2026 Design, Automation & Test in Europe Conference (DATE 2026)

Multicore Design Verification Directed by Reinforcement Learning

Luiz M. V. Pereira, Diego M. Meditsch, Guilherme O. Campos, Luiz C. V. dos Santos
Federal University of Santa Catarina
Florianópolis, SC, Brazil
luiz.m.v.p@posgrad.ufsc.br; {d.meditsch, guilherme.oliveira.campos}@grad.ufsc.br; luiz.santos@ufsc.br

***Abstract*—A reinforcement learning agent requires proper state approximation when handling partially observable environments. Bounded sequences of action-observation pairs are generally employed to approximate the notion of state, but the choice of representations for actions and observations depends on the specific environment. When targeting the verification environment of a multicore design, observations should capture coherent shared-memory behavior, which can be modeled with order relations capturing properties of the execution of concurrent programs, such as the reads-from (RF) and the coherence (CO) relations. A recent work proposed the use of RF-signatures as observations. However, that relation only provides value observation from loads (all having effect limited to the local core consuming its value), but not from stores (each potentially having effect on all cores sharing the same block, due to the coherence protocol). That is why this paper proposes an agent-directed approach that relies on CO-signatures and combined CO/RF-signatures to improve verification. We evaluated variants of a directed test generator based on the DQN agent under distinct signatures choices for different verification tasks involving 16 and 32-core ARMv8 2-level MOESI designs. Experimental results show that CO/RF-signatures lead to faster median coverage evolution.**

***Index Terms*—Reinforcement learning, design verification, directed test generation, shared memory**

I. Introduction

The functional validation of a multicore chip follows a hierarchical methodology. First, we validate the components of a core, then the core as a whole, and finally the interaction among cores. This last phase includes the validation of the shared memory subsystem, which is the scope of this paper.

Shared memory behavior is defined by a *memory consistency model* (MCM) [1]–[3], which is not a one-to-one golden model, because it combines the complexity of non-determinism, weak consistency, and coherence. Many coverage-based [4]–[18] and litmus-based [19], [20] approaches have addressed that challenge. The recent progress in reinforcement learning [21]–[24] has fostered agent-based approaches [25], [26].

This paper proposes an agent-directed verification approach, and it addresses a crucial issue: the proper state approximation from partial observations of an environment containing a multicore design. It proposes forms of observation serving as proper witnesses to the execution of a suite of concurrent programs.

The text is organized as follows. Section II discusses related work. Section III describes our approach. Section IV explains the key ideas to proper observation. Section V reports our experimental evaluation. Section VI draws general conclusions.

This work was supported in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brasil, under grants PIBIC and PQ (307093/2021-2), and by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brasil, under Finance Code 001.

II. Related work

A. Conventional coverage-based approaches

The first technique proposed to avoid the enumeration of the interactions between processors through shared memory was constrained *Random Test Generation* (RTG). When each thread is independently generated at random (under range constraints), the probability of covering all interactions increases with thread size [4]. Since test programs consist of concurrent threads of execution, multiple behaviors are allowed for the same program, leading to many possible outcomes for the same input stimuli. This can be handled in two ways: (1) the generator produces *self-checking tests* [5], [6], and the non-unique results for each program are predicted with basis on MCM rules; (2) the generator produces *tests without results*, and error diagnosis is done by an external MCM checker [8]–[11], [13], [14], [18].

RTG is very efficient for multicore chip testing, where large test programs can directly execute in the hardware prototype, but not for design verification, where simulated execution largely limits program size. That is why *Directed Test Generation* (DTG) is preferred for simulation-based verification.

Model-based DTG techniques decide about the sequence of tests to generate according to a coverage model [12], [15], [16], and coverage is measured only for tracking purposes. Data-driven DTG techniques [7], [14], [17], however, feed coverage values back to the generator to influence its next decisions.

B. Litmus-based approaches

A litmus test is a self-checking test that can exhibit distinct possible executions. The MCM defines which outcomes are legal and which are forbidden. Litmus tests are often manually written, but a few approaches have automated their synthesis.

The diy tool [19] generates litmus tests by considering specific relaxations of *sequential consistency* (SC) [1]. It assumes an axiomatic modeling style where non-SC executions are encoded as cyclic graphs induced by order relations that define the target MCM. The generator takes a set of edges as inputs, enumerates the possible cycles that can be formed with them, and generates a litmus test corresponding to each cycle. The technique requires the relaxations to be provided as inputs.

