A Reinforcement Learning Approach to Directed Test Generation for Shared Memory Verification

Nícolas Pfeifer, Bruno V. Zimpel, Gabriel A. G. Andrade, Luiz C. V. dos Santos
Federal University of Santa Catarina
Florianópolis, Brazil

***Abstract*—Multicore chips are expected to rely on coherent shared memory. Albeit the coherence hardware can scale gracefully, the protocol state space grows exponentially with core count. That is why design verification requires directed test generation (DTG) for dynamic coverage control under the tight time constraints resulting from slow simulation and short verification budgets. Next generation EDA tools are expected to exploit Machine Learning for reaching high coverage in less time. We propose a technique that addresses DTG as a decision process and tries to find a decision-making policy for maximizing the cumulative coverage, as a result of successive actions taken by an agent. Instead of simply relying on learning, our technique builds upon the legacy from constrained random test generation (RTG). It casts DTG as coverage-driven RTG, and it explores distinct RTG engines subject to progressively tighter constraints. We compared three Reinforcement Learning generators with a state-of-the-art generator based on Genetic Programming. The experimental results show that the proper enforcement of constraints is more efficient for guiding learning towards higher coverage than simply letting the generator learn how to select the most promising memory events for increasing coverage. For a 3-level MESI 32-core design, the proposed approach led to the highest observed coverage (95.81%), and it was 2.4 times faster than the baseline generator to reach the latter's maximal coverage.**

***Index Terms*—Multicore chips, shared memory, design verification, reinforcement learning, decision process.**

I. Introduction

Multicore chips are expected to rely on coherent shared memory [5]. Albeit the coherence hardware can scale gracefully [19], the protocol state space grows exponentially with core count. Besides, sophisticated architectures (e.g. ARMv8, IBM Power9, RISC-V) relax sequential consistency, largely increasing the number of valid execution witnesses of a parallel program, making the detection of invalid witnesses more difficult. The combination of coherence and relaxed consistency makes the validation of shared memory behavior a very challenging task that has deserved specific techniques. Most of them fall in two main approaches: litmus test generation [3], [17] and random test generation (RTG) combined with memory model checking [13], [15], [18]. The first approach exploits a memory model for synthesizing small programs able to expose invalid execution witnesses. Although quite efficient to find errors, its coverage control is limited. The second approach exploits RTG for raising the coverage of memory events and lets an independent checker verify the axioms of a memory consistency model. When applied at design time, however, such approach requires directed test generation (DTG) for efficient coverage control under the tight time constraints resulting from slow simulation and short verification budgets. Coverage control can be obtained statically [20] or dynamically [8], [11], [22]. A pragmatic approach to dynamic coverage control is the casting of DTG as coverage-driven RTG [10]. This paper's proposal adopts a similar casting.

Figure 1 shows our coverage-driven RTG approach. A few parameters $(p_1, p_2, \cdots, p_M)$ are used to constrain the generation of parallel *test* programs. From a given setting for the parameters, the RTG engine produces a test, which is executed in a simulator. During simulation, a few events serve as coverage witnesses, which are converted by a Coverage Analyzer into a coverage estimate. With basis on estimated coverage and elapsed time, a Directing Engine makes a decision that redefines the settings for the parameters so as to dynamically try to maximize coverage under a time constraint.

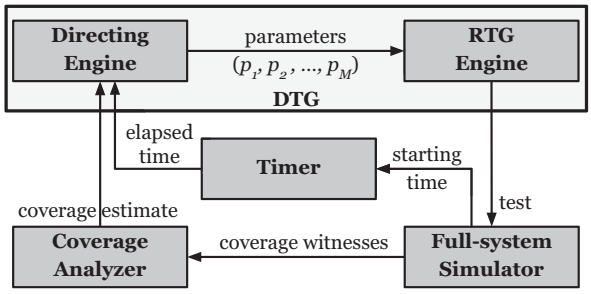

Fig. 1. A coverage-driven RTG approach to DTG

The main contributions of this paper are the following:

1) A novel technique for DTG that exploits Reinforcement Learning for reaching higher coverage in less time, as a result of successive actions taken by an agent that influences RTG. The agent was designed to be *reusable* (regardless of different coverage metrics adopted in distinct verification environments), and its actions are *customizable* (depending on the choice of random test generator).
2) An evaluation of how domain-specific information (properties of shared memory and parallel programs),

This work was financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq) - grant 141686/2019-7 - and by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

usually captured as constraints on RTG, can be exploited for improving learning.

