read and write accesses to memory from processor cores and accelerators, write operations to accelerator configuration registers, and special atomic memory operations for synchronization. Each access in a memory trace includes the cycle, memory address, type, size, and data for the access. The individuality of each accelerator and the existence of multiple memory transactions for configuring and executing an accelerator are features unique to accelerator-rich SoCs. Previous works on analyzing software behavior focus on capturing performance and target processor cores alone [8, 9].

A. Analyzing memory traces

We scan the memory traces from the processor cores to detect patterns of memory accesses to accelerator configuration registers, which we refer to as *accelerator configuration patterns*. We also extract *accelerator access patterns* from accelerator memory traces and match them with their triggering configuration patterns to get *accelerator execution patterns*. We group these accelerator execution patterns into *execution classes* based on their configured modes of operation.

AGARSoC's analysis is a mostly automated procedure with minimal engineer input. Engineers specify the memory mappings for the different components, which is obtained from the SoC configuration. We expect accelerator designers to provide descriptions of start and end conditions for accelerator configuration and access patterns, as well as specify which configuration writes define a mode of operation for their accelerator. Integration engineers can then encapsulate these conditions as Python functions and easily pass them as arguments to AGARSoC's analysis tools. Below are examples of designer-specified conditions for the multi-writers, multi-readers (MWMR) protocol in the Soclib framework [15], encapsulated in the Python functions shown in Figure 3:

- *"An accelerator signals task completion by writing a non-zero value to a memory location written to configuration register 0x4c"*.
- *"Writes to configuration registers 0x44, 0x48, 0x60, 0x50, and 0x5c define mode of operation."*

```
def MWMREndDetector(configPat, accOp):
    endLoc = configPat.accesses[0x4c].data
    if accOp.addr != endLoc or not accOp.isWrite():
        return False
    return accOp.data != 0

def MWMRConfigHash(configPat):
    mode = [configPat.target]
    for reg,acc in configPat.accesses.iteritems():
        if reg in [0x44, 0x48, 0x60, 0x50, 0x5c]:
            mode.append(acc.data)
    return tuple(mode)
```

Fig. 3. User-defined functions. `MWMREndDetector` returns `True` if a particular accelerator memory operation is signaling the end of a task. `MWMRConfigHash` returns a hashable tuple that contains the mode of execution. The `configPat` (accelerator configuration pattern) and `accOp` (accelerator memory operation) data structures are provided by AGARSoC.

B. Identifying interaction scenarios

We identified two types of interaction that are likely to lead to scenarios that were not observed during independent verification. Firstly, concurrently executing accelerators interact indirectly through shared resources. Any conflicting assumptions about shared resource usage manifest during concurrent executions. Secondly, the state of the system after one accelerator execution affects subsequent accelerator executions. We develop heuristics that identify concurrent and ordered accelerator interaction scenarios by analyzing the observed execution patterns. We detect and retain only execution classes in our scenarios, instead of the actual execution instances. Thus, our heuristics avoid capturing redundant scenarios that do not belong to a diverse set of interactions.

Concurrent accelerator interaction scenarios gather accelerator execution classes whose execution instances have been observed to overlap. The overlap is detected by comparing the start and end times of the accelerators' execution patterns. While start and end times observed from a high-level model may not be exactly identical to those observed in RTL, they represent **the absence of ordering constraints** and therefore indicate a possibility of concurrent execution on the SoC. Concurrently executing accelerators interact indirectly through shared resources (interconnect, memory, *etc.*) leading to behaviors that may not have been observed during the independent verification of the accelerators.

Ordered accelerator interaction scenarios are either intended by the programmer (program ordered, and synchronized) or coincidentally observed in a particular execution. We infer three types of ordered accelerator executions:

Program-ordered: non-overlapping accelerator execution classes invoked in program order from the same thread, detected by comparing start and end times of accelerator executions invoked from the same thread.

Synchronized: accelerator execution classes whose instances are invoked from multiple threads, but are synchronized by a synchronization mechanism. We detect these by first identifying synchronization scopes (critical sections) from core traces

and grouping accelerator invocations that belong in scopes that are protected by the same synchronization primitive. For lock-based synchronization, for instance, we scan the memory traces to identify accelerator invocations that lie between lock-acquire and lock-release operations. We group the accelerator execution classes for these invocations by the lock variables used, giving us groups of synchronized classes.