A later technique [20] overcame the need to specify which SC relaxations are interesting to validate (as required by diy).

978-3-9826741-1-7/DATE26/© 2026 EDAA

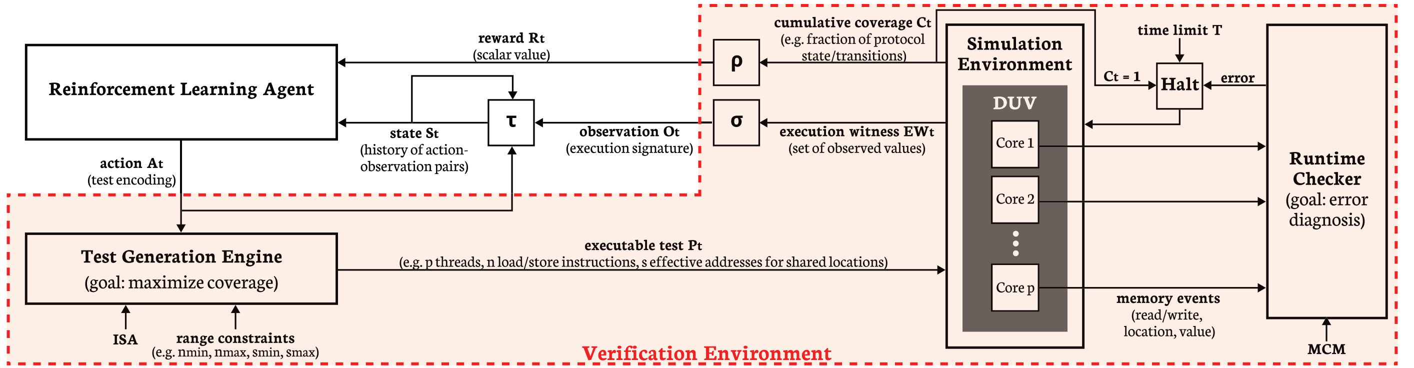


Fig. 1: The agent-based verification approach

It can produce up to one order of magnitude more tests than available in well-know test suites, but the runtime to generate the test suite is super-exponential with the test size bound.

Two relations used by litmus test generation are crucial: the *reads-from* (RF) relation maps each load to the store producing the consumed value [19], and the *coherence* (CO) relation defines the linear order of stores to the same location [20].

C. Agent-directed approaches

Temporal difference [21] and neural networks for value estimation of expected returns [24] gave rise to reinforcement learning agents able to efficiently handle a collection of different problems from a same application domain [22], [23]. This fostered a recent shift towards agent-directed test generation.

An early technique [25] used an agent to direct an RTG engine. A test is represented by a triple of parameters (n, s, k) for constraining RTG. Actions increase or decrease the number of operations (n), shared locations (s), and cache indices (k). The use of RTG parameters to define actions is a severe limitation, because many different random tests can be induced by a given choice of parameters, and they often lead to distinct coverage improvements. This leads to imprecise rewards, reducing the agent's control over coverage evolution. Besides, environment state representation is limited, because it ignores the trajectory of action-observation sequences to approximate the state [24].

A recent work uses a DQN agent [22] for directing test generation. It does not rely on an RTG engine, because it formulates actions as program transformations, which rules out imprecise rewards. It relies on a proper state representation based on the bounded history of action-observation pairs, and it proposes the notion of *execution witness* as an adequate observation for multicore environments. Inspired by a trace compaction technique [27], it imposes constraints on store values so as to produce a unique signature for each witness, thereby repurposing the use of *signatures* to represent environment observations. As a legacy from [27], the RF relation was used to define execution witnesses. However, RF-signatures are limited, because the RF relation only provides value observation from loads (all having effect limited to the local core consuming its value), but not from stores (each potentially having effect on all cores sharing the same block, due to the coherence protocol).

This motivated us to investigate the alternative or the complementary use of the coherence (CO) relation in execution witnesses for improved state approximation.

III. The Agent-Directed Verification Approach

Fig. 1 outlines our approach, which consists of an *agent* interacting with a verification *environment*. A few converters (ρ, τ, σ) are used to translate domain-specific information into the classical reinforcement learning elements at the agent-environment interface (A_t, R_t, S_t), as explained next.

A. The environment

The verification environment contains a test generator, a simulator, and a runtime checker. The generator produces test programs without results [8]–[10], [28] for a target *instruction-set architecture* (ISA), and its goal is to improve coverage. The goal of the runtime checker [11], [13], [18] is to perform error diagnosis according to the axioms of a target MCM.