II. Target problem and proposed approach

This paper addresses DTG for shared-memory verification. It casts DTG as coverage-driven RTG, which is formulated as a time-constrained optimization problem, as follows.

Problem: Given a state space defining valid shared-memory behaviors, find a sequence of random tests that maximize some coverage metric c subject to a time constraint at (available time).

However, we do not address that general verification problem directly. Instead, we model the definition of a sequence of tests as a decision process, and we try to find a decision making policy that maximizes cumulative rewards resulting from successive actions taken by an agent. Since the action probabilities of causing state transitions are unknown, the target instance becomes a Reinforcement Learning (RL) problem.

III. Related work

A. Constrained RTG legacy

RTG has been used for synthesizing parallel programs for shared memory validation of prototype multicore chips [13], [18]. Constraints usually enforce *structural properties of parallel programs*. They are formulated as generation parameters, such as the number of operations, the number of shared locations, and the number of threads (assuming that each test thread is bound to a different core).

RTG has also been used for validation at design time. For instance, Genesys-Pro [1] is an approach to functional processor verification that is based on constrained RTG. The handling of constraints is formulated as follows. The approach casts test generation into a constraint satisfaction problem and uses a generic solver customized for RTG to improve test program quality. Albeit part of the constraints capture design-*specific* testing-knowledge information from a database, the approach also provides generic *biasing* constraints that are applicable to any processor. Biasing constraints do not capture program properties, but try to enforce *functional hardware properties* (*e.g.* address alignment, cache eviction, etc.) for improving test quality.

A recent technique proposed a complementary way of constraining RTG. It enforces canonical multiprocessor chains of operations across different threads [4]. Since canonical chains consist of operations colliding at the same location in different threads and at least one of them is a store, a *chaining* constraint has the potential to raise the number of data races among threads, which is a known mechanism to expose design errors faster [13]. In other words, a chaining constraint enforces *functional properties of parallel programs* for improving test quality.

B. DTG for shared memory verification

An early learning approach to DTG proposed the exploitation of statistical inference to build a Bayesian network for defining the most probable generator settings that would achieve a certain coverage goal [11]. The Bayesian network was used as a centralized directing engine for dynamic coverage control, thereby casting DTG as coverage-directed constrained RTG. This technique required an offline training phase (to establish the basis for future online decision making), which might become a drawback, unless its contribution to the overall effort can be kept negligible. To ensure fast and proper training, however, test expertise may be required [10].

Instead of a centralized directing engine, a later learning-based approach relied on distributed intelligent agents, each working at a distinct core domain, which cooperate to improve the overall transition coverage. MCjammer [22] is a scalable scheme that avoids the enumeration of the full protocol space. Each agent formulates its coverage goals according to a dichotomic finite state machine (FSM), which captures the protocol behavior from the perspective of each core domain. Given a core domain, a state in its dichotomic FSM captures the state of a block in the local cache and an *aggregation* of the state of that block in the caches from other domains. The agents exploit the insufficiently verified transitions to formulate their goals towards higher transition coverage. The generator is reusable only for derivative designs that comply with the same protocol, because the dichotomic FSM must be modified for porting the generator to a protocol variant.

The most recently reported approach relied on Genetic Programming for learning how to build new tests from old ones. McVerSi [8] tailors the fitness function to the target verification scope. To obtain a new population from the fittest tests, it employs a selective crossover function that favors the selection of memory operations contributing to higher non-determinism. In McVerSi, the RTG engine is largely unconstrained, while its centralized directing engine exploits non-determinism. As opposed to MCjammer [22], whose mechanism is tied to its inner coverage metric, McVerSi's directing engine distinguishes the externally measured coverage from the inner mechanism for fostering coverage improvement. In other words, McVerSi is reusable across verification environments, as opposed to MCjammer.