Observed pairs: pairs of accelerator execution classes whose execution instances were observed not to overlap. Unlike program-ordered scenarios, which can be of any length, observed pairs are only between two execution classes.

C. Abstract representation

Following the analysis and scenario detection steps, AGARSoC creates a representation of software execution that abstracts away execution-specific information irrelevant to our goal, such as specific memory locations and actual contents of data manipulated by the accelerators. Our representation contains three major categories: accelerator execution classes, concurrent scenarios, and happens-before (ordered execution) scenarios. Each entry in these categories contains a unique hash for the entry and a count of how many times it was observed in the analyzed execution. The entries for accelerator execution classes and concurrent scenarios are directly obtained from the analysis. The happens-before category contains combined entries for all unique observed pairs and program-ordered sequences. In addition, we generate all possible non-deterministic interleavings of synchronized accelerator executions. The entries in the happens-before sections have counts of how many times they were observed in the execution traces. For any generated ordering that has not been observed, we store a count of 0.

Figure 4 illustrates an example of how a software execution is turned into an abstract representation. The SoC in this example has 3 cores and 3 accelerators. The software threads running on these cores trigger multiple accelerator executions, labeled **1** – **6** in the figure. A circle, a diamond, and a triangle are used to represent invocations of A_0, A_1, and A_2, respectively. The different colors indicate different modes of operation. AGARSoC will identify 5 different classes of accelerator execution corresponding to the different accelerators and modes observed during execution. Executions **1** and **4**, **3** and **5**, and **3** and **6** overlap, resulting in the three concurrent scenarios in the abstract representation. Similarly, AGARSoC identifies two sequences of program-ordered execution classes. Lock-acquire and lock-release operations using the same lock variable, indicated by locked and unlocked padlocks, respectively, mark one group of synchronized accelerator executions. AGARSoC generates all 6 possible interleavings of sequence length 2 for the accelerator execution classes in this group. The unobserved interleavings are marked with a count of 0. The immediate observed pairs extracted from the execution are not shown in the Figure because they have all been included in the program-ordered and synchronized interleavings groups shown. Note that the observed pair resulting from executions **4** and **2** is also a possible interleaving of **2** and **5**.

Abstract representations for different software executions can be combined to create a union of their scenarios by simply summing up the total number of occurrences for each unique

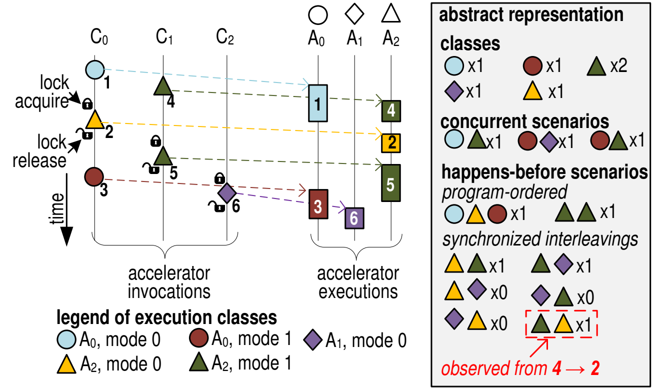

Fig. 4. Abstracting an execution. Accelerator tasks are launched from the general-purpose cores. We group these task executions into unique execution classes, represented by shape and color. We represent concurrently executing execution classes and ordered execution classes. The ordering among classes is obtained from program-order, generated interleavings of synchronized invocations, and observed immediate ordered pairs.

scenario observed across the multiple executions. Through the process of combining, redundant accelerator interactions that are exhibited across multiple programs are abstracted into a compact set of interaction scenarios. This is one of the key features that allows AGARSoC to generate compact test programs that exhibit the interaction scenarios observed across several real programs.

III. Coverage model generation and analysis

AGARSoC generates a coverage model that guides the RTL verification effort towards high-priority accelerator interactions as identified from software execution analysis. The coverage model generation algorithm performs a direct translation of the abstract representation extracted from the analysis: scenarios are translated into coverage events and scenario categories are translated into event groups. The coverage events in each event group can be sorted by the count associated with each scenario entry in the abstract representation. A higher count indicates a more likely scenario.

