Dynamic Adjustment of Test-Sequence Duration for Increasing the Functional Coverage

Zacharias Takakis[†], Dimitrios Mangiras[†], Chrysostomos Nicopoulos[‡], Giorgos Dimitrakopoulos[†]
[†]Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece
[‡]Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus

Abstract—**The importance of functional coverage during front-end verification is steadily increasing. Complete coverage statistics, possibly spanning from block- to top-level, are required as a proof of verification quality and project development status. In this work, we present a coverage-driven verification methodology that relies on coverage-directed stimulus generation, with the goal being to increase functional coverage and decrease test application time. The test application time given to each one of the available constrained-random test sequences is dynamically adjusted by a feedback-based mechanism that observes online the quality of each applied test. The higher the quality, the more cycles are assigned to this test for future trials. Misbehaving test sequences are automatically replaced by new ones, in order to spend verification cycles on other tests that actually improve functional coverage. The proposed methodology is successfully applied to the register renaming sub-system of a 2-way superscalar out-of-order RISC-V processor. The results demonstrate both increased functional coverage and reduced test application time, as compared to a purely random approach.**

I. Introduction

Diminishing technology feature sizes deep into the nanoscale regime have enabled digital systems of enormous sizes and complexities. As designs become increasingly more complex, the time required to verify their functionality – i.e., ensuring that all specifications are met under all circumstances – becomes prohibitively long, and occupies an ever increasing portion of the entire product development cycle.

Moreover, the criticality of verification is progressively growing, as modern approaches go beyond traditional functional verification, i.e., verifying that a chip "does what it is supposed to do". Nowadays, verification extends to new emerging requirements arising from security and functional safety demands, which have transformed the role of verification into the more elaborate process of verifying also that a chip "does nothing that it is not supposed to do" [1]. Given these elevated verification demands, the time needed to achieve desired coverage goals inevitably increases.

Therefore, it is imperative to develop effective techniques that can expedite design verification, in order to minimize the time-to-market effort.

As a fundamental ingredient of the overall verification plan, simulation-based verification is of paramount importance in identifying design bugs. During simulation, appropriately selected test sequences are applied to the Design Under Test (DUT) and the obtained response is monitored. The DUT's behavior is checked for correctness by comparing it to a golden reference model, and/or by utilizing assertions to identify design-property violations. The applied test sequences may either target specific features of the design (aka direct tests), or they may take the form of constrained-random tests that gradually and progressively explore the entire design state-space.

The two main figures of merit used to quantify the success of simulation-based functional verification are the *structural* and *functional* coverages [2]. The former metric checks how much of a design's RTL code has been exercised, and it is typically automatically calculated during simulation. On the contrary, functional coverage is more nuanced and cannot be automatically deduced by the design [3]. Instead, the verification engineer must identify the architecturally "interesting" semantics of the design and validate the overall functionality by counting events that trigger salient aspects of the design. Obviously, the quality of the applied test stimuli dictates the highest achievable level of functional coverage.

Assuming that an appropriate set of test sequences – of unknown quality – has been developed by the verification engineer, a critical question is how to apply those sequences (in terms of application order and simulation duration) to reach the maximum possible functional coverage in as little time as possible [4], [5], [6]. Ideally, one would prefer an automated test application process that (a) scales efficiently with design size and complexity, and (b) can rapidly converge to the maximum coverage possible under the chosen set of test sequences.

This overarching goal is precisely the target of this paper, which focuses on one particular aspect of the test application phase: how to intelligently decide the simulation duration of each of the available test sequences, in order to achieve coverage closure as quickly as possible [7]. Specifically, a novel feedback-based technique is proposed, which autonomously modulates the simulation duration of each test sequence during the entire verification campaign. The mechanism uses each sequence's incremental contribution to the functional coverage as feedback in deciding how long each sequence will execute in the next round of simulations. This feedback loop is instrumental in dynamically identifying sequence durations that enable faster increases in the coverage. Even though the proposed approach does not tinker with the *order* that the various test sequences are applied, our experimental results indicate that by optimizing the sequence *duration* alone is still enough to yield impressive gains in simulation times.