The generator produces an executable program which complies with range constraints on instruction count and amount of shared locations. That program is executed within a simulation environment containing a design representation of the multicore *device under verification* (DUV). Its execution induces stimuli at the interfaces between each core and private cache controllers. The runtime checker relies on monitors at relevant points of the shared-memory interface for sampling memory events to be analyzed on-the-fly. If they do not comply with the MCM, the checker stops simulation and reports an error.

If the executed test did not expose any error, its contribution to the cumulative coverage is converted into a reward signal sent to the agent, which drives the generator to synthesize another test. A new cycle of verification is launched until some time limit (T) is attained, or full coverage is reached, or an error is found. The goal of the test generator is to produce a sequence of programs, i.e. a test suite, such that the cumulative coverage is maximal within the available time limit.

B. The interaction at the agent-environment interface

The interaction between agent and environment occurs at discrete time steps $0, 1, 2, \cdots, t$. At a given time step t, the agent receives some representation S_t of the environment state, and it responds with some action A_t. In the next time step, the environment sends to the agent a scalar reward signal R_{t+1}, and

evolves to a new state S_{t+1}. Both R_{t+1} and S_{t+1} are determined as a consequence of action A_t. The agent learns from interaction how to take actions on the environment in order to maximize expected rewards. To handle partial observability, we do not assume that the environment emits the actual state, but it exposes some observation O_t of signals that depends on that state. To approximate a Markov state, we rely on the bounded history of action-observation pairs [24].

C. How the environment responds to actions

At time step t, when the environment is in state S_t, the action A_t induces the generation of a test program P_t, which is executed in the DUV. At time step $t+1$, the simulation environment provides a witness EW_{t+1} to the execution of that program, and updates the cumulative coverage C_{t+1}. The variation in coverage between successive time steps is used to define a scalar value serving as a reward R_{t+1}.

The execution witness EW_{t+1} is a set of values observed by monitors placed at relevant points of the design representation. For convenience, the witness is converted into a unique signature serving as an observation O_{t+1} of the environment.

Thus, as a result of multiple cycles of interaction between agent and environment, a sequence of concurrent programs is synthesized, executed, and evaluated in terms of coverage. For a verification episode with a time limit corresponding to T time steps, our approach aims to synthesize a test suite $P_1, P_2, \cdots, P_t, \cdots, P_T$ that maximizes C_{T+1} (as far as no error is diagnosed) or minimizes the time to discover errors.

IV. Key ideas

A. Monitoring the multicore environment

The *reads-from* relation is a partition RF = rfi ∪ rfe, where rfi and rfe capture load-store pairs issued by the same processor (i.e. *internal* pairs) or by distinct processors (i.e. *external* pairs), respectively. The *coherence* relation, also knows as *write serialization* [19], is a partition CO = wsi ∪ wse, where wsi and wse are linear orders capturing store-store pairs issued, respectively, by the same processor or by distinct processors.

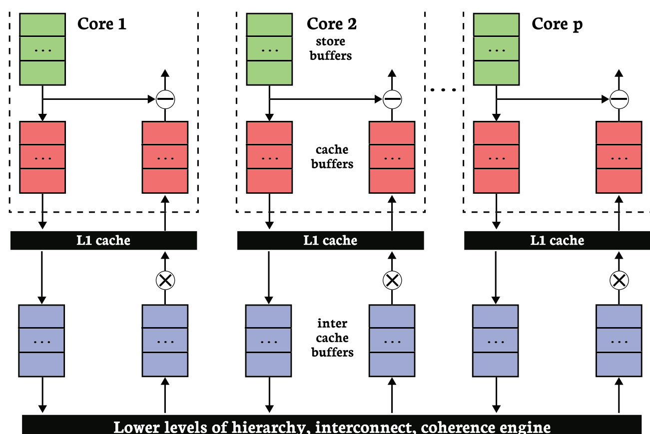


Fig. 2: A partially observable multicore environment

Let us show how pairs in such relations can be inferred from memory events in the DUV. Fig. 2 depicts a partially observable multicore environment where monitors are placed at distinct points of each core to sample values during test execution.

Monitor ⊖ observes either a L1 cache reply or a store buffer reply (implementing the feature read own write early) to a read request. After a miss in the L1 cache, monitor ⊗ will eventually observe a data reply when the owner responds to a request for the intended block. Depending on the owner, such data reply may come from the directory as a immediate response to a GetS or GetM transaction or it may come from another core as an indirect response to a FwdGetS or FwdGetM transaction.