In addition to coverage model generation, AGARSoC's coverage engine is capable of evaluating and reporting the functional coverage data from RTL simulation. To take advantage of this capability, the RTL verification environment should output traces of memory operations. We believe this is a feature that most real-world RTL verification environments have, since memory traces are valuable during RTL debugging. AGARSoC's coverage engine is designed to be very flexible to adapt to changing verification requirements. Coverage models that AGARSoC generates can be merged without the need to analyze executions again. In addition, coverage output files are designed such that a coverage analysis output for a particular execution can be used as a coverage model for another execution. These features give engineers the flexibility to incorporate new software behaviors at any stage of verification, to compare the similarity between different test programs, and to incrementally refine verification goals.

Figure 5 shows a snippet of a coverage report generated by AGARSoC. This report shows the coverage analysis for an AGARSoC generated program versus a coverage model ex-

Classes

Name	Count	Goal	Seen	% of goal
A0.1	7	29		24.1
A1.1	6	24		25.0
A2.1	6	25		24.0
A2.2	5	18		27.8
A1.2	4	4		100.0
A2.2	4	4		100.0
A0.1	3	4		75.0
A1.1	3	7		42.9

Concurrent

Name	Count	Goal	Seen	% of goal
A0.1 A1.1 A2.1	2	2		100.0
A0.1 A1.1 A2.2	2	6		33.3

Happens-Before

Name	Count	Goal	Seen	% of goal
A1.2 A2.2	2	2		100.0
A1.1 A1.2	1	3		33.3
A0.1 A0.1	1	4		25.0
A2.1 A0.1	0	0		0.0

Fig. 5. Sample coverage analysis output.

tracted from a software suite. The *Goal* column shows the counts of each coverage event as observed in the original software suite. AGARSoC currently uses these goals only as indicators of how frequently scenarios are observed, which can guide the choice of scenarios to include for test generation. The *Count* column shows how many times the events were observed in the generated test program. The *Seen* column indicates the presence of an event occurrence with a green color and its absence with red. The *% of goal* column indicates how many times the synthetic events were observed compared to the events in the original software suite. This column is ignored when all that is required is at least one hit of a coverage event. However, it becomes relevant when comparing the similarity between different test programs.

IV. Test generation

In addition to coverage models, AGARSoC also generates test programs that exhibit the accelerator interaction scenarios captured in its abstract representation. The test generation engine is highly flexible, allowing for the generation of test programs that target chosen scenarios. The number of threads in the generated test program, the length of ordered sequences to reproduce, and the range of scenarios to generate (from a given scenario group, based on frequency of occurrence) are some of the configurable parameters that control AGARSoC's creation of a multi-threaded schedule of accelerator executions to be executed by the generated test program.

A. Generated test program structure

AGARSoC generates multi-threaded C programs that invoke accelerator executions through calls to library functions. Using barriers, the executions of these threads are divided into phases where the accelerator executions invoked in each phase can execute concurrently and accelerator executions in multiple phases are guaranteed to be ordered. For each execution, AGARSoC generates the definition of accelerator execution classes, a data layout, and queues of accelerator execution classes for each thread to execute. At the beginning of each phase, each thread in the phase-based execution mechanism pops an entry from its queue and invokes an accelerator, based on the generated data layout and execution class definitions.

B. Schedule generation algorithm

AGARSoC uses multiple heuristics to generate a compact schedule of accelerator executions that exhibits the desired accelerator scenarios. Our first step adapts the concurrent scenarios to an N-threaded environment, where N is specified by the engineer. If there are any concurrent scenarios with more than N concurrent executions, we enumerate all possible N-long sets of concurrent executions. The outcome of the first step is a list of unique scenarios with N or less concurrently executing accelerator execution classes. Our second step creates a list of ordered unique, execution class groups of size M or less. M is also specified by the engineer.

Our third step uses a heuristic that creates a multi-phase schedule of the concurrent scenarios that also satisfies the most number of ordered scenarios. Note that all invocations of a concurrent scenario are placed in a single phase, to be triggered by multiple threads. All accelerator executions invoked in a phase must complete before the threads proceed to the next phase. Therefore all accelerator executions in the earlier phase are ordered before all the accelerator executions in the subsequent phase, satisfying multiple ordered scenarios. Our heuristic creates partial schedules of length M until all concurrent scenarios are scheduled. At each step, it searches the length M schedule that satisfies the largest number of previously unsatisfied ordered scenarios. We perform this schedule search multiple times, starting from a different concurrent scenario each time. We finally pick the complete schedule that satisfies the most number of ordered scenarios.