To validate the efficacy of the new feedback-based verification technique, we apply it to the UVM-based verification [8] of the Register Renaming Sub-system (RRS) of a 2-way superscalar out-of-order RISC-V processor [9], [10], the architecture of which is similar to the ARM Cortex-A53 and ARM Cortex-A57 [11]. The proposed technique takes a set of given constrained-random sequences of different parameters and various direct-test sequences and orchestrates their application (in terms of sequence duration) to the RRS module under test. The obtained results demonstrate that much higher functional coverage levels are achieved in substantially smaller simulation time, as compared to conventional simulation that executes each test sequence for the same number of a priori decided cycles. Reaped simulation-time savings of approximately 70% are reported for different sets of test sequences.

II. The Proposed Feedback-based Verification Methodology

The proposed methodology optimizes the duration of the test sequences applied to the DUT, with the goal being to reach coverage goals faster than a traditional approach that applies fixed-duration test sequences.

Initially, the test sequences are constructed and the expected coverage goals defined. In general, each test sequence targets one of the various interfaces of the DUT, and each interface is exercised by a number of distinct test sequences. The test sequences of each interface can either be monolithic, or parameterizable objects. Selecting a different parameter for each parameterized sequence can generate different instances of the same sequence. The granularity of parameter selection depends on the number of parameters and the relation of each parameter to the functionality of the DUT.

In the proposed framework, the test sequences are fixed beforehand, and all their parameters are pre-selected by employing a representative random parameter sampling. After this selection step, the framework does not allow any sequence to change any of its parameters, or its constraints. Note that the process of constructing and selecting test sequences is orthogonal to this work; it is up to the verification engineer to select a set of high-quality (in terms of achievable functional coverage) test sequences. The methodology introduced in this paper focuses on how a given set of test sequences is applied to the DUT.

In addition to constructing a set of appropriate test sequences for the design, the verification engineer also defines a set of coverpoints that describe architecturally interesting aspects of the DUT; e.g., a particular queue being empty, or full. The achieved coverage of those coverpoints during simulation provides a quantitative measure of the effectiveness of the test plan, and the ability of the test sequences to adequately explore the DUT. Each coverpoint is characterized by a set of bins, with each bin assuming a different number of values. For example, if a coverpoint includes the value of a 2-bit counter, then 2^2 bins will be created to keep track of the number of times each value of the counter occurs. For each bin, one must also define a coverage goal. A bin is considered fully covered when it is "hit" (actually encountered during simulation) at least as many times as its pre-defined coverage goal.

Each test sequence is allowed to execute for a certain number of cycles. During the application of the sequence to the DUT, the exercised coverpoints and bins are recorded. The "quality" of each test sequence – which is calculated upon completion of the sequence's run – is a measure of how many coverpoints/bins it hits, and, by extension, its contribution to increasing the attained functional coverage. When giving a quality rating to an applied sequence, we only count those bins that have still not reached their goal. The bins that have reached their goal are removed from the active list of bins that determine the quality. In this way, as the various bins reach their coverage goal, the proposed technique focuses on the hard-to-reach properties.

In general, there are two key attributes orchestrating the application of the various test sequences to the DUT: (1) the *order* of the test sequences, and (2) the *duration* of each sequence. The proposed technique targets the second attribute (duration), so the application order is assumed to be random, as is frequently the case in existing approaches.

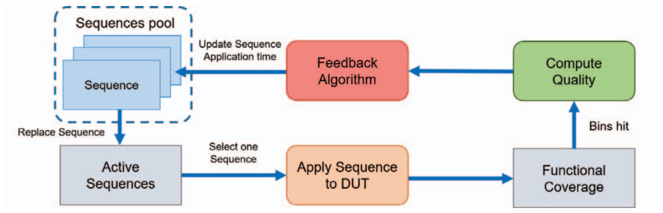

Fig. 1. Overview of the proposed feedback mechanism for selecting the duration of active test sequences. Poorly performing test sequences are replaced by new ones from a pool of available constrained-random test sequences.

