UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Nícolas Pfeifer

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

Florianópolis
2021

Nícolas Pfeifer

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

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.
Orientador: Prof. Luiz Cláudio Villar dos Santos, Dr.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Pfeifer, Nícolas
A reinforcement learning approach to directed test generation for shared memory verification / Nícolas Pfeifer ; orientador, Luiz Cláudio Villar dos Santos, 2022.
72 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2022.

Inclui referências.

1. Ciência da Computação. 2. Verificação. 3. Memória compartilhada. 4. Geração aleatória de testes. 5. Aprendizado por reforço. I. Santos, Luiz Cláudio Villar dos. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Nícolas Pfeifer

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

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof.ª Cristina Meinhardt, Dr.ª
Universidade Federal de Santa Catarina

Prof. Márcio Bastos Castro, Dr.
Universidade Federal de Santa Catarina

Prof. Sandro Rigo, Dr.
Universidade Estadual de Campinas

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Ciência da Computação.

Prof.ª Patricia Della Méa Plentz, Dr.ª
Coordenadora do Programa

Prof. Luiz Cláudio Villar dos Santos, Dr.
Orientador

Florianópolis, 2021.

ACKNOWLEDGEMENTS

I would first like to thank my supervisor, Luiz C. V. dos Santos, whose expertise helped shape this dissertation. I would like to thank Gabriel A. G. Andrade, Marleson Graf, Bruno V. Zimpel, and Rafael P. Alevato for conceptual discussions and technical help at our research group. I would also like to thank my colleagues at Embedded Computing Laboratory for great conversations during coffee breaks. In addition, I would like to thank my friends and family for their companionship during difficult times. Finally, I could not have completed this dissertation without the unwavering support of my wonderful wife, Ana Paula.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

RESUMO

É esperado que *multicore chips* continuem a utilizar memória compartilhada coerente. Apesar do *hardware* de coerência escalar elegantemente, o espaço de estados do protocolo de coerência aumenta exponencialmente com o aumento do número de *cores*. É por isso que verificação de *hardware* requer geração de testes dirigida (DTG), para o controle dinâmico de cobertura no limite curto de tempo proveniente de lenta simulação e curto tempo disponível para a verificação. É esperado que a próxima geração de ferramentas de *Electronic Design Automation* utilizem aprendizado de máquina para obter valores mais altos de cobertura em menos tempo. Esta dissertação propõe uma técnica que modela DTG como um processo de decisão e busca encontrar uma política de decisão que maximiza a cobertura acumulada como resultado da execução de ações sucessivas por um agente. A técnica proposta utiliza conhecimento consolidado de geração aleatória de testes (RTG), em vez de somente utilizar aprendizado. Ela modela DTG como RTG dirigida por cobertura e explora mecanismos de RTG distintos, sujeitos a limitações progressivamente mais restritivas. Quatro geradores de aprendizado por reforço foram comparados com dois geradores estado da arte baseados em geração híbrida e programação genética. Os resultados experimentais mostram que a aplicação de restrições apropriadas é mais eficiente para guiar o aprendizado para atingir coberturas maiores, do que somente permitir que o gerador aprenda como selecionar os eventos de memória mais promissores para o aumento de cobertura. Para um projeto com 32 núcleos e o protocolo MOESI com dois níveis, a abordagem proposta obteve a cobertura mais alta (94.06%) observada ($p < 0.05$), e foi duas vezes mais rápida para alcançar a maior cobertura obtida pelo gerador híbrido, assim como foi sete vezes mais rápida para alcançar a cobertura máxima obtida pelo gerador baseado em programação genética.

Palavras-chave: *Multicore*. Memória compartilhada. Verificação. Geração aleatória de testes. Aprendizado por reforço. Processo de decisão.

RESUMO EXPANDIDO

Introdução

Multiprocessadores estão presentes em dispositivos pessoais (como, por exemplo, os *smartphones*), em computadores e em servidores usados em *data centers*. Os processadores que compõe um multiprocessador comunicam-se através de memória compartilhada e adotam uma hierarquia de *cache* para diminuir o tempo de acesso à memória. Multiprocessadores em um único *chip* são conhecidos como *multicores*. O uso de caches privadas dá origem ao problema de coerência de cache: para o mesmo endereço de memória, poderiam ser observados valores diferentes em caches privadas distintas. Este problema é geralmente resolvido através do uso de protocolos de coerência, que oferecem ao programador uma visão coerente da memória. O comportamento de um sistema de memória compartilhada coerente é formalmente descrito por um modelo de consistência de memória. Como modelos de consistência de memória mais restritivos limitam muitas otimizações de *hardware*, arquiteturas sofisticadas (como ARMv8, IBM Power9 e RISC-V) utilizam modelos de consistência de memória relaxados. Entretanto, modelos de memória relaxados aumentam o número de execuções válidas de um programa paralelo, tornando mais difícil a validação do projeto de um *multicore*. Apesar de existirem *multicores* com caches não coerentes, espera-se que *multicores* de uso geral continuem a utilizar memória compartilhada coerente. Felizmente, a complexidade do *hardware* que implementa a coerência aumenta moderadamente com o crescimento do número de núcleos de um *multicore*, o que viabiliza o projeto. Infelizmente, o número de estados induzidos por um protocolo de coerência aumenta exponencialmente com o crescimento do número de núcleos, o que desafia a *verificação* do projeto. Devido à limitação imposta pela potência térmica, ocorre uma subutilização dos transistores disponíveis para a fabricação de *multicore chips*, um efeito conhecido como *dark silicon*, que limita o aumento do número de núcleos. Entretanto, apesar dessa limitação, o aumento esperado para o número de núcleos ainda representa um grande desafio para a verificação. Assim, a combinação de consistência relaxada e coerência de cache torna a validação do comportamento de memória compartilhada uma tarefa muito desafiante, a qual requer abordagens específicas. A abordagem utilizada para a validação do protótipo de um *multicore chip* consiste em usar um gerador aleatório de testes para induzir eventos de memória, os quais são então monitorados e analisados por um *checker* independente que verifica os axiomas do modelo de memória. Porém, quando essa abordagem é aplicada em tempo de projeto, a baixa velocidade de simulação e o curto tempo disponível para verificação tornam necessária uma nova abordagem: a geração *dirigida* de testes. Esta dissertação formula a geração dirigida de testes como geração aleatória de testes dirigida por cobertura.