If test generation assigns a *unique* value to each store, the values observed by monitor ⊖ capture RF. To determine which relations are captured by monitor ⊗, note that, for a given location a, it samples one of the following:

- A value consumed by a load L_a^x executing in core x and produced by some store S_a^i executed in a distinct core i. The value written by S_a^i is observed by monitor ⊗ in core x when the block holding it is sent by the owner i to the requester x as a result of a FwdGetS transaction. Thus, the value relates a pair $(S_a^i, L_a^{x \neq i})$ in the mapping rfe.
- A value observed by a store S_a^x that executes in core x after another store S_a^i was executed in a distinct core i. The value written by S_a^i is observed by monitor ⊗ in core x when the block holding it is sent by the owner i to the requester x as a result of a FwdGetM transaction. Thus, the value relates a pair $(S_a^i, S_a^{x \neq i})$ in the linear order wse.

Hence, the *values* observed by monitor ⊗ can be used to capture the relation rfe ∪ wse.

Table I shows how monitors are associated with relations. The last two rows show basic relations captured by the individual monitors. The first row shows the result of their combination. The third column anticipates illustrative examples. The last column assigns a symbol to every observation variant.

TABLE I: Correspondence between monitors and relations

Monitors	Relation	Example	Observations
⊖, ⊗	RF ∪ wse	Fig. 4(a)	♥
⊗	rfe ∪ wse	Fig. 4(b)	♠
⊖	RF	Fig. 4(c)	♦

There is no interest in monitoring wsi, because the order of stores in each thread is defined by program order, which is a property of the program (stimulus), *not* of the DUV (response). Thus, monitoring wsi would lead to useless observations.

B. Witnessing execution instances

Fig. 3 illustrates the notion of execution instances by means of an example. Fig. 3(a) shows a (non-synchronized) concurrent test program with two threads, where capital letters denote shared memory locations, and small letters represent variables allocated in registers. The example assumes that all locations are initialized with zero. Note that the values written by stores are unique. Fig. 3(b), 3(c), and 3(d) show different linear orderings ($\prec$, $<$, and $\ll$) defining distinct execution *instances* of the program in Fig. 3(a). In each core, a trace captures the sequence of memory accesses resulting from the execution of load and store operations. Each access is represented by a triple

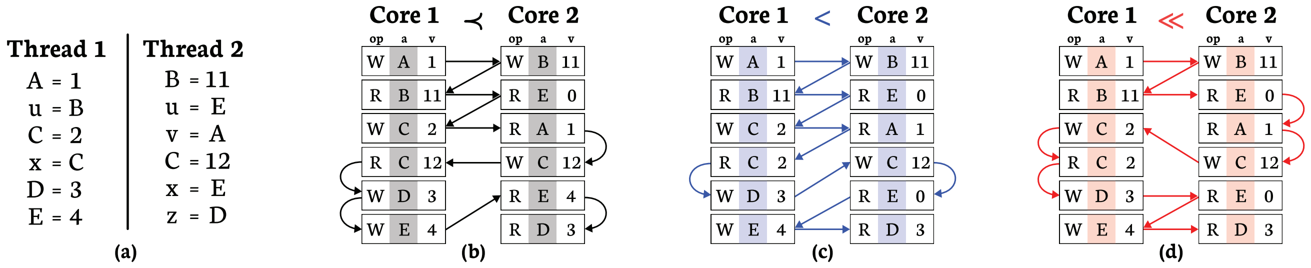


Fig. 3: Three execution instances of a same test program

specifying the type of access as write (op =W) or read (op =R), the location (a), and the value (v).

Fig. 4 shows an example where the monitors specified in Fig. 2 are used to observe values for each execution instance defined in Fig. 3. For simplicity, this example assumes a primitive MSI protocol defining the interaction between a single level of private caches and a directory residing at the second level. The example also assumes that all locations map to different cache lines (so that only compulsory misses are observed at runtime), that each location resides in a distinct cache block (so that no false sharing effects are observed), and that sequential consistency is adopted (so that program order is preserved in each core and stores are atomic operations). However deliberately simple, the example was conceived to capture fundamental properties of real-life coherence protocols.