C. Reinforcement learning for DTG

RL has already been used for hardware verification, but at the logic level. It was exploited, for instance, to influence the generation of random tests so as to raise the probability of design error discovery [21]. On average, as compared to a conventional technique, that approach led to a 15.3% improvement on fault coverage.

Besides, RL has been used for software validation. The validation of software modules requires the laborious work of generating data for a given set of tests. That is why validation techniques usually focus on relevant *data* generation (not in *test* generation). In contrast, an approach proposed a change of focus: the use of RL *not* to generate relevant data, but to synthesize new tests [12]. In this case, the agent was rewarded only if a newly synthesized test led to some software behavior not yet observed. As compared to software test based on random generation, the technique was clearly

superior only when targeting modules requiring complex input sequences (e.g. heaps). In general, however, albeit competitive, the technique was slightly inferior to random testing.

A more recent approach [16] exploited RL under the conventional data generation focus. It proposed a framework that casts the software under test as the environment and relies on it for training a neural network. After training, the agent learned how to mimic the behavior of meta-heuristic techniques that had shown good results on the creation of new data for the tests. As a result, even when faced with unseen environments, the approach was able to achieve a considerable coverage value. However, random search still required less time to reach higher coverage values.

In short, such early uses of RL for DTG [12], [16], [21] led to small improvements over random generation/search. This indicates that RL should rely on domain-specific properties for improving test quality, as did RTG approaches [2], [4], [13], [18] and previous learning approaches [8], [10], [11], [22]. The next section shows how our proposal bridges that gap.

IV. The proposed technique

A. Formulation of the Decision Process

The *environment* includes an RTG engine, the simulator, and the Coverage Analyzer. The Directing Engine is formulated as an *agent* that takes *actions* in such environment. The Coverage Analyzer and the Timer interpret the environment into a *state* representation and a *reward* value assigned to each action taken in a given state.

As verification is constrained by a time limit for reaching coverage goals, a suitable representation for an environment state would be a pair (c, t), where c denotes the cumulative coverage value (quantified by some metric adopted by the verification framework) and t denotes the time when that value was reached. However, to bound the number of states, we apply quantization on the values of coverage and time.

Coverage is quantized in C levels, and time is quantized into T levels[1]. As a result, when a pair (c, t) is observed from the environment, it is interpreted into the state representation $e = (\gamma, \tau)$, where $\gamma \in \{1, 2, \cdots C\}$ and $\tau \in \{1, 2, \cdots T\}$ denote, respectively, values of c and t rounded to the nearest quantization levels. Therefore, under such interpretation, the environment state space is $E = C \times T$.

Since the agent interacts with the environment through the RTG engine's interface, we formulate actions in terms of the parameters of a given constrained random test generator adopted as RTG engine. Let $p_1, p_2, \cdots, p_j, \cdots, p_M$ be the parameters that command a given RTG engine. Let V_j denote the collection of allowed values for parameter p_j. Therefore, $V = V_1 \times V_2 \times \cdots \times V_j \times \cdots V_M$ is the generation space for the RTG engine. Let $v = (v_1, v_2, \cdots, v_j, \cdots, v_m)$ and $v' = (v'_1, v'_2, \cdots, v'_j, \cdots, v'_M)$ denote allowable settings for the RTG parameters. Let a be an action that changes the RTG parameters from v to v'. Let (c, t) and (c', t') denote the cumulative coverage and the elapsed time observed after the execution of the tests generated with the settings v and v', respectively. An action should be better rewarded than another when the former induces a higher coverage increment in less time. Therefore, a suitable reward for an action a is $R_a(v, v') = (c' - c)/(t' - t)$.

With this formulation, we want to find a policy (sequence of actions) for reaching the maximal coverage ($\max c$) within the available time ($t < at$).