Objetivos

A geração aleatória de testes dirigida por cobertura é formulada como um problema de otimização restringido por tempo da seguinte maneira. *Problema:* Dado um conjunto de comportamentos de memória válidos, encontrar uma sequência de testes aleatórios que maximize a cobertura obtida no tempo disponível. Entretanto, este trabalho não aborda esse problema diretamente. Em vez disso, o problema é modelado como um processo de decisão e o objetivo se torna encontrar uma política de decisão que maximize a recompensa acumulada obtida a partir da execução de ações sucessivas por um agente. Como a probabilidade de transição de estados pela execução de ações é desconhecida, o problema se torna um problema de aprendizado por reforço. As contribuições desta dissertação são as seguintes: (1) Uma nova técnica de geração dirigida de testes baseada em aprendizado por reforço, denominada *Reinforcement Learning*

Generator (RLG). Essa técnica é reutilizável, pois é independente da métrica de cobertura e do protocolo de coerência utilizados. Ela também conta com ações customizáveis que dependem do gerador aleatório de testes escolhido; (2) Uma avaliação de como informações específicas do domínio de aplicação (normalmente modeladas como restrições à geração) podem ser utilizadas para aprimorar o aprendizado.

Metodologia

O ambiente no qual o processo de decisão ocorre contém um gerador aleatório de testes, um simulador de *multicore*, um cronômetro e um analisador de cobertura. O agente que executa ações no ambiente é denominado *Directing Engine*. O analisador de cobertura e o cronômetro criam um estado que representa o ambiente e atribuem uma recompensa a cada ação executada em um dado estado. Como a verificação é restringida por tempo para alcançar objetivos de cobertura, uma representação adequada do ambiente pode ser definida como o par (c,t), em que c representa o valor de cobertura acumulada (quantificada pela métrica de cobertura adotada em cada ambiente) e t representa o tempo necessário para atingir essa cobertura. Para limitar a quantidade de estados possíveis, esses valores são quantizados. Uma vez que o agente interage com o ambiente através do gerador aleatório de testes, as ações são formuladas em termos dos parâmetros do gerador, que são usados para restringir o gerador aleatório de testes. Uma ação deve obter uma melhor recompensa, caso ela produza um maior aumento na cobertura em menos tempo. Portanto, uma função de recompensa adequada pode ser definida como a diferença na cobertura acumulada dividida pelo tempo necessário para executar o teste definido pela ação. Utilizando essa formulação, o objetivo do agente é encontrar uma política de decisão para chegar ao máximo de cobertura no tempo disponível. Foram construídos quatro geradores a partir da técnica proposta (RLG-, RLG+, RLG* e RLG**). Cada gerador foi construído a partir do mesmo *Directing Engine*, mas utilizando um gerador aleatório de testes distinto. As variantes do gerador proposto utilizaram quatro geradores aleatórios de testes, cada um sujeito a restrições progressivamente mais limitantes. O RLG- utiliza um gerador que restringe a quantidade de operações e de variáveis compartilhadas utilizadas nos testes. O RLG+ utiliza um gerador que também restringe a quantidade de operações e de variáveis compartilhadas, porém, utiliza restrições de *biasing* para controlar a ocorrência de *cache eviction*. O RLG* utiliza um gerador com as mesmas restrições dos geradores anteriores, mas emprega restrições extras de *chaining*. E, por fim, o RLG**, além de todas as restrições anteriores, também controla *sharing*, impondo *true sharing* ou habilitando *false sharing*. Os geradores propostos foram comparados com dois geradores dirigidos de teste do estado da arte: *McVerSi Test Generator* (MTG) e *Hybrid Test Generator* (HTG). O simulador utilizado para a representação de *multicores* foi o *gem5*, com 32 núcleos. Para que os resultados não dependessem do protocolo de coerência utilizado, empregamos dois protocolos distintos: MESI e MOESI. Tomando como base os objetivos usuais da verificação de *hardware*, os geradores foram analisados sob os seguintes critérios: aumento da descoberta de erros e da cobertura obtida, bem como a redução do esforço necessário para descobrir erros e alcançar valores de cobertura. Para a averiguação da evolução de cobertura, foram utilizadas duas métricas complementares: estrutural e funcional. Para os experimentos de esforço, foram injetados erros artificiais na máquina de estados do controlador da cache (mudando o próximo estado de uma transição ou impedindo sua ação correspondente).