In Fig. 4, each subfigure corresponds to one of the cases specified by Table I, and each shows scenarios induced by $\prec$, $<$, and $\ll$. Each scenario shows, from left to right, the chain of memory events induced by the thread running in each core. Each (gray or white) box encapsulates an event induced by a single operation of a thread. It indicates whether a memory operation causes a miss (M) or a hit (H), it shows the value observed by each monitor, and it indicates the origin of the value: a data reply from the directory (D), from the owner (O), or from the local cache (C). Colors distinguish values according to their request type: initial value (black), reply value from local cache (green), from FwdGetS (blue), or from FwdGetM (red).

Each scenario shows its set of monitored values. Each set ($EW^i_{\prec}$, $EW^i_{<}$, $EW^i_{\ll}$) serves as a *witness* to the respective execution instance from the perspective of a given core i.

Core 1 | Core 2

(a) RF ∪ wse
Core 1: ≺ EW = {0, 11, 12}; < EW = {0, 11, 2}; ≪ EW = {0, 11, 12, 2}; **EW = {0, 2, 11, 12}**
Core 2: ≺ EW = {0, 1, 2, 4, 3}; < EW = {0, 1, 2, 3}; ≪ EW = {0, 1, 3}; **EW = {0, 1, 2, 3, 4}**

(b) rfe ∪ wse
Core 1: ≺ EW = {0, 11, 12}; < EW = {0, 11}; ≪ EW = {0, 11, 12}; **EW = {0, 11, 12}**
Core 2: ≺ EW = {0, 1, 2, 4, 3}; < EW = {0, 1, 2, 3}; ≪ EW = {0, 1, 3}; **EW = {0, 1, 2, 3, 4}**

(c) RF
Core 1: ≺ EW = {11, 12}; < EW = {11, 2}; ≪ EW = {11, 2}; **EW = {2, 11, 12}**
Core 2: ≺ EW = {0, 1, 4, 3}; < EW = {0, 1, 3}; ≪ EW = {0, 1, 3}; **EW = {0, 1, 3, 4}**

Fig. 4: How values can serve as witnesses to program execution instances

Note that, the bottom of each subfigure displays the union of the witnesses to the set $I = \{\lessdot, <, \ll\}$ from the perspective of a given core i. That summary (EW_I^i) is intended to capture the impact of non-determinism on execution, and the sum of its values is used as a *signature* of a given core. Finally, a multicore signature is used as an *observation* of the environment.

In Fig. 4(a), note that the witnesses in each core are all different. This means that witnesses perfectly distinguish between execution instances. In Fig. 4(b), the witnesses perfectly distinguish between execution instances in core 2, but cannot distinguish between $\lessdot$ and $\ll$ in core 1. In Fig. 4(c), witnesses do not distinguish between $<$ and $\ll$ in core 2, and do not distinguish between any execution instances in core 1.

This example suggests that ♥ and ♠ are more accurate observations than ♦. That is why this paper proposes them as alternatives to approximate environment state information. Indeed, ♦ corresponds to the original witness proposed in a related work [26]. The next section evaluates the proposed observations ♥ and ♠, as compared to that baseline.

V. Experimental evaluation

We reproduced (as closely as possible) the experimental conditions described in a related work [26] where the so-called *Deep-Q Generator* (DQG) was built upon a DQN agent [22]. Although two versions were proposed, we have rebuilt the one named as DQG*, which relies on execution witnesses based on the RF relation. We will henceforth refer to our rebuild of DQG* as DQG♦. As a result, we compared three variants of that generator, denoted as DQG♦, DQG♠, and DQG♥, all with the same set of actions, but with distinct state representations. We compared them with a *Random Agent Generator* (RAG), which selects each action with same probability. RAG serves as a baseline to tell if the variants are learning. We also compared the DQG variants with the *McVersi Test Generator* (MTG) [14].

We evaluated verification tasks targeting 16 and 32-core ARMv8 2-level MOESI designs under different coverage metrics (structural [14] and functional [17]). We constrained the numbers of operations (n_{min} = 1Ki, n_{max} = 4Ki) and shared locations ($s_{min} = 4$, $s_{max} = 128$) for all generators. We allowed them to synthesize sequences of programs with variable number of operations, except for MTG, which requires all programs of a same suite to have fixed size. That is why we report individual results for test suites of fixed-size programs (MTG{1Ki} and MTG{4Ki}). The number of execution instances of each test was set to five, and they were induced as suggested in [14].

We performed independent 3-hour verification episodes for each task. We launched each generator ten times with distinct random seeds. At each new launching, we initialized the neural networks of DQG variants with random weights.

Each verification task relied on the gem5 simulation environment [29]. Runtimes were measured on a Linux Server (AMD Ryzen 9950X, 4.3 GHz, with 192 GB of main memory).