Our last step constructs a single sequence of program-ordered executions from the remaining ordered scenarios. Using partial pattern matching, we ensure that the sequence we create has no repetitions and has the shortest length possible.

V. Experimental results

We tested AGARSoC's capabilities on two different SoC platforms. We used Soclib [15] to create a **Soclib ARM platform in SystemC** that has 3 ARMv6k cores with caches disabled, 3 accelerators, 3 memory modules, and 1 peripheral, all connected via a bus. The three accelerators are a quadrature phase shift keying (QPSK) modulator, a QPSK demodulator, and a finite impulse response (FIR) filter. All accelerators communicate with software running on the cores using a common communication middleware. Multi-threaded synchronization is implemented using locks, which are acquired by code patterns that utilize ARM's atomic pair of load-locked and store-conditional instructions (LDREX and STREX). Using Xilinx Vivado® Design Suite [3], we created a **MicroBlaze**

TABLE I
Software suites

suite group	# of progs	description
Soclib 1	9	sequential accesses with locks
Soclib 2	9	sequential accesses without locks
Soclib 3	9	concurrent accesses with locks
Soclib 4	9	concurrent accesses without locks
Soclib 5	18	combinations of the four above
μBlaze 1	7	no synchronization
μBlaze 2	8	lock, single-accelerator invocation
μBlaze 3	5	lock, multiple-accelerator invocation
μBlaze 4	7	barrier, redundant computations
μBlaze 5	13	semaphore

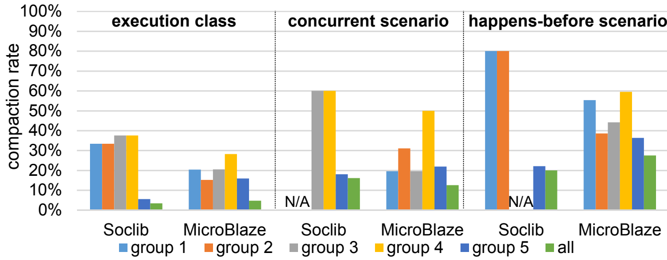

Fig. 6. Compaction rates from abstract representation. The compaction of accelerator executions into unique instances of accelerator execution classes, concurrent scenarios, and happens-before scenarios for our two software suites. The compaction rate is measured as a ratio of the number of unique instances versus the total number of instances.

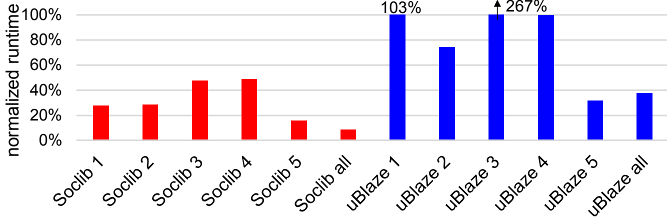

Fig. 7. Normalized simulation runtime for each test group. While generated tests are usually shorter than the original test suites, MicroBlaze suites 1 and 3 show longer runtime, because of the inability of generating minimal schedules. However, the test program generated from all MicroBlaze suites does not exhibit this property.

platform in RTL that comprises three MicroBlaze processors with caches and local memories, six accelerator IP blocks, one shared memory module, and peripheral devices connected via AXI interconnects. The six IP blocks are a FIR filter, a cascaded integrator-comb (CIC) filter, a CORDIC module, a convolutional encoder, a fast Fourier transfer (FFT) module, and a complex multiplier. Synchronization primitives (*i.e.*, lock, barrier, semaphore) are implemented using a mutex IP.