Resultados e Discussão

Foram executados experimentos em quatro ambientes de verificação distintos. Cada um dos ambientes utilizou um protocolo de coerência (MESI ou MOESI) e uma métrica de cobertura (estrutural ou funcional). Nos experimentos de cobertura, os geradores foram comparados tanto

pela evolução de cobertura quanto pela cobertura final alcançada. A técnica proposta alcançou a maior cobertura em todos os quatro ambientes estudados, além de mostrar uma evolução de cobertura comparável com os outros geradores (MTG e HTG) na maioria dos casos. No ambiente de verificação mais desafiador estudado (MOESI com métrica de cobertura funcional), o RLG+ atingiu uma cobertura de 94,09%, enquanto o HTG atingiu 93,13% e o MTG, 92,54%. Além disso, o RLG+ foi mais de duas vezes mais rápido para atingir a cobertura final do HTG e 7 vezes mais rápido para atingir a cobertura final do MTG. Nos experimentos de esforço de descoberta de erros, o RLG também se mostrou superior ou competitivo com os outros geradores em todos os casos. Para o protocolo MESI, o gerador proposto conseguiu expor todos os erros estudados em 21 minutos, enquanto o HTG precisou de 27 minutos e o MTG, de 3 horas. Para o protocolo MOESI e a métrica funcional, nosso ambiente mais complexo, o RLG+ necessitou de 8 minutos para expor todos os erros (menos um), enquanto o HTG necessitou de 21 minutos e o MTG, 80 minutos. Concluiu-se então que o gerador proposto foi superior na maioria dos casos estudados, especialmente nos ambientes de verificação mais desafiadores. Isso indica que uma técnica de geração dirigida de testes baseada em aprendizado por reforço, quando aliada a restrições de geração apropriadas, tem alta probabilidade de lidar bem com as necessidades de verificação de novos projetos de *multicores*. Além disso, os experimentos indicaram que existe uma relação entre a evolução de cobertura e a velocidade de aprendizado da técnica proposta. Restrições de geração podem influenciar o aprendizado, especialmente quando as restrições afetam o tamanho do espaço de geração. Pois, se por um lado, restrições que levam a espaços de tamanho pequeno podem limitar o agente ao não fornecer a ele uma quantidade suficiente de testes para alcança altos valores de cobertura. Por outro, espaços de geração também podem ser grandes demais, dificultando a evolução de cobertura ao sobrecarregar o agente com uma quantidade de testes muito grande, tornando a tarefa de encontrar bons testes mais difícil. Portanto, a escolha de restrições de geração deve ser feita cautelosamente, especialmente quando se tratando de métodos que utilizam aprendizado.

Considerações Finais

A lenta simulação de *multicores* e o curto tempo disponível para a validação requerem o uso de geração dirigida de testes para o controle eficiente de cobertura. Esta dissertação propõe modelar geração dirigida de testes como geração aleatória de testes dirigida por cobertura. Com base na estimativa de cobertura e no tempo decorrido, a técnica proposta seleciona alguns parâmetros de geração, de forma que maximiza a cobertura no tempo disponível. O gerador proposto utiliza aprendizado por reforço para aprender como selecionar os parâmetros de geração, de maneira a maximizar a evolução de cobertura. Os parâmetros dos testes são selecionados através do uso de ações customizáveis, o que torna o agente adaptável a geradores aleatórios de testes distintos. O agente também é reutilizável para diferentes protocolos de coerência, pois ele somente tem acesso ao percentual da cobertura alcançada através de uma métrica de cobertura. Foram propostas quatro variantes do gerador, cada uma utilizando restrições de geração mais limitantes baseadas em informações específicas do domínio de aplicação. Experimentos foram realizados para comparar o gerador proposto com dois geradores do estado da arte sob três diferentes aspectos: evolução de cobertura, detecção de erros e esforço. Os geradores foram analisados sob dois protocolos de coerência (MESI e MOESI) e duas métricas de cobertura (estrutural e funcional). No ambiente de verificação mais desafiador estudado, o gerador proposto conseguiu atingir uma cobertura final mais alta do que os geradores comparados, assim como foi de 2 a 7 vezes mais rápido para atingir a cobertura final desses geradores. O RLG+ precisou de 8 minutos para descobrir todos os erros (menos um) neste ambiente, enquanto os outros geradores necessitaram de 21 a 80 minutos. Os resultados experimentais mostram que aprendizado por reforço pode gerar uma técnica de geração dirigida de testes efetiva quando ela

utiliza informações específicas do domínio de geração aleatória de testes. Os experimentos também mostraram que a utilização de aprendizado por reforço é altamente aperfeiçoada quando restrições adequadas são utilizadas na geração. O particionamento de tarefas complementares em módulos independentes (um utilizando o conhecido, outro explorando o desconhecido) parece ter a sinergia necessária para a nova geração de ferramentas de verificação. Como trabalho futuro, pretendemos conduzir uma análise sobre o impacto de novas ações, realizar um estudo sobre o impacto do treinamento *online* e propor um novo agente generalizado que tenha um controle mais refinado sobre o processo de geração.

Palavras-chave: *Multicore*. Memória compartilhada. Verificação. Geração aleatória de testes. Aprendizado por reforço. Processo de decisão.

ABSTRACT

Multicore chips are expected to continue 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 Electronic Design Automation tools are expected to exploit Machine Learning for reaching high coverage in less time. This dissertation proposes 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, the proposed 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. Four Reinforcement Learning generators were compared with two state-of-the-art generators based on hybrid generation and 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 2-level MOESI 32-core design, the proposed approach led to the highest observed ($p < 0.05$) coverage (94.09%), and it was 2 times faster than the hybrid generation method to reach the latter's maximal coverage, as well as 7 times faster than the Genetic Programming technique to achieve its maximal coverage.

Keywords: Multicore Chips. Shared Memory. Design Verification. Random test generation. Reinforcement Learning. Decision process.