Let us assume that the verification process commences with a set of K active test sequences, and an additional number of inactive test sequences that can be used on-demand to replace any of the K active ones (as will be described later on). The proposed technique uses the K active test sequences and the feedback mechanism highlighted in Figure 1 to guide the verification of the DUT. The technique outlined in the algorithm evaluates the quality of each test sequence and uses this measure as feedback in deciding the duration of each test sequence the next time it will be selected again.

Initially, all K active test sequences have a duration of STEP cycles. The feedback-based methodology randomly selects Sequence i to run for $STEP$ cycles. Upon completion of the run, the quality of the sequence is calculated (based on the number of hit coverpoint bins), as follows:

$$Quality = \frac{bins\_hit}{total\_uncovered\_bins}$$

However, this quality metric refers only to the most recent run of Sequence i. In order to have a more complete picture of the entire performance of Sequence i thus far, the proposed technique also utilizes a so called *drifting quality* metric for each test sequence, which is updated each time a sequence completes its execution:

$$DriftQuality[i] = (1-\alpha) \cdot DriftQuality[i] + \alpha \cdot Quality \quad (1)$$

The *drifting quality* is calculated by placing more weight on the last recorded quality figure than on the previous ones. The α parameter in Equation 1 takes values between 0 and 1, and it is a weight that modulates the emphasis to be given to the most recent quality, as opposed to prior values. The higher the value of α, the higher the emphasis on the most recent quality. For example, if $\alpha = 0.8$, the current drifting average $= 0.9$, and the most recent calculated

TABLE I
EACH TEST SEQUENCE IS CLASSIFIED INTO ONE OF 5 REGIONS, BASED ON ITS MEASURED DRIFTING QUALITY VALUE. THESE REGIONS THEN DICTATE THE NEW RUNTIME FOR EACH SEQUENCE.

Region	Range	New runtime
R1	[0, max_bound/5]	0 or STEP/2
R2	(max_bound/5, 2*max_bound/5]	STEP/2
R3	(2*max_bound/5, 3*max_bound/5]	STEP
R4	(3*max_bound/5, 4*max_bound/5]	2*STEP
R5	(4*max_bound/5, max_bound]	3*STEP

$quality = 0.1$, then: $drift\_avg[i] = (1-0.8) \cdot 0.9 + 0.8 \cdot 0.1 = 0.2 \cdot 0.9 + 0.8 \cdot 0.1 = 0.26$. As can be seen, even though Sequence i has had a very high drifting quality so far (0.9), a bad quality result in its most recent execution (0.1) is enough to drop the drifting quality from 0.9 to 0.26.

The drifting quality approach ensures that any sequence that starts yielding lower quality figures – after a potentially long streak of good-quality runs – will quickly be identified as a sequence that no longer contributes significantly to the coverage.

It is this drifting quality value that is subsequently used to decide the duration (in cycles) that Sequence i will be allowed to execute the next time it is selected again. Specifically, we first designate the value of the (currently) maximum drifting quality among all sequences as max_bound. Next, we divide the range [0, max_bound] into 5 regions, as shown in Table I. Finally, the drifting quality of Sequence i is used to classify the sequence into one of those 5 regions, which dictate the new runtime (duration) for Sequence i the next time it is selected (as per Table I).

For instance, if the calculated drift quality for Sequence i falls into R2, then the runtime of Sequence i will be $STEP/2$ clock cycles the next time it is selected again.

If the calculated drift quality for Sequence i falls into region R1, then Sequence i will be removed from the K active sequences, and it will be replaced with a new sequence (from the inactive pool of sequences), assuming that the number of sequence replacements so far has not reached an arbitrarily chosen maximum number ($max\_replacements$). If Sequence i cannot be replaced (due to reaching $max\_replacements$), it will have a runtime of $STEP/2$ cycles the next time around.