Fig. 5 summarizes our results. The boxes represent samples observed between the first and third quartiles, the horizontal lines represent the median, and coverage is expressed in percentage. Each subfigure shows coverage distributions at the end of successive 1-hour intervals of the verification episode.

To interpret the results, recall that cumulative coverage is an asymptotically increasing function tending to 1 for a sufficiently long sequence of tests. Thus, the *variation* in coverage decreases with runtime along the sequence, leading to diminishing returns that, when negligible, impairs DTG in the long run.

Let us be aware that the structural metric does not distinguish between instances of the same transition between states of different cache controllers, but the functional metric does. Thus, under the functional metric, the total number of transitions grows with core count, but it is constant under the structural metric. Therefore, the number of uncovered transitions under the structural metric decreases faster with time than under the functional metric. That is why the harming effect of diminishing returns on DTG is observed earlier under the structural metric.

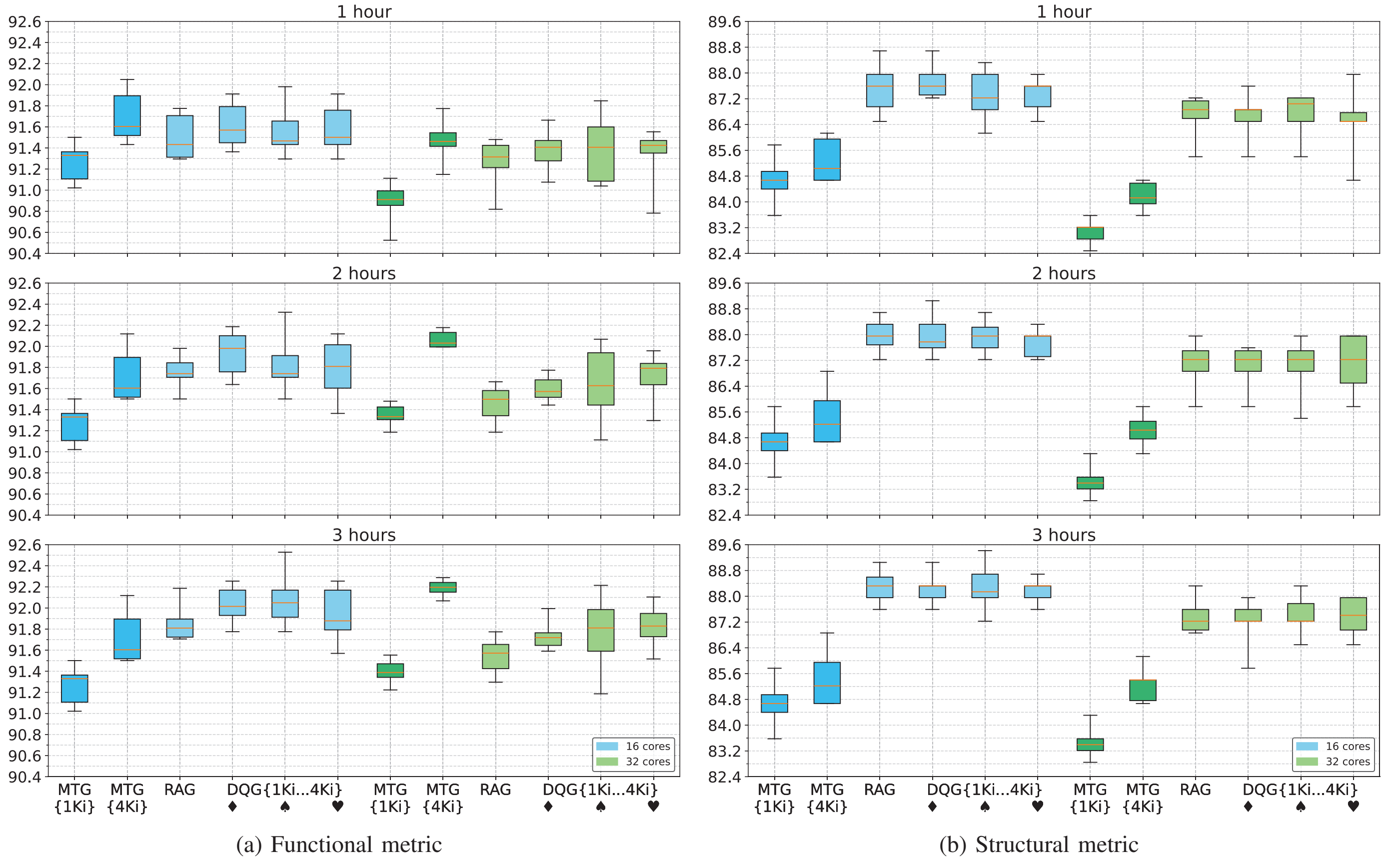
Recall that transitions between states of a coherence protocol are due to two types of events: (1) *evictions* occur when operations access *different* locations competing for the same cache block; (2) *collisions* occur when the *same* shared location is accessed by operations belonging to the same thread (*internal*) or to distinct threads (*external*). Since most protocol transitions are induced by collisions [5], raising their probability tends to increase transition coverage.