LIST OF FIGURES

Figure 1 – A coverage-driven RTG approach to DTG. 27

Figure 2 – FSM of the MESI coherence protocol 32
Figure 3 – Example of false sharing occurring in a 2-core processor 33
Figure 4 – An example of a recurrent neural network 35

Figure 5 – Coverage evolution for 3-level MESI using the structural coverage metric . . 50
Figure 6 – Coverage evolution for 2-level MOESI and the structural coverage metric . . 52
Figure 7 – Coverage evolution for 3-level MESI using the functional coverage metric . 53
Figure 8 – Coverage evolution for 2-level MOESI using the functional coverage metric 54
Figure 9 – Impact of constraints on learning for 3-level MESI and given coverage metric 56
Figure 10 – Impact of constraints on learning for 2-level MOESI and given coverage metric 57
Figure 11 – Coverage evolution for 2-level MOESI using the functional coverage metric with extra random seeds 63
Figure 12 – Impact of test size granularity on coverage evolution for 3-level MESI . . . 64
Figure 13 – Impact of time quantization on coverage evolution of 3-level MESI 65

LIST OF TABLES

Table 1 – An illustrative example for the coherence problem 30

Table 2 – RTG Related Work 39

Table 3 – DTG Related Work 40

Table 4 – RL for DTG Related Work 41

Table 5 – Comparison between DTG techniques being evaluated 48

Table 6 – Studied errors for MESI 3-level designs 49

Table 7 – Studied errors for MOESI 2-level designs 49

Table 8 – Effort for finding errors in MESI 3-level designs with both structural and functional coverage metric 58

Table 9 – Effort for finding errors in MOESI 2-level designs with both structural and functional coverage metric 60

LIST OF ABBREVIATIONS AND ACRONYMS

CPU	Central Processing Unit
RTG	Random Test Generator
DTG	Directed Test Generator
MCM	Memory Consistency Model
RL	Reinforcement Learning
LRU	Least Recently Used
FSM	Finite State Machine
GA	Genetic Algorithms
MDP	Markov Decision Process
RNN	Recurrent Neural Network
RLG	Reinforcement Learning Generator
MTG	McVerSi Test Generator
HTG	Hybrid Test Generator
MESI	Modified Exclusive Shared Invalid
MOESI	Modified Owned Exclusive Shared Invalid
mult	$f(i) = 2^i$ test size induction function
sqrt	$f'(i) = \lceil 2^{i/2} \rceil$ test size induction function

LIST OF SYMBOLS

t	Elapsed time
c	Cumulative coverage value
C	Coverage quantization levels
T	Time quantization levels
γ	Quantized cumulative coverage value
τ	Quantized elapsed time value
E	Environment state space
p_j	j-th random test generator parameter
V_j	Set of allowed values for parameter p_j
V	Generation space
v	Allowable setting for RTG parameters
a	Agent action
R_a	Reward for executing action a
n	Number of memory operations
s	Number of shared locations
k	Number of distinct cache sets to which locations can be mapped
f	Enforce true sharing or enable false sharing
N	Set of allowed values for the number of memory operations
S	Set of allowed values for the number of shared location
K	Set of allowed values for parameter k

CONTENTS

1 INTRODUCTION . . . 25
1.1 TARGET PROBLEM AND PROPOSED APPROACH . . . 27
1.2 CONTRIBUTIONS . . . 27
1.3 ORGANIZATION OF THIS DISSERTATION . . . 28

2 BACKGROUND . . . 29
2.1 COHERENT SHARED MEMORY CONCEPTS . . . 29
2.1.1 Cache basics . . . 29
2.1.2 The cache coherence problem . . . 30
2.1.3 Coherence invariants . . . 30
2.1.4 How to maintain cache coherence . . . 31
2.1.5 An example of coherence protocol . . . 31
2.1.6 False Sharing . . . 32
2.2 MACHINE LEARNING CONCEPTS . . . 33
2.2.1 Genetic Algorithms . . . 33
2.2.2 Recurrent Neural Networks . . . 35
2.2.3 Reinforcement Learning . . . 36

3 RELATED WORK . . . 39
3.1 CONSTRAINED RTG LEGACY . . . 39
3.2 DTG FOR SHARED MEMORY VERIFICATION . . . 40
3.3 REINFORCEMENT LEARNING FOR DTG . . . 41

4 THE REINFORCEMENT LEARNING GENERATOR . . . 43
4.1 FORMULATION OF THE DECISION PROCESS . . . 43
4.2 PROPOSED ACTIONS . . . 44
4.2.1 Two-parameter actions . . . 44
4.2.2 Three-parameter actions . . . 44
4.2.3 Four-parameter actions . . . 45
4.3 THE UNDERLYING MODEL . . . 46

5 EXPERIMENTAL VALIDATION . . . 47
5.1 EXPERIMENTAL SET UP . . . 47
5.2 IMPACT OF LEARNING ON COVERAGE EVOLUTION . . . 50
5.3 IMPACT OF PROBLEM-SPECIFIC INFORMATION ON LEARNING . . . 55
5.4 ERROR DISCOVERY RATE AND EFFORT . . . 58
5.5 STATISTICAL SIGNIFICANCE OF FINAL COVERAGE VALUES . . . 62
5.6 IMPACT OF TEST SIZE STEP ON COVERAGE EVOLUTION . . . 63

5.7 IMPACT OF TIME QUANTIZATION ON COVERAGE EVOLUTION . . 64

6 CONCLUSIONS AND PERSPECTIVES 67
6.1 FUTURE WORK 68
6.2 PUBLICATIONS 69

BIBLIOGRAPHY 71

1 INTRODUCTION

Since the dawn of the microprocessor, its advancement is expected at a rapid pace. This belief was synthesized into Moore's law (MOORE, 1965). However, this trend began to die out in the beginning of the 21st century with the end of Dennard scaling (DENNARD et al., 1974) and the thermal design power of CPUs exceeding the dissipation capacity of usual cooling systems. Chip manufacturers, unable to obtain increased performance from higher clock frequencies, changed focus to adding extra processing cores to new projects, adopting multiprocessor designs.

Multiprocessors can be defined as computers consisting of tightly coupled processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space (HENNESSY; PATTERSON, 2017). Multiprocessors are very common, present from personal devices, such as smartphones and personal computers, to servers in data centers. A single-chip multiprocessor is known as a multicore chip.