B. Proposed actions

The set of actions is largely dependent on the adopted RTG engine. For instance, conventional RTG engines [13], [18] use two main parameters: the number of memory operations (n) and the number of shared locations (s). On the other hand, the RTG engines proposed in [4] employ a third parameter: the number of distinct cache sets to which locations can be mapped (k). Without loss of generality, this paper defines actions only for the above mentioned RTG engines.

We assume that the verification engineer defines bounds on the allowed test sizes (n_{min} and n_{max}) and on the allowed amount of shared locations (s_{min} and s_{max}).

1) Two-parameter actions: Let N and S be the sets of allowed values for the parameters n and s (respectively) that are within user-defined bounds, and are induced by the range of functions[2] $f(i) = \lceil 2^{i/2} \rceil$ and $f'(i) = 2^i$, as follows:

$$N = \{n : n_{min} \leq n \leq n_{max} : n = \lceil 2^{i/2} \rceil \; for\, some \; i \in \mathbb{N}\},$$

$$S = \{s : s_{min} \leq s \leq s_{max} : s = 2^i \; for\, some \; i \in \mathbb{N}\}.$$

We define the following actions:

- $a_1(n, s) = (\lceil \sqrt{2}n \rceil, s)$
- $a_2(n, s) = (\lceil n/\sqrt{2} \rceil, s)$
- $a_3(n, s) = (n, 2s)$
- $a_4(n, s) = (n, s/2)$

2) Three-parameter actions: The first two parameters are n and s, whose sets of allowable values are defined above. Let us now consider the third parameter. The values allowed for the parameter k are bounded for each allowed value of s, and are constrained to be multiples[3] of that value, as follows:

$$K = \{k : (1 \leq k \leq s) \wedge (s \in S) \wedge (s \bmod k = 0)\}.$$

We define the following actions:

- $a_1(n, s, k) = (\lceil \sqrt{2}n \rceil, s, k)$
- $a_2(n, s, k) = (\lceil n/\sqrt{2} \rceil, s, k)$
- $a_3(n, s, k) = (n, 2s, k)$
- $a_4(n, s, k) = (n, s/2, k)$
- $a_5(n, s, k) = (n, s, 2k)$
- $a_6(n, s, k) = (n, s, k/2)$

[1]Without loss of generality, but for simplicity, this paper assumes uniform quantizers.

[2]These could be replaced by other functions without loss of generality, as far as actions are accordingly adjusted. They were designed for finer-grain control on parameter n than on the others, because n largely affects test throughput.

[3]This is a constraint leading to a uniform distribution of locations competing for cache sets, which tends to foster higher coverage.

C. The underlying model

In order to build a directed test generator that could be *reused* for distinct coverage metrics adopted in different design environments, we do not give the agent direct access to coverage events. As a result, it needs to be able to handle *partial* observation of the state in order to learn in the environment.

Recurrent Neural Network (RNNs) are a viable option for partially observable Markov Decision Processes, because their ability of handling time and memory makes them suitable for modeling any type of dynamical system [6].

We opted for an RNN with a single 10-neuron recurrent layer between fully-connected input and output layers, and 11 distributional RL atoms[4]. Since an RNN is trained with sequences, we used subsequent RL transitions[5] for training. We relied on sequences of length 8 and learning rate of 0.01.

As the tests used for training would not impair coverage, but actually contribute to its cumulative effect, the nature of the problem allowed us to opt for *online* training. At the start of every test-suite execution, we used a new set of random weights for the network, and trained them during its execution.

Our implementation is an adaptation of the Rainbow agent [14], but the original neural network was replaced by our RNN.