Note that the coverage values in Fig. 5 are higher for 16-core DUVs. This behavior can be explained as follows. Given a program size, the number of operations per thread decreases with core count, which raises the probability of external collisions. Since most *protocol* state transitions are induced by external collisions, we should expect larger coverage values for 32-core DUVs, as opposed to what is observed. However, simulation time increases with core count (due to the larger number of events to simulate). This results in a smaller number of *environment* state transitions for the same episode duration. Since the agent is trained with a smaller number of transitions, the learned policy is poorer. That is why comparatively smaller coverage values are observed for 32-core DUVs. However, when doubling the core count, the decrease in coverage due to more limited learning is partially compensated by the the raise in the probability of external collisions, making the attenuation smaller than it would be observed if, instead, the core count was kept constant, but the episode duration was halved.

Consider that, when some variant leads to higher median coverage as compared to RAG, this means that it has actually learned a policy. Notice that the impact of the learned policy on coverage increases along the hours, especially under the functional metric. In most 1-hour intervals, ♥ dominates the other variants or breaks even in terms of median coverage.

Let us interpret that behavior for distinct core counts. For the same constraints on program size, 32-core tasks induce more *external* collisions per test than 16-core tasks, because they handle a larger number of shorter threads. Since it covers rfe and wse (induced by external collisions), ♥ provides more accurate execution witnesses for 32-core tasks. That is why ♥ generally dominates the other variants, especially under the functional metric. Under the structural metric, it takes longer to dominate the others, due to the earlier effect of diminishing returns.

On the other hand, 16-core tasks induce more *internal*



(a) Functional metric

(b) Structural metric

Fig. 5: Coverage distributions at the end of 1-hour intervals of the verification episode

collisions per test than 32-core tasks, because they handle a smaller number of longer threads. Since ♦ covers rfi (induced by internal collisions) and rfe (induced by external collisions), it offers accurate execution witnesses for 16-core tasks. However, since it does not cover wse, it cannot indefinitely outperform other variants. That is why, under the functional metric, ♦ dominates the other variants in the first two hours of the episode, but it is outperformed by ♠ (which covers wse) in the last hour. Under the poorer structural metric, the combination of a smaller number of external collisions and the early effect of diminishing returns ends up making the variants indistinguishable from a random policy.

We conclude that ♥ generally offers more accurate execution witnesses than the other variants, especially as core count grows and when more comprehensive coverage metrics are adopted.

Notice that MTG{1Ki} is inferior to all others in every task, and MTG{4Ki} is also inferior to all, except for the 32-core task under the functional metric. In that case, MTG{4Ki} outperforms all DQG variants. This can be explained as follows. MTG relies on an *evolutionary* algorithm, which makes decisions based only on immediate rewards, and it is not affected by the process of estimating future rewards. DQG is affected due to the limited number of environment state transitions fitting in the episode duration (which is smaller as compared to 16-core tasks). Episode duration restrains on-line training and, therefore, limits learning. Besides, MTG{4Ki} synthesizes suites of fixed-size programs, whereas DQG synthesizes suites of variable-size programs (1Ki, 2Ki, 4Ki). Thus, for 32-core tasks, the smaller programs (1Ki, 2Ki) in the test suite have smaller probability of external collisions, which reduces the cumulative coverage effect of the sequence of programs generated by DQG, as compared to MTG{4Ki}.

To evaluate the DQG variants in the long run, we repeated our experiments for 10-hour episodes. We focused on the 32-core task under the functional metric (the hardest case in the literature targeting 2-level MOESI designs). MTG has outperformed related works [17], [25], [26], and all DQG variants for our 3-hour episodes. However, for our 10-hour episodes, ♠ (92,25%) outperformed MTG{4Ki} (92,21%) in terms of median coverage, and both ♠ (92,43%) and ♥ (92,80%) outperformed MTG{4Ki} (92,28%) in terms of maximum coverage. To our knowledge this is the first time a generator has outperformed MTG in that hardest case.