Most multicore chips adopt cache hierarchies to decrease memory access time. Such hierarchies are usually comprised of one or more levels of private caches, only accessible by a single core, and one last level that is shared among all cores. Private caches give rise to the cache coherence problem: the same memory address may contain different values in distinct private caches. This problem is usually addressed by coherence protocols, which allow the programmer to be provided with a coherent view of memory.

The memory consistency model (MCM) formally describes the behavior of the memory system. It effectively limits the value reads can return (ADVE; GHARACHORLOO, 1996). MCMs define rules regarding the program order requirement (ordering of accesses to distinct addresses), coherence requirement (ordering of writes to the same address), and write atomicity requirement (when a written value can be observed by the issuing core and other cores). The simplest MCM is called *sequential consistency* (LAMPORT, 1979). It fully enforces program order and leads to behavior that works as if every written value becomes available to all cores simultaneously.

Sequential consistency, however, limits many hardware optimizations. Therefore, multiple sophisticated architectures (e.g. ARMv8, IBM Power9, RISC-V) relax sequential consistency in pursuit of faster performance. Relaxed memory models, however, largely increase the amount of valid behaviors of a parallel program execution, making the design validation more difficult.

Even though certain processors used in high performance computing have non-coherent caches (DINECHIN et al., 2013; FRANCESQUINI et al., 2015), most general purpose multicore chips are expected to rely on coherent shared memory (DEVADAS, 2013). Multicore scaling is power limited (ESMAEILZADEH et al., 2011) because of transistor under-utilization (*i.e.* dark silicon). Despite that and the fact that coherence hardware can scale gracefully with increasing core counts (MARTIN; HILL; SORIN, 2012), projected scaling still poses major

challenges to verification because of how the coherence protocol state space grows exponentially with core count.

Multicore functional verification can be divided into three stages: pre-silicon, post-silicon and runtime verification solutions (WAGNER; BERTACCO, 2010). Pre-silicon techniques are employed before any prototype has been made and their major drawback is time, since simulation is very slow. After being sufficiently tested in the pre-silicon phase, a prototype of the processor is created and post-silicon techniques can be applied. Post-silicon techniques can exploit the much higher speed of physical hardware and deliver better correctness guarantees. However, the observability of real hardware is very limited, which leads to only detecting bugs when they generate severe problems (*e.g.* a hang) and time consuming debugging sessions. The last phase is comprised of novel methods which deal with detecting and fixing errors after production. This dissertation focuses on methods regarding pre-silicon verification.

Pre-silicon techniques can be classified into two categories: formal and simulation-based techniques. Formal verification techniques use an abstract model of the hardware and mathematical derivations to prove that certain erroneous behaviors cannot happen. The limitation of formal methods is that they can only be applied to a few small components within the processor, those that can be described using mathematical formulas (ZHANG et al., 2015). Simulation-based verification techniques execute instructions in a simulation of the design and its behavior is checked for correctness. Although these techniques are effective in detecting design errors, they take time, since simulations are orders of magnitude slower than the actual processor. Therefore, only relatively short test sequences can be used.

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 the pre-silicon generation-based techniques fall into two main approaches: litmus test generation (ALGLAVE et al., 2010; LUSTIG; PELLAUER; MARTONOSI, 2014) and random test generation (RTG) combined with memory model checking (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; HU et al., 2012). The first approach exploits a memory model for synthesizing small programs able to expose invalid execution witnesses. Albeit 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 (QIN; MISHRA, 2012) or dynamically (FINE; ZIV, 2003; WAGNER; BERTACCO, 2008; ELVER; NAGARAJAN, 2016; ANDRADE et al., 2020). A pragmatic approach to dynamic coverage control is the casting of DTG as coverage-driven RTG (FINE; FOURNIER; ZIV, 2009). This dissertation's proposal adopts a similar casting.

1.1 TARGET PROBLEM AND PROPOSED APPROACH

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

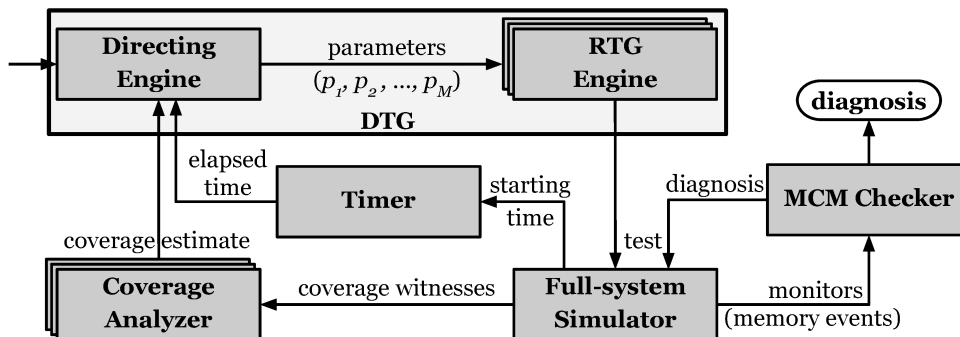
Problem: Given a state space defining valid shared-memory behaviors, find a sequence of random tests that maximize some coverage metric c subject to a time constraint at (available time).

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 provoking state transitions are unknown, the target instance becomes a Reinforcement Learning (RL) problem.

1.2 CONTRIBUTIONS

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 of 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. A runtime checker analyzes monitored events at relevant locations in each core domain in order to verify if the design under verification complies with the expected shared-memory behavior for a given architecture, as specified by an MCM (ADVE; GHARACHORLOO, 1996). Based 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.

Figure 1 – A coverage-driven RTG approach to DTG.



Source – adapted from Andrade et al. (2018)

The main contributions of this dissertation 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)[1].

2. An evaluation of how domain-specific information (properties of shared memory and parallel programs), usually captured as constraints on RTG, can be exploited for improving learning.

This dissertation reuses text of an article by the same author (PFEIFER et al., 2020), however, we present a much expanded experimental validation section and analysis.