V. Experimental validation

A. Experimental set up

Three generators were built under the proposed approach. To build each Reinforcement Learning Generator (RLG), we used the same Directing Engine and selected a distinct RTG engine. We selected three RTG engines subject to progressively tighter constraints, which are denoted as follows. RLG- relies on an RTG engine that constrains the numbers of operations and locations only, similarly to [13], [18]. RLG+ relies on an RTG engine that not only constrains operations and locations, but also employs *biasing* constraints for controlling cache evictions, similarly to [11]. RLG* relies on an RTG engine that enforces the same constraints as the previous ones, but imposes extra *chaining* constraints, similarly to [4]. RLG- employs two-parameter actions, while both RLG+ and RLG* employ three-parameter actions. We set the same ranges for their common parameters: $n_{min} = 1Ki$, $n_{max} = 64Ki$, $s_{min} = 4$, and $s_{max} = 128$.

We compared the proposed RLGs with the McVerSi [8] test generator (MTG), which is available in the public domain [7]. We preserved all genetic parameters exactly as they were originally set in [8]. Since the MTG can only generate fixed-size tests (as opposed to our generators), we launched experiments for test sizes at the extremes of the range adopted for our generators (*i.e.* $n = 1Ki$ and $n = 64Ki$). To ensure that the MTG operated in a similar range of shared locations as our generators, we adopted the test memory constraint of 8KB, as defined in [8].

We relied on gem5's infrastructure [9] for simulation and design representation of 32-core designs (O3 processor model) under coherent shared memory (Ruby model). To reduce a possible dependence of the results on protocol variant, we adopted either a 2-level (L1, L2) or a 3-level (L0, L1, L2) MESI directory protocol with 4KB (directed-mapped) private caches at L0, 64KB (2-way) private caches at L1, and a 2MB (8-way) shared L2 cache, all with same block size (64 bytes).

To capture coverage evolution, we used the structural metric defined in [8], which tracks the state transitions of the cache controller's FSMs at all hierarchical levels and in every core domain. However, to reflect the hardware structure and not the protocol state space, the metric does not distinguish between transitions from different core domains. We report a coverage value as the fraction of transitions covered after the execution of a sequence of tests. The agent relies on the cumulative coverage up to a given test to decide on the most adequate setting of parameters for the next test.

Without loss of generality but for experimental convenience, we let each generator run until it stopped or a time limit of 10 hours (emulating a verification budget) was reached. Taking into account the average test runtime for the adopted generation space, we opted for T=1000 levels for time quantization. We arbitrarily selected C=100 levels for coverage quantization.

B. Impact of learning on coverage evolution

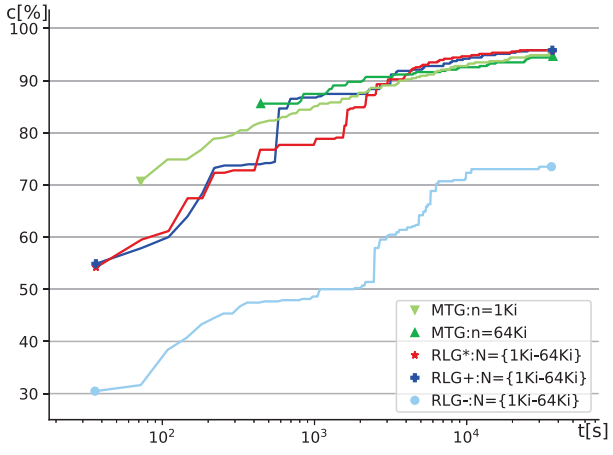

Fig. 2. Coverage evolution for 3-level MESI

Figure 2 shows the coverage evolution for a 3-level design. Notice that the evolution of RLG-, which relies on conventional random generation, is the poorest among all generators. This indicates that the simple exploitation of structural properties of parallel programs by the RTG engine is not sufficient for proper learning (as will be explained in Section V-C). The RLG+ and RLG* are not only superior to RLG-, but also competitive with the MTG. This indicates that the exploitation of hardware properties and functional properties of parallel programs are able to improve the quality of the generated tests under RL.

[4]Distributional RL is an optimization where the agent learns to approximate a distribution of the rewards, instead of the expected reward. These distributions are modeled as probability masses placed on a discrete support defined as a vector, where each component is called an *atom* [14].

[5]An RL transition is essentially a transition between environment states that was induced by a given action and was assigned a given reward [14].