We conclude that, in the long run, all variants tend unavoidably to similar coverage values due to diminishing returns, but the proposed variants (♠ and ♥) are likely to prevail and be crucial to reach larger coverage.

VI. Conclusions

To make agent-directed verification viable, we proposed proper forms of observing a multicore environment. Experimental results show that they are accurate enough to outperform related approaches [14], [17], [26], especially when targeting large core counts and more comprehensive coverage goals.

REFERENCES

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[2] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 12 1996.

[3] V. Nagarajan, D. Sorin, M. Hill, and D. Wood, *A Primer on Memory Consistency and Cache Coherence, 2nd Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, Feb. 2020.

[4] D. Wood, G. Gibson, and R. Katz, "Verifying a multiprocessor cache controller using random test generation," *IEEE Design & Test of Computers*, vol. 7, no. 4, pp. 13–25, 1990.

[5] A. Adir and G. Shurek, "Generating concurrent test-programs with collisions for multi-processor verification," in *7th IEEE Int. High-Level Design Validation and Test Workshop*, 2002, pp. 77–82.

[6] 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*, vol. 21, no. 2, pp. 84–93, Mar 2004.

[7] S. Fine and A. Ziv, "Coverage Directed Test Generation for Functional Verification Using Bayesian Networks," in *ACM Design Automation Conference (DAC)*, 2003, pp. 286–291.

[8] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A program for verifying memory systems using the memory consistency model," *ACM SIGARCH Comp. Arch. News*, vol. 32, no. 2, pp. 114–123, 2004.

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

[10] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang, *Fast and Generalized Polynomial Time Memory Consistency Verification*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, vol. 4144, pp. 503–516.

[11] O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, and M. Horowitz, "Verification of chip multiprocessor memory systems using a relaxed scoreboard," in *IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*, 2008, pp. 294–305.

[12] 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)*, 2012, pp. 3–8.

[13] W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li, "Linear Time Memory Consistency Verification," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 502–516, Apr 2012.

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

[15] G. A. G. Andrade, M. Graf, N. Pfeifer, and L. C. V. dos Santos, "Steep Coverage-Ascent Directed Test Generation for Shared-memory Verification of Multicore Chips," in *IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, 2018, pp. 29:1–29:8.

[16] Y. Lyu, X. Qin, M. M. Chen, and P. Mishra, "Directed test generation for validation of cache coherence protocols," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 1, pp. 163–176, Jan 2019.

[17] G. A. G. Andrade, M. Graf, N. Pfeifer, and L. C. V. dos Santos, "A directed test generator for shared-memory verification of multicore chip designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 5295–5303, 2020.

[18] M. Graf, G. A. G. Andrade, and L. C. V. dos Santos, "Evecheck: An event-driven, scalable algorithm for coherent shared memory verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 2, pp. 683–696, 2023.

[19] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," in *Computer Aided Verification*, vol. 6174, Dec 2010, pp. 258–272.

[20] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated Synthesis of Comprehensive Memory Model Litmus Test Suites," in *ACM Int. Conf. on Arch. Support for Progr. Lang. and Op. Systems (ASPLOS)*, 2017, pp. 661–675.

[21] G. Tesauro, "Temporal difference learning and td-gammon," *Commun. ACM*, vol. 38, no. 3, p. 58–68, Mar. 1995. [Online]. Available: https://doi.org/10.1145/203330.203343

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop 2013*, 2013.

[23] 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 *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[24] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[25] N. Pfeifer, B. V. Zimpel, G. A. G. Andrade, and L. C. V. dos Santos, "A Reinforcement Learning Approach to Directed Test Generation for Shared Memory Verification," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 3 2020, pp. 538–543.

[26] B. Miranda, L. M. V. Pereira, M. Castro, and L. C. V. Santos, "Multicore Environment State Representation for Agent-Directed Test Generation," in *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[27] D. Lee and V. Bertacco, "Mtracecheck: Validating non-deterministic behavior of memory consistency models in post-silicon validation," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 201–213.

[28] A. Saha, N. Malik, B. O'Krafka, J. Lin, R. Raghavan, and U. Shamsi, "A simulation-based approach to architectural verification of multiprocessor systems," in *Proceedings International Phoenix Conference on Computers and Communications*, 1995, pp. 34–37.

[29] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug 2011.