1.3 ORGANIZATION OF THIS DISSERTATION

This dissertation presents a novel approach for directed test generation for multicore verification based on reinforcement learning. The rest of this dissertation is organized as follows. Chapter 2 discusses related work and shows how our approach differs from related ones. Chapter 3 describes the proposed technique, RLG, a reinforcement learning directed test generator for multicore verification. Chapter 4 presents experimental validation of our technique when compared to other state-of-the-art generators in multiple verification environments. Chapter 5 puts our conclusions in perspective.

[1] The specific contribution of this dissertation lies in the Directing Engine module. All remaining modules were either created by others (BINKERT et al., 2011; FREITAS; RAMBO; SANTOS, 2013; HANGAL et al., 2004; ANDRADE; GRAF; SANTOS, 2016) or developed in collaboration.

2 BACKGROUND

This chapter explains a few notions required to better understand the scope and the contribution of this dissertation. We first review basic notions required to deal with shared memory events. Then we summarize the concepts related to Machine Learning that are directly used in this dissertation.

2.1 COHERENT SHARED MEMORY CONCEPTS

This section first reviews cache basics, before explaining cache coherence. Then it presents an example of a coherence protocol. Finally, it describes the notion of false sharing, which is widely used in following chapters[1].

2.1.1 Cache basics

Caches have been used for hiding memory latency long before the rise of multicore chips. They are specified by the quantum of data (the number of bytes within a block), by their structure (direct-mapped, set-associative or fully associative), by their replacement policy (e.g. LRU), and by their memory update policy (write through or write back).

A memory location lies in some memory block, which is mapped to a cache block (in direct-mapped caches) or to a *set* of cache blocks (in set-associative caches). Let us consider the general case of a set-associative cache, the others being corner cases (a directed-mapped cache holds a single block in every set and fully-associative cache holds all blocks in a single set). From the point of view of cache controllers, a *memory address* is seen as split into three fields: tag, index, and offset. The offset is used to select the location within a given block. The index is used to find to which cache set the memory block is mapped to. Given an index, the tag is used for an associative search of the block within that cache set. In other words, the *address of a memory block* corresponds to the fields tag and index of a given memory address.

Assume an n-way set-associative cache, and consider the set corresponding to a given index. Suppose that each of the *n* blocks of that set is holding a valid copy of some memory block. In such scenario, suppose that a processor makes a reference to a memory block mapping to that same set. If no cache block inside that set corresponds to the referenced block, one of them is replaced by the new block. We say that a *replacement* event takes place. Each replacement requires a criterion: which cache block of a given set will be replaced?

Replacement policies adopt different criteria to select a block for eviction. The most commonly used policy is called Least Recently Used (LRU), and it chooses to replace the block which has been unused the longest, that is, it tries to keep recently used blocks in the cache. LRU seeks to take advantage of temporal locality: items that have been recently accessed tend

[1] For this review, the author relied on two classic textbooks (PATTERSON; HENNESSY, 2013; SORIN; HILL; WOOD, 2011), and adapted the original text and focus to the needs of this section.

to be accessed again soon. Other replacement policies include first-in/first-out, least frequently used, and random replacement.

2.1.2 The cache coherence problem

When caches are used in multicore chips they usually employ one or more levels of private caches, and a shared last level cache. Since multiple cores may have the same memory block stored in their private caches, they might end up observing different values for the same memory location, as illustrated in the following example.

Table 1 shows an example of the coherence problem for a 2-core chip where each core has its own private cache. Consider the initial in memory value of location X is 0. If core C_0 reads location X from memory, its value (0) gets loaded into C_0's cache. Then core C_1 also reads the same location, so 0 is loaded into its cache as well. Currently, both private caches agree on the value for location X. However, if C_0 writes 1 to X, now its cache and the memory (consider a write-through cache) will have a new value for X, where the same location in the cache of C_1 will still have the value 0 stored. Future reads to location X made by C_1 will return 0, the old value. The memory system is now in an incoherent state.

Table 1 – An illustrative example for the coherence problem

Time step	Event	Cache contents C_0	Cache contents C_1	Memory contents Location X
0				0
1	C_0 reads X	0		0
2	C_1 reads X	0	0	0
3	C_0 stores 1 into X	1	0	1

Source – adapted from Patterson & Hennessy (2013)

2.1.3 Coherence invariants

A memory system is considered coherent if it maintains the following two invariants: *Single-Writer, Multiple-Reader* (SWMR) and Data-Value. The Single-Writer, Multiple-Reader invariant can be defined as follows "For any given memory location, at any given moment in time, there is either a single core that may write to it (and also read from it) or some number of cores that may read from it" (SORIN; HILL; WOOD, 2011). Therefore, there should be no moment in time where a core could write a new value to a memory location and another core could read from it (or also write to it) simultaneously. Another way of looking at the SWMR invariant is to divide a memory location's lifetime into epochs. Each epoch could have, either, a single core with permission to write to the memory location (and also read from it, a read-write epoch), or a number or cores with permission to read from it (read-only epoch).

However, the SWMR invariant does not seem to be enough to keep the system coherent. If, during a read-only epoch, different cores read distinct values for the same location, the system is not coherent. Likewise, the system is incoherent if any core fails to read the last value written to a location in the last read-write epoch. This indicates the need for an extra invariant which governs the propagation of written values between epochs, this invariant is called the Data-Value invariant. The Data-Value invariant can be defined as follows "The value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch" (SORIN; HILL; WOOD, 2011).