For each platform, we created software suites that contain several patterns of accelerator execution, as summarized in Table I. We designed our software suites to have a diverse set of accelerator usage patterns. After analyzing all of the Soclib software suites, AGARSoC identified 264 accelerator invocations, 130 observed concurrent scenarios, 315 observed happens-before scenarios. Similarly for MicroBlaze, it identified a total 358 accelerator invocations, 592 observed concurrent scenarios, and 502 observed happens-before scenarios. Figure 6 reports average compaction rates from AGARSoC's abstraction process of these observed scenarios. Better compaction rates are achieved for test suites that exhibit redundant interactions. We report the compaction rates for each suite group and the three categories in the abstract representation.

Note that by automatically identifying important scenarios, AGARSoC focuses verification to a small, albeit important, portion of the large space of possible interactions. For instance, for the 17 unique classes identified from the MicroBlaze suites, there can potentially be 680 concurrent scenarios. AGARSoC identified only 74 cases. The potential number of unique classes is much larger than 17 and it is only due to AGARSoC's analysis that we were able to narrow it down to 17. If we conservatively assume that the SoC allows 100 unique classes for instance, 161,700 concurrent scenarios are possible.

We observed that the test programs generated from the compact abstract representation exercise a 100% of the scenarios captured in the abstract representations. Figure 7 summarizes the reduction in simulation cycles achieved by running these tests instead of the original software suites. The AGARSoC generated tests simulate in only 8.6% of the time taken by the complete software suite for the Soclib platform and 38% for the MicroBlaze platform. However, the tests generated independently from the MicroBlaze software suites 1, 3, and 4 offer no execution time reduction, mainly due to the inability of the test generator to create minimal schedules to satisfy all scenarios. Suite 3 contains a program that is intentionally designed to exhibit accelerator patterns that push AGARSoC's schedule generator to the limit. Also, generated tests potentially exercise more happens-before scenarios than the original suites, due to all unobserved interleavings generated for synchronized accesses. Note that we are able to generate a more efficient schedule when the scenarios observed in all test suites are combined with other suites because we can get more compaction from scenarios that repeat across multiple programs. Compared to the MicroBlaze suites, the Soclib suites execute more routines that are not related to accelerator execution and hence are not reproduced in the generated tests.

AGARSoC detects synchronized accelerator executions and generates all possible interleavings as discussed in Section II-C. During execution, only a handful of these possible interleavings are observed. We can instruct AGARSoC's test generator to generate tests to exercise the unobserved interleavings. We evaluated this capability by generating separate tests for the unobserved scenarios in the MicroBlaze suite groups 2 and 3. Our generated tests successfully exhibited all of the interleavings that were yet to be observed.

VI. Discussion

A. Flexibility

AGARSoC provides a flexible set of tools that can support the demands of different design efforts. Capturing design-specific knowledge, as discussed in Section II-A, is simple and easily portable to any verification effort. We were able to adapt AGARSoC to two quite different SoC platforms with minimal effort. In addition, AGARSoC can analyze and incorporate new software into the coverage model as it is developed, and generate tests that hit different coverage goals.

B. Enhancements

Even though we have not fully implemented them yet, we have several enhancements planned. Firstly, in addition to capturing concurrency and ordering among accelerator executions, we can also track their data sharing patterns to identify producer-consumer or data race scenarios. These scenarios can be easily incorporated in the abstract representation as extra scenario groups and can be specified as constraints for AGARSoC's test generation algorithm. These data sharing scenarios, together with a straight-forward enhancement to detect cases where accelerators can invoke other accelerators, can enable AGARSoC to support several different accelerator-rich SoC design approaches. For instance, we can support scenarios where a task is executed by a pipeline of multiple accelerators, without the involvement of general-purpose cores in between. Secondly, we can randomize some parameters of the

high-level simulation model to model non-deterministic execution of software. This allows us to generate multiple, legal memory traces per program increasing the richness of our software suite and further refining the accuracy of our analysis. Thirdly, we can implement mechanisms to detect other multi-threaded, synchronized accelerator executions, in addition to the lock-based and mutex-IP-based mechanism we currently support. Finally, AGARSoC's test generator can be enhanced, with minimal effort, to fully utilize the parallelism available in the SoC-under-verification and also in the verification environment. The schedules we generate to target happens-before scenarios can be multi-threaded. In addition, we can launch multiple concurrent accelerator executions from a single thread. We can also generate multiple, execution-time balanced tests that hit distinct coverage goals to be simulated in parallel.