Note that the regions described above are of equal size. In reality, it is up to the verification engineer to decide the region bounds, i.e., the size of each region, and this decision is expected to vary with each DUT. In general, the size of each region affects the balance of fairness between the sequences and the overall aggressiveness of the algorithm. For example, one can artificially restrict the number of sequence replacements (decided by the algorithm) by making the R1 region smaller. Moreover, one may assign more clock cycles to the sequences exhibiting good drifting quality to make the algorithm more aggressive.

Upon completion of the above-mentioned steps, a new test sequence is randomly selected and the algorithm is repeated. This process continues until a desired functional coverage goal is reached.

The expected outcome of the proposed approach is shown graphically in Figure 2. We expect both higher coverage and lower overall test application time, since both easy-to-reach properties and harder corner cases are covered by test sequences applied according to their effectiveness (as identified through the feedback loop).

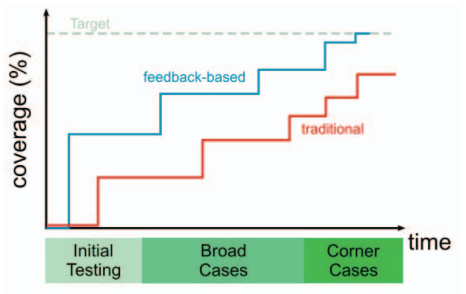

Fig. 2. The expected outcome of feedback-based test-sequence application. The proposed technique achieves both higher coverage and lower overall test application time.

III. CASE STUDY: FEEDBACK-BASED VERIFICATION OF THE RRS MODULE OF A SUPERSCALAR, OUT-OF-ORDER PROCESSOR

The proposed feedback-based verification technique can be applied at various levels of abstraction. At a low abstraction level, verification may target an individual sub-system, or unit, of a design. At the highest abstraction level, verification targets the entire system as a whole. Without loss of generality, in this paper we will focus – as proof of concept – on the verification of a single (albeit quite complex) sub-system in a superscalar Out-of-Order (OoO) microprocessor. Specifically, we will verify the Register Renaming Sub-system (RRS) of a 2-way superscalar RISC-V processor.

A. Fundamentals of Register Renaming

Register renaming is a technique that enables OoO execution by eliminating false register dependencies between instructions. There exist several alternative implementations of register renaming [12]. In this work, we evaluate register renaming with a merged register file. In such implementation, the results of operations are stored in a single physical register file that combines architectural and speculative state [13].

Register renaming with a merged register file uses a large pool of physical registers and translates the logical destination register of each instruction that produces data to a physical register. Typically, each instruction consists of a *logical destination* register (i.e., an architectural register that is part of the Instruction Set Architecture), and two *logical* source (or input) registers. During register renaming, each instruction's logical register specifiers are replaced with corresponding *physical* register specifiers, i.e., Pdst (physical destination) and Psrc (physical source) specifiers. Register renaming can be performed on either a single instruction at a time (in scalar processors), or on multiple instructions simultaneously (in superscalar processors). For example, the 2-way superscalar processor used in this work employs 2-way register renaming, i.e., 2 instructions can be renamed and retired per clock cycle.

Figure 3 shows the RRS assumed in this work (for simplicity the RRS of a scalar processor is drawn) that consists of the following hardware arrays:

Free List (FL): is a FIFO where Pdsts are initialized each time a core is powered on. A free Pdst is allocated to rename the logical destination register of an instruction. The Pdst

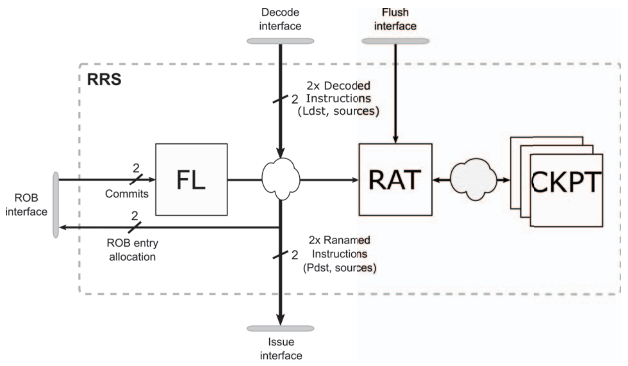

Fig. 3. The organization of the Register Renaming Sub-system (RRS) unit and its interfaces to the rest of the processor's pipeline.

is sent to the Reservation Station (RS) where the renamed instruction waits to execute. When the instruction executes, it updates the physical register pointed by its Pdst.