2.1.4 How to maintain cache coherence

To enforce the invariants and provide the multicore with a coherent view of memory, cache coherence protocols are implemented in hardware. They govern the interactions between cores, dictating how they need to communicate in order to access a location. If a core wants to read from a location, it sends a message to other cores in order to obtain its current value, as well as to make sure no other core has a read-write permission to that location. These messages effectively end any active read-write epoch and start a read-only epoch. If a core wants to write to a location, it sends a message to other cores in order to obtain its current value, as well as to make sure no other cores have read-only or read-write permission to this location. These messages end any active read-only or read-write epoch and start a new read-write epoch. Protocols that work like this are called invalidate protocols.

Processors usually have distinct instructions for accessing memory at various granularities, typically ranging from 1 to 64 bytes. However, coherence is commonly maintained at the granularity of cache blocks. In other words, the coherence invariants are enforced on the basis of cache blocks instead of individual locations.

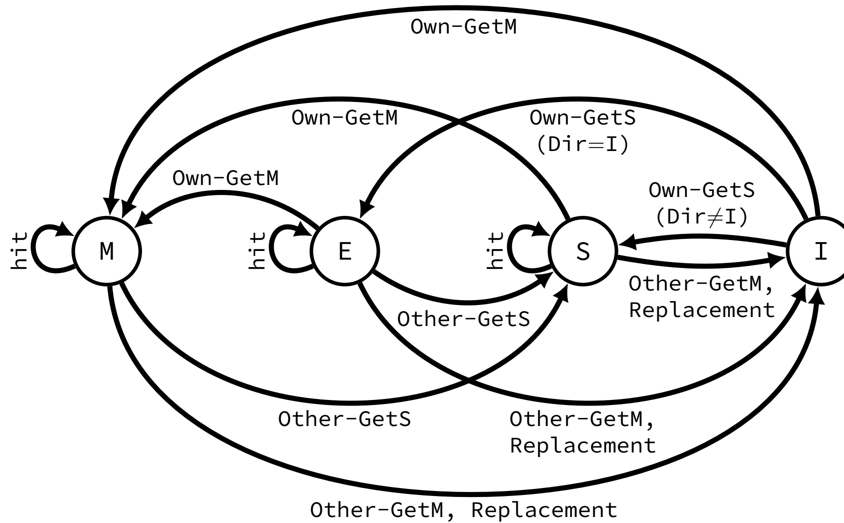
2.1.5 An example of coherence protocol

This section illustrates how a protocol can be used to maintain the coherence invariants. Figure 2 represents a simplified FSM for the MESI coherence protocol. Cache coherence protocols can be specified by a Finite State Machine (FSM) where each transition is associated with output actions. A state of a FSM tracks whether the content of a cache block is valid or not and whether it has read-write or read-only permissions. The FSM controls the interaction between caches of distinct cores and different hierarchical levels.

The MESI protocol contains four states: Modified (M), Exclusive (E), Shared (S), and Invalid (I). Each state controls the access permission a given core has for a memory block, as follows:

- Invalid: no permission (this memory block is not present in this core's cache);
- Shared: read-only (this memory block is present in multiple caches);

Figure 2 – FSM of the MESI coherence protocol



Source – adapted from Andrade et al. (2018)

- Exclusive: read-only (this memory block is present in only a single cache);
- Modified: read-write (this memory block is present in only a single cache)

When a cache block is written by a core, a status bit called *dirty bit* is set. A cache block in states I, E or S has not been modified (its dirty bit is not set). A cache block in state M has been modified (its dirty bit is set). Thus, when write-back caches are used, only transitions leaving state M require update of main memory. A block in state M must be written to main memory when it is about to be replaced or invalidated (transition from M to I) or when it is about to become shared (transition from M to S). Write back actions take place at those transitions.

The FSM in Figure 2 tracks the state of a block in the private cache of a local core. Transitions marked with the *Own* prefix result from accesses generated by that local core, while transitions with the *Other* prefix are induced by accesses from remote cores. *GetM* represents a permission request to write to a memory block, while *GetS* represents a permission request to read from a memory block. A *hit* event represents a successful read or write access to a cached block. *Replacement* events happen after an attempt to access an uncached memory block that maps to an already full cache set, as explained in Section 2.1.1.

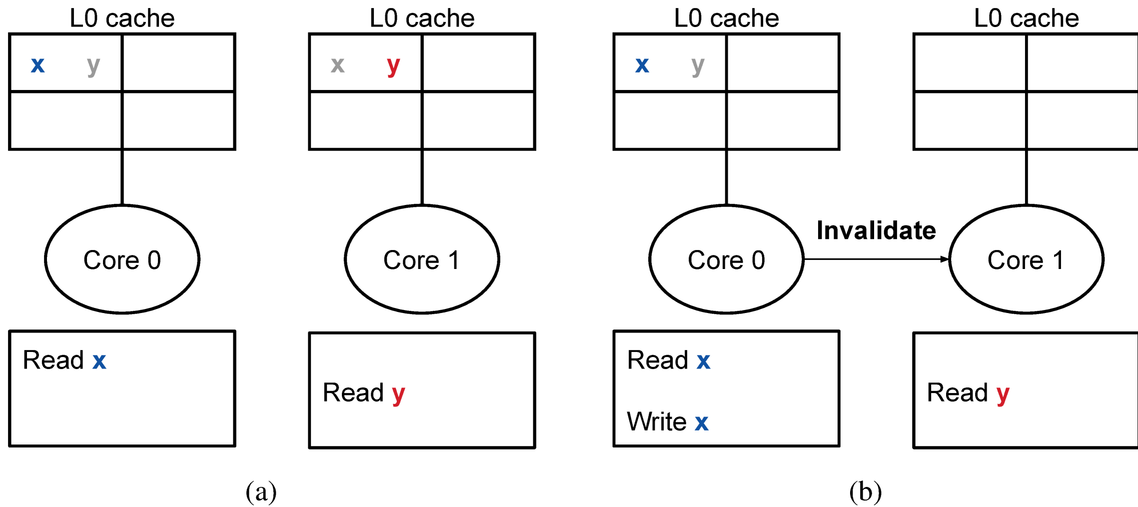
2.1.6 False Sharing

This section discusses a side effect of the block granularity, which is used to maintain coherence. It significantly affects the effectiveness of test generation.