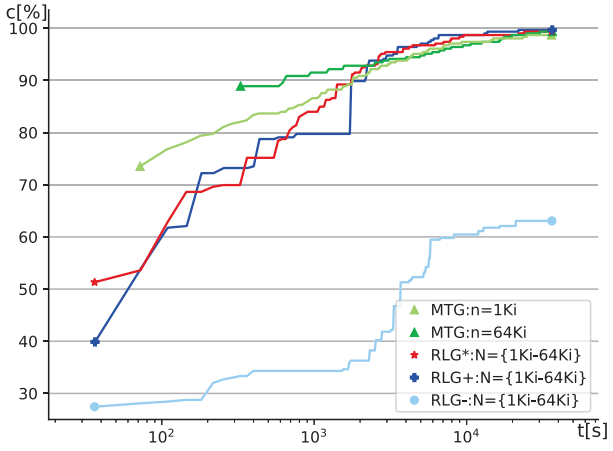

Fig. 3. Coverage evolution for 2-level MESI

Note also that RLG+ reaches higher coverage than RLG* until around 1 hour. After that, the tighter constraints exploited in its RTG engine enable RLG* to reach higher coverage values faster. Albeit their difference might seem small, it is indeed relevant in face of the diminishing returns as coverage approaches 100%. This explains why, after the 3-hour mark, both RLG* and RLG+ exhibit similar evolution. Therefore, in general, as more constraints are added, the generator becomes better suited to reach higher coverage values.

Observe that, for 64Ki, MTG is better than its 1Ki counterpart until around the 2-hour mark. After that point, the shortest test size started to pay off, because the higher test throughput allowed the genetic algorithm to create a larger number of tests in the same interval to cover new transitions. This permitted the MTG to reach higher coverage with the shortest tests.

RLG*'s final coverage was 95.81%, while MTG's was 94.88% (with $n = 1Ki$). However, MTG took around 26.000 seconds to reach its highest coverage, while RLG* took around 10.700 seconds to reach that same coverage, i.e. 2.4 times faster. Thus, as far as the RTG engine is properly constrained with biasing and chaining constraints, the use of RL for dynamic coverage control is not only competitive with MTG, but may allow the RLG to achieve higher final coverage[6].

Figure 3 shows that coverage evolution is similar for 3-level and 2-level designs, except that a higher final coverage was reached for the latter. Since the suppression of one hierarchical level reduced the overall number of transitions, a larger fraction of them was covered in the same time frame. Surprisingly, only RLG- did not benefit from that, since it reached a smaller final coverage for a 2-level design. This can be explained as follows. The RTG engine used by RLG- randomly assigns addresses to locations, as opposed to the others, which constrain address assignment for better control on replacements. Thus, it is harder for RLG- to stimulate transitions induced by replacements, especially as cache associativity increases. As the 2-level design was built by suppressing the directed-mapped L0 cache, the first hierarchical level was granted higher (2-way) associativity. As a result, a higher fraction of the remaining transitions became harder to cover with RLG-.

As expected from the suppression of one level, the MTG reached a final coverage value (99.35%, with $n = 64Ki$) closer to RLG+'s and RLG*'s (99.67%, and 99.35%, respectively). Albeit it seems to indicate that tighter constraints do not contribute as much to coverage evolution when verifying a less challenging design, the enforcement of constraints may be crucial to stimulating the hardest-to-cover transitions