Register Alias Table (RAT): is a table with the most recent mapping of each logical register specifier to a Pdst. It is used to rename the input (i.e., source) logical registers of an instruction. The renamed Pdsts are forwarded to the RS of the instruction to determine when the instruction can execute.

Re-Order Buffer (ROB); not explicitly shown in Figure 3: is a FIFO with an entry allocated per instruction. Even though the ROB is not a part of the RRS, it interacts extensively with it. Each ROB entry has a field to hold the Pdst that is evicted from the RAT by the instruction (if the instruction writes to a register). The Pdst is freed when the instruction retires.

Checkpoint Table (CKPT): is used to take snapshots of the RAT. A single snapshot is taken every time an incoming branch instruction is encountered.

During processor operation, the CKPT buffer is useful for expediting the restoration of the RRS state following *pipeline flushes*. When a mispredicted branch instruction causes a flush, the RRS state is restored using the CKPT. The RAT is restored with the checkpoint stored when the offending branch instruction was originally encountered. The restoration process also returns to the FL all the Pdsts allocated after the offending instruction.

In addition to restoring the state of the arrays, the tail pointer of the ROB must be restored to the position corresponding to the flush-causing instruction. The FL head pointer is not restored, since the wrong-path Pdsts are written back to the FL upon completion of the CKPT retrieval using the FL tail.

B. Applying Feedback-based Verification to the RRS Unit of a Processor

To verify the RRS unit, we employ the ubiquitous UVM approach, which exemplifies verification modularity and reusability. The crux of UVM-based verification is the test-bench that comprises two main parts: (a) the one that produces the stimulus driving the inputs of the design, and (b) the checker that is used to verify the design's output against the modeled (expected) output. The stimulus – e.g., a test sequence – is produced by sequence generators, as will be explained shortly.

The RRS unit is connected to the rest of the processor via the four separate interfaces described in Table II. During simulation, each interface is fed by distinct test sequences that mimic the behavior of the interfaces when connected to the remaining sub-systems of the processor, and while the processor is executing real programs. The test sequences that drive each interface are the result of an independent constrained-random generator customized by a set of parameters, which are depicted in the right-most column of Table II for each interface.

TABLE II
THE PARAMETERIZED TEST SEQUENCES USED IN THE RRS UNIT.

Interface	Functionality	Test sequence parameters
Decode	The connection to the Decode stage. The RR stage receives the decoded instructions from the ID stage using a valid/ready protocol.	**Generation rate:** The probability to generate decoded instructions. **Dual instruction fetch rate:** The probability to generate two decoded instructions. **Branch rate:** The probability of a decoded decoded instruction to be considered a branch. **Dependence rate:** The probability that each source of a decoded instruction is equal its destination.
Issue	The RR stage outputs the renamed instructions to the Issue stage for execution.	**Ready rate:** The probability that the Issue stage will backpressure the RR stage by accepting or not the renamed instructions.
Re-Order Buffer (ROB)	The RR stage is updated by the ROB unit about the commits and the freed physical registers.	**Commit rate:**The probability that the ROB will retire an executed instruction. **Rob full rate:** The probability that the ROB will backpressure the RR stage by being Full. **Rob two empty rate:** The probability that the ROB will issue that it has two or more available entries.
Flush	The RR stage is updated by the flush controller about the issued flushes to properly restore RAT.	**Flush rate:** The probability for a branch to be mispredicted.

For example, the Issue interface sits at the output of the RRS unit and transfers the renamed instructions (i.e., instructions that have had their logical register identifiers replaced with corresponding physical register identifiers) to the processor's Issue stage. The latter is then responsible to dispatch the renamed instructions to the various execution pipelines/units. The random sequences driving this interface are controlled by a single parameter, as shown in Table II. Said parameter represents the probability that the Issue stage can accept the renamed instructions (as opposed to back-pressuring the RRS unit).