C. Accuracy

AGARSoC relies on several heuristics to abstract executions into scenarios. These heuristics rely on timing information from a high-level simulation model, which does not necessarily reflect the timing that will be observed during RTL simulation. However, note that we utilize the timing information from the high-level model only to infer accelerator executions that *are likely to run concurrently or are likely to be ordered*. As such, we believe that the accuracy of our heuristics are sufficient enough to guide verification engineers to the high-priority scenarios. In addition, AGARSoC uses scheduling heuristics when generating test programs that may result in sub-optimal simulation cycles. Even though the resulting schedule may result in large test programs for unfavorable cases, it is mostly simulated significantly faster than the original software suites.

D. Bug detection

We only focus on identifying high-priority scenarios, defining coverage, and generating tests that hit these coverage goals. We do not investigate the ability to check for and find bugs.

VII. Related works

Hardware and software co-design and co-verification has been investigated extensively as a means to shorten time-to-market in designing SoCs [4, 6, 7]. Here, virtual prototyping platforms allow software engineers to develop and verify their software while hardware is still in development. AGARSoC extends beyond the current usage of HW/SW co-verification methodologies to enable early analysis of software with the purpose of guiding hardware verification.

Simpoints [9] have been widely used to find representative sections of single-threaded software for use in architectural studies. Unlike AGARSoC, Simpoints are neither designed for analyzing concurrent software in accelerator-rich SoCs nor for guiding the verification process. Ganesan *et al.*[8] propose a mechanism for generating proxy programs to represent multi-threaded benchmarks. Unlike AGARSoC, their solution does not address accelerator interaction scenarios.

In contrast to previous work on automatic coverage-directed test generation [10], AGARSoC presents a solution to automatically extract coverage models by analyzing software. While a few solutions have been proposed for generating coverage models from dynamic simulation traces [11], they focus on low-level interactions obtained from RTL simulations. AGARSoC's coverage models focus on high-level accelerator interactions and are ready much earlier in the verification process.

VIII. Conclusion

In this work, we developed a methodology and a set of tools that can guide the verification of accelerator interactions in accelerator-rich SoCs. Our proposed solution analyzes software using a high-level model of the SoC to identify high-priority accelerator interactions. Using this analysis, our tools generate coverage models for these high-priority interactions to be used during RTL verification. In addition, our tools can also generate compact test programs that can hit these coverage goals. We built and successfully demonstrated the capabilities of our tools for two different SoC platforms. Future work will improve the test generation capabilities of our tools to support more design approaches and generate more compact tests.

Acknowledgments: This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] Intel Atom x3 processor series. http://www.intel.com/content/www/us/en/processors/atom/atom-x3-c3000-brief.html.

[2] Qualcomm Snapdragon processors. https://www.qualcomm.com/products/snapdragon.

[3] Xilinx Vivado design suite. http://www.xilinx.com/products/design-tools/vivado.html.

[4] B. Bailey and G. Martin. *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer Publishing Company, Incorporated, 2009.

[5] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-rich architectures: opportunities and progresses. In *Proc. DAC*, 2014.

[6] G. De Micheli, R. Ernst, and W. Wolf, editors. *Readings in Hardware/-Software Co-design*. Kluwer Academic Publishers, 2002.

[7] M. Fujita, I. Ghosh, and M. Prasad. *Verification Techniques for System-Level Design*. Morgan Kaufmann Publishers Inc., 2008.

[8] K. Ganesan and L. K. John. Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. *IEEE Trans. on Computers*, 63(4), 2014.

[9] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: faster and more flexible program analysis. *Journal of Instruction Level Parallel*, September 2005.

[10] C. Ioannides and K. I. Eder. Coverage-directed test generation automated by machine learning – a review. *ACM TODAES*, 17(1), 2012.

[11] E. E. Mandouh and A. G. Wassal. CovGen: a framework for automatic extraction of functional coverage models. In *Proc. ISQED*, 2016.

[12] G. Martin, B. Bailey, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., 2007.

[13] S. Mochizuki *et al*. A 197mW 70ms-latency full-HD 12-channel video-processing SoC for car information systems. In *Proc. ISSCC*, 2016.

[14] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: a pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proc. ISCA*, 2014.

[15] E. Viaud, F. Pêcheux, and A. Greiner. An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles. In *Proc. DATE*, 2006.