C. Impact of problem-specific information on learning

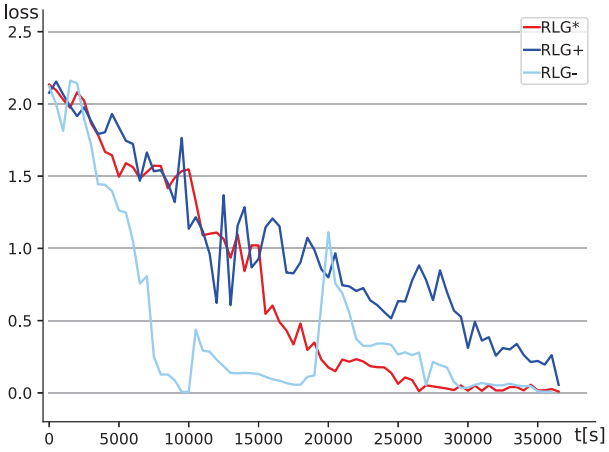

Fig. 4. Impact of constraints on learning for 3-level MESI

Figures 4 and 5 show the learning evolution for 2-level and 3-level designs. They plot the median loss[7] as a function of time. The loss essentially represents the difference between the prediction and the actual output of the environment. In our case, the prediction of the RNN is the expected reward for each one of the possible actions, and the output of the environment is the actual reward obtained. Therefore, a sharp decrease in the loss function means that the agent is learning faster.

Figure 4 shows the learning evolution for a 3-level design. All RLG variants are clearly learning with time, but their behavior is quite different. The sharpest evolution observed for RLG- indicates that, albeit the agent learns fast, it stops learning prematurely. The fact that the agent is unable to keep learning after the 3-hour mark explains why it got stuck at practically the same coverage after that time (as seen in Figure 2). This happens because the two-dimensional generation space of the underlying RTG engine was exhausted.

In contrast, the RTG engines underlying RLG+ and RLG* have much larger three-dimensional generation spaces. That is

[6]The MTG and the RLG* covered, respectively, 204 and 206 transitions. Thus, their difference in the final coverage corresponds to 2 hard-to-stimulate transitions that RLG* covered due to chaining and biasing constraints.

[7]The median loss was obtained as follows: First, for each random seed, the 10-hour runtime was divided into 72 intervals, each representing 500 seconds. Then we determined the median loss value in the scope of each interval, and we built a 72-point function for each random seed. Finally, we obtained the overall loss function by taking the median value of each point over 10 seeds.

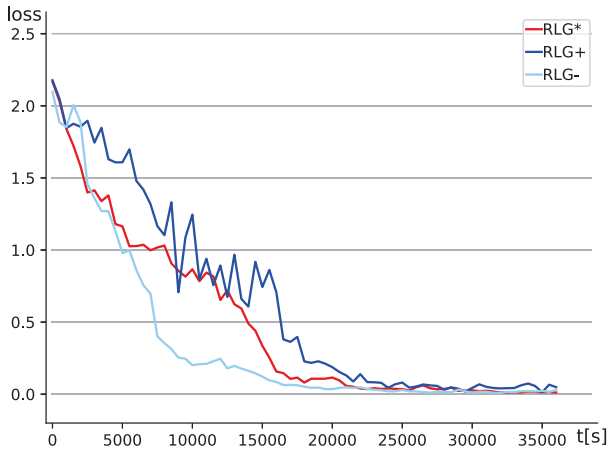

Fig. 5. Impact of constraints on learning for 2-level MESI

why RLG+ and RLG* can keep learning longer. Thus, RTG engines with more parameters or larger ranges of parameters have higher potential for coverage control under RL.

Albeit, for RLG-, the loss increased after the 3-hour mark, this does not represent the generator's behavior in every execution. In 7 of the 10 executions, its generation space was exhausted in 3 hours. Therefore, loss values reported after the 3-hour mark represent only the remaining 3 executions.

Note that, although the learning behavior of RLG+ and RLG* are quite 'noisy' up to the 6-hour mark, this changes afterwards. In the final phase of the verification process, the residual loss is higher for RLG+ than for RLG*. This indicates that RLG* is able to learn how to reach the few harder-to-cover transitions left. This is evidence that the combination of biasing and chaining constraints in the RTG engine (as it is the case for RLG*) improves the quality of RL exactly when it is more difficult to further increase coverage.

Figure 5 shows that the learning evolution for a 2-level design is similar to 3-level, as seen in Figure 4.

VI. Conclusions and Future Work

The experimental results show that Reinforcement Learning leads to an effective technique for directed test generation when it builds upon the legacy from random test generation. They show that Reinforcement Learning for shared memory verification is largely improved when shared memory and parallel program properties are exploited by an RTG engine.