IV. RRS INTERFACES TO THE REST OF THE PROCESSOR

Given that the four interfaces are characterized by a number of parameters, we need to select a specific set of parameters for each test sequence that would remain fixed during the simulation. For example, in the case of the Re-Order Buffer (ROB) interface, a specific parameter choice could be Commit Rate = 40%, ROB Full Rate = 5%, and ROB Two Empty Rate = 80%; this would imply that the generated test sequence would exercise the ROB interface with stimuli conforming to those specific parameter rates.

In order to limit the search space, we assume that each of the interface parameters listed in Table II can take at most three distinct values. For example, the "Commit Rate" parameter

TABLE III
FUNCTIONAL COVERPOINTS AND NUMBERS OF BINS USED FOR THE SIMULATION OF THE RRS UNIT. EACH COVERPOINT HAS A PRE-DETERMINED "HIT" GOAL.

No	Coverpoint [#Bins]	Hits (at least)
1	Instruction rename from port 1 [1]	1K
2	Instruction rename from port 2 [1]	1K
3	Free list gone full [1]	1K
4	Free List gone empty [1]	1K
5	Dual push to FL [1]	1K
6	Dual pop from FL [1]	1K
7	Write to every entry of RAT [31×2 ports]	1K
8	RAT checkpoint [1]	1K
9	Dual RAT checkpoint [1]	1K
10	Write to every entry of Checkpoint array [4]	1K
11	Flush to every RAT id [4]	1K
12	RAT: Both write ports write to same entry [31]	100
13	Two instructions fetched for rename but only 1 preg available [1]	100
14	All combination of renames [4]	100
15	All combination of allocations between Lregs and Pregs [(31lregs*63pregs)= 1953 * 2 ports]	20

of the ROB interface assumes values of 5%, 40%, and 80%, which represent a low, medium, and high instruction-commit rate. From all available sequences, we select 50 test sequences randomly. This number was empirically chosen to ensure that each one of the possible parameter values (rates) is used at least one time. However, the verification engineer is free to choose any specific parameter set.

After selecting the test sequences that will be used during simulation, we need to define the coverage goals that each run must cover. A set of representative cover properties defined for the RRS unit are shown in Table III. For each functional coverpoint, Table III also lists – in square brackets – the number of bins associated with each coverpoint. For example, coverpoint 7 ("Write to every entry of RAT") is associated with 31×2=62 bins, because the Register Alias Table (RAT) has a size of 31 entries and is dual-ported. Thus, each bin measures the number of times a write has been issued to a particular RAT entry from a particular write port.

The coverpoints of Table III are considered fully covered when the bins of each coverpoint are "hit" at least as many times as the number indicated in the right-most column for each property. The hit goal set for each coverpoint is design-specific and is decided by the verification engineer. The goals may vary across the various coverpoints depending on their architectural significance. The actual hit goal numbers are orthogonal to how the proposed methodology works; in this work, we use the indicative numbers of Table III without loss of generality, and for proof-of-concept purposes.

V. EXPERIMENTAL RESULTS

During simulation-based verification, the proposed feedback-based framework applies the selected test sequences to the DUT. The duration of each sequence is dynamically adjusted based on its measured quality. Additionally, sequences that no longer contribute to increasing the coverage – and, consequently, decreasing the simulation time – are eventually replaced with other sequences. The ultimate goal is to reach coverage closure in as little time as possible.

In general, the development of a verification plan is an iterative process, and the coverpoints shown in Section IV

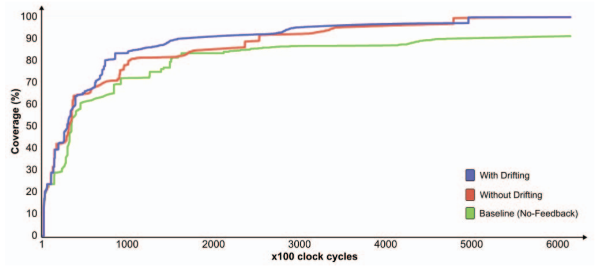

Fig. 4. The evolution of obtained coverage over simulation time. The proposed feedback-based methodology rises to a higher coverage in less time, and it eventually achieves 100% coverage.

constitute a 'snapshot' of the test plan used for the in-house verification process of the RRS. With this set of properties, we were allowed to (a) find interesting bugs during simulation, and (b) quantify the quality of the test sequences in covering the functional properties of the DUT.

In the first set of experiments, we highlight the effectiveness of the proposed feedback-based technique in achieving better coverage in smaller simulation time. This is demonstrated through a comparison with a baseline technique that does not alter the duration of test sequences, i.e., each test sequence has a fixed duration time. A set of 50 test sequences is available for the experiments.

When using the proposed feedback-based technique, we initially pick the first 10 test sequences (out of the available pool of 50) to be the active ones (see Section II). As the testing progresses, the algorithm intelligently replaces active sequences with other ones from the pool of 50, as described in Section II. Thus, under the proposed technique, only 10 sequences are active at any given time during the experiment, and sequence replacements seize if and when the entire pool has been exhausted. Each active sequence is selected randomly. Initially, all sequences have a runtime duration of 200 cycles. Based on their measured quality, the duration of each sequence changes dynamically during the experiment and can take values of 0, 50, 100, 200, and 300, which correspond, respectively, to the 5 regions (R1–R5) described in Section II.

Under the baseline technique, all 50 test sequences are available throughout the experiment. Each sequence is selected randomly (just like under the proposed technique), and it is applied to the DUT for a fixed duration of 200 cycles.

Both experiments consisted of 5000 trials, i.e., 5000 test-sequence applications, and the results are depicted in Figure 4. The graph illustrates the evolution of the total coverage of the DUT over time for one initial random seed. It is evident in the figure that the proposed feedback-based methodology quickly rises to a higher coverage in smaller time, and it manages to reach coverage closure and 100% coverage. On the other hand, the baseline approach without feedback control only manages to reach a coverage of 91% by the end of the experiment. The proposed method was tested *with* and *without* drifting. When drifting is not enabled, the measured quality for each test sequence is simply the average quality observed so far for that particular sequence. When drifting is enabled (and by using $\alpha = 0.5$), the achieved coverage increases more quickly than in the case where drifting is not enabled. Regardless, both schemes (with and without drifting) eventually converge to the same final coverage.

TABLE IV
RESULTS OF COMPARISON BETWEEN THE PROPOSED FEEDBACK-BASED TECHNIQUE AND A BASELINE APPROACH WITH NO FEEDBACK. THE PROPOSED METHODOLOGY REACHED 100% COVERAGE UNDER ALL 5 INITIAL SEEDS.

Seed	Baseline Coverage	Baseline #cycles	Feedback #cycles	Timing Savings
1	94.56	1,000,000	300,900	70%
2	95.00	1,000,000	309,600	69%
3	94.97	1,000,000	307,600	69%
4	94.83	1,000,000	307,000	69%
5	94.79	1,000,000	301,700	70%
Average	94.83	1,000,000	305,360	69%

The proposed method used all 50 sequences available in the pool; i.e., after starting with the initial 10 active sequences, the algorithm made a total of 40 replacements throughout the experiment. All of the replacements were triggered when the drifting quality of a sequence was too low, and the quality was inside the R1 region.

This comparison between the baseline and feedback-based approaches was repeated for 5 additional random initial seeds (and, thus, 5 different pools of 50 test sequences each). The proposed methodology reached 100% coverage under all 5 initial seeds, as opposed to the baseline approach. The results are shown in Table IV. As indicated, the proposed technique achieved full coverage very rapidly (in around 300k cycles in all cases). Instead, the baseline approach used up the entire duration of the experiment (1M cycles) without reaching coverage closure. On average, the feedback-based approach resulted in around 70% smaller simulation time, while also achieving higher coverage (full closure).