The partitioning of complementary tasks into different modules (one exploiting what is known, another exploring what is unknown) seems to have the synergy required by next generation verification tools.

As future work, we intend to broaden the evaluation with additional coverage metrics and a more extensive set of designs. Besides, we intend to study the impact of our generators on design error discovery.

References

[1] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-pro: innovations in test program generation for functional processor verification. *IEEE Design Test of Computers*, 21(2):84–93, Mar 2004.

[2] A. Adir, D. Goodman, D. Hershcovich, O. Hershkovitz, B. Hickerson, K. Holtz, W. Kadry, A. Koyfman, J. Ludden, C. Meissner, A. Nahir, R. R. Pratt, M. Schiffli, B.St. Onge, B. Thompto, E. Tsanko, and A. Ziv. Verification of transactional memory in power8. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[3] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 258–272, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[4] G.A.G. Andrade, M. Graf, and L.C.V. dos Santos. Chain-Based Pseudorandom Tests for Pre-Silicon Verification of CMP Memory Systems. In *34th IEEE International Conference on Computer Design (ICCD)*, pages 552–559, 2016.

[5] S. Devadas. Toward a coherent multicore memory model. *Computer*, 46(10):30–31, 2013.

[6] S. Duell, S. Udluft, and V. Sterzing. Solving partially observable reinforcement learning problems with recurrent neural networks. In *Neural Networks: Tricks of the Trade*, pages 709–733. Springer, Berlin, Heidelberg, 2012.

[7] M. Elver. Mcversi framework. https://github.com/melver/mc2lib, 2016.

[8] M. Elver and V. Nagarajan. McVerSi: A test generation framework for fast memory consistency verification in simulation. In *IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*, pages 618–630, 2016.

[9] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug 2011.

[10] S. Fine, L. Fournier, and A. Ziv. Using bayesian networks and virtual coverage to hit hard-to-reach events. *International Journal on Software Tools for Technology Transfer*, 11(4):291–305, 10 2009.

[11] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 286–291, New York, NY, USA, 2003. ACM.

[12] A. Groce. Coverage rewarded: Test input generation via adaptation-based programming. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 380–383, Nov 2011.

[13] S. Hangal, D. Vahia, C. Manovit, and J.J. Lu. TSOtool: A program for verifying memory systems using the memory consistency model. *ACM SIGARCH Comp. Arch. News*, 32(2):114–123, Mar 2004.

[14] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI Conference on Artificial Intelligence*, 2018.

[15] W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li. Linear Time Memory Consistency Verification. *IEEE Transactions on Computers*, 61(4):502–516, Apr 2012.

[16] J. Kim, M. Kwon, and S. Yoo. Generating test input with deep reinforcement learning. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*, SBST '18, pages 51–58, New York, NY, USA, 2018. ACM.

[17] D. Lustig, M. Pellauer, and M. Martonosi. Pipe check: Specifying and verifying microarchitectural enforcement of memory consistency models. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 635–646, Washington, DC, USA, 2014. IEEE Computer Society.

[18] C. Manovit and S. Hangal. Completely verifying memory consistency of test program executions. In *IEEE Int. Symposium on High-Performance Computer Architecture (HPCA)*, pages 166–175, 2006.

[19] M.M.K. Martin, M.D. Hill, and D.J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, June 2012.

[20] X. Qin and P. Mishra. Automated generation of directed tests for transition coverage in cache coherence protocols. In *Design, Automation, and Test in Europe (DATE)*, pages 3–8, 2012.

[21] N. Shakeri, N. Nemati, M.N. Ahmadabadi, and Z. Navabi. Near optimal machine learning based random test generation. In *2010 East-West Design Test Symposium (EWDTS)*, pages 420–424, Sep. 2010.

[22] I. Wagner and V. Bertacco. MCjammer: Adaptive Verification for Multi-core Designs. In *Design, Automation, and Test in Europe (DATE)*, pages 670–675, 2008.