Finally, the sensitivity of the proposed method to the size of the set of *active* test sequences was also evaluated. We increased the number of active test sequences from 10 (out of the total of 50) sequences to 20, 30, 40, and 50, and recorded the functional coverage achieved after 5000 trials. The obtained results are depicted in Figure 5. The smaller the number of active sequences at any given time, the higher the coverage obtained after 5000 trials. When the number of active sequences is small, the sequences are explored quickly for their efficiency. If they behave well, they are regularly repeated. If they provide poor coverage, they are quickly replaced by new test sequences. On the contrary, when the number of active test sequences approaches the total number of available sequences, then each sequence is repeated less often, and judgment about its quality is delayed. Inevitably, many test trials are "lost" to useless (in terms of their contribution to coverage) sequences, until they are replaced by new sequences.

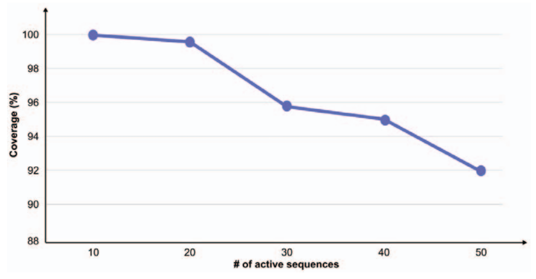

Fig. 5. The coverage obtained after a total of 5000 trials, by varying the size of the set of active test sequences.

VI. CONCLUSIONS

Successful test application involves many parameters, such as appropriate test sequence selection, tuning of constrained-random generator parameters, determining the order of application of the derived tests, and deciding the duration of each test sequence. In this paper, we focus on optimising the test-duration parameter by dynamically optimizing how long each test sequence should be applied. Instead of relying on predetermined test durations, we measure the coverage achieved by the applied test after each trial. The higher the coverage, the more cycles will be dedicated to this sequence in future trials. Test sequences that do not improve coverage are penalized by receiving fewer cycles in the future. When the number of cycles per sequence continuously diminish, the test sequence is replaced by a new one, hoping for greater performance. This coverage-driven test application technique is applied to the register renaming sub-system of a RISC-V processor, demonstrating that it can efficiently reduce test application time and improve the final functional coverage. In the future, we plan to apply the proposed technique to top-level software tests that verify the processor as a whole.

ACKNOWLEDGMENTS

Dimitrios Mangiras is supported by the Onassis Foundation - Scholarship ID: G ZO 014-1/2018-2019.

REFERENCES

[1] W. C. Rhines, "Design verification challenges: Past, present and future," Design and Verification Conf. (DVCON)- Keynote Address, 2016.
[2] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, 2005.
[3] S. Mutschler, "Mitigating risk through verification: Automatic coverage model generation technology continues to advance," Dec, 2018. [Online]. Available: http://semiengineering.com/mitigating-risk-through-verification/
[4] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test*, vol. 18, no. 4, pp. 36–45, Jul. 2001.
[5] E. Nelson, "Improving constrained random testing by achieving simulation verification goals through objective functions, rewinding and dynamic seed manipulation," in *Design and Verification Conf. (DVCON)*, 2017.
[6] R. Roy, C. Duvedi, S. Godil, and M. Williams, "Deep predictive coverage collection," in *Design and Verification Conf. (DVCON)*, 2018.
[7] M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal, "A study in coverage-driven test generation," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1999, pp. 970–975.
[8] R. Salemi, *The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology*. Boston Light Press, 2013.
[9] A. Waterman, "Design of the RISC-V instruction set architecture," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html
[10] K. Patsidis, D. Konstantinou, C. Nicopoulos, and G. Dimitrakopoulos, "A low-cost synthesizable RISC-V dual-issue processor core leveraging the compressed instruction set extension," *Microprocessors and Microsystems*, vol. 61, pp. 1 – 10, 2018.
[11] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanović, "BOOM v2: an open-source out-of-order RISC-V core," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html
[12] A. González, F. Latorre, and G. Magklis, *Processor Microarchitecture: An Implementation Perspective*. Synthesis Lectures on Computer Architecture,Morgan & Claypool Publishers, 2010.
[13] K. C. Yeager, "The mips r10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, April 1996.