Cache blocks are usually purposefully made bigger than a single variable size in order to exploit spacial locality. Since variables that are allocated closely in memory are usually used together, fetching a whole block from the memory is advantageous because it reduces

the amount of times we have to access the main memory. Bigger cache blocks, however, also increase the probability of a detrimental effect occurring: *false sharing*.

Figure 3 – Example of false sharing occurring in a 2-core processor



Source – the author

False sharing is the unnecessary triggering of coherence operations when distinct cores access different variables closely allocated in memory, sharing a block (BOLOSKY; SCOTT, 1993). Figure 3 contains an example of false sharing occurring in a 2-core processor. After core 0 reads from location *x*, its cache is loaded with the block containing it. This block also contains location *y*, which was allocated closely to *x*. Subsequently, core 1 reads from location *y*, loading its cache with the block containing it and location *x*. Figure 3a represents their caches content at this moment. If core 0 writes to location *x*, core 1 will receive an invalidate message to this block, even though they were accessing distinct locations. Figure 3b depicts the moment false sharing occurred.

2.2 MACHINE LEARNING CONCEPTS

This section focuses on three machine learning techniques. It first reviews genetic algorithms, which underlies one of the related works with which we compare our technique. Then we review recurrent neural networks and reinforcement learning, which are directly used in the technique proposed in this dissertation[2].

2.2.1 Genetic Algorithms

Genetic Algorithms (GA) are techniques whose goal is to make computers learn how

[2] For this review, the author relied on four classic textbooks (KOZA, 1992; SUTTON; BARTO, 2018; GOODFELLOW; BENGIO; COURVILLE, 2016; GRAVES, 2012), and adapted the original text and focus to the needs of this section.

to solve problems without being explicitly programmed. This goal is achieved by emulating Darwinian evolution and natural genetic operations on chromosomes (KOZA, 1992). GA happens in generations, where each generation is comprised of a population of individuals. The next generation is created from the last generation through the use of operations influenced by the Darwinian principle of reproduction and survival of the fittest as well as by naturally occurring genetic operations (*e.g.* genetic mutation and sexual recombination).

Individuals represent possible solutions to the objective problem. Each distinct problem requires a specific way to codify solutions. Solutions are generally represented by a fixed-size string (*i.e.* the chromosome) composed of characters from an alphabet, where each character represents a value for a characteristic of the solution. For example, in the traveling salesman problem (FLOOD, 1956) with 5 cities (A, B, C, D, and E), a solution to this problem might be *ABCDE*, representing the route that starts at city A, goes to B, C, D and finally to E before going back to A. In this example, the string size is 5 and the alphabet size is also 5.

In order to evaluate how well an individual is at solving the objective problem, a fitness function is needed. Fitness functions generally assign a number to individuals and allow us to produce a strict order out of all individuals in a population, this is important so that we can prioritize individuals which represent better solutions to our problem. In the example of the traveling salesman problem, a possible fitness function is the distance of the route the individual represents.

A mutation operation can be used to increase and restore genetic diversity lost from exploitation steps in GA. This operation acts upon a single individual and modifies it to create a new individual. Mutations generally randomly select a character from the chromosome and change it. Mutations should be used sparingly, otherwise they risk interfering with the improvement of fitness.

Genetic algorithms require a *termination criterion*. This function is evaluated at every generation and determines if the algorithm should continue executing or if the result found is considered adequate by the user. For example, in the traveling salesman problem, a possible termination criterion could be to find an individual with fitness less than or equal to 50 or to reach 100 generations.

The crossover operation represents sexual recombination and is used to generate two new individuals by combining the genetic material of a pair of individuals (parents). This operation is usually comprised of an initial random step, where a cutoff point is determine. Using this point to divide the chromosome of the parents, each child receives a part from each parent, generating two new genetic materials.

A few steps of GAs require the proportional selection of individuals based on their fitness. This step is important in order to prioritize high fitness individual but still not totally exclude low fitness individuals because they might have genetic material that is part of the optimal (or near optimal) solution. The selection of individuals takes into consideration if the problem requires us to minimize or maximize fitness.

The population of individuals for the first generation is generated randomly. The cre-

ation of the next generation using the previous one can be done by executing the following steps:

1. Evaluate the fitness function for each individual in the population;

2. Apply a mutation operation to randomly selected individuals from the population;

3. Select a portion of the population to go through crossover, the rest of the individuals will go directly to the next generation;

4. Execute the crossover operation (sexual recombination) on pairs of individuals (parents) selected proportionate to fitness, creating two new individuals for each pair;

5. The new individuals replace their parents in the population and are accompanied by the individuals not selected in step 3 creating the next generation.

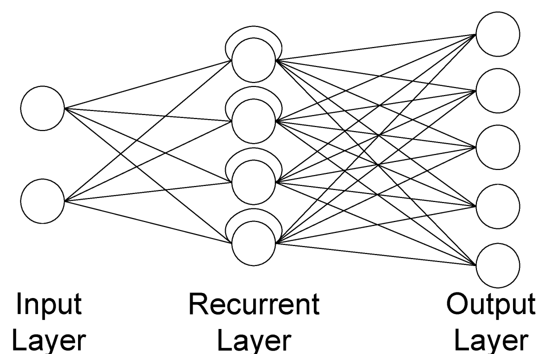
6. Evaluate the *termination criterion* and, if not satisfied, go back to step 1.

Genetic programming, the technique used by McVerSi (presented in Chapter 3), is a *tailoring* of genetic algorithms where individuals are test programs for shared memory verification.

2.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a family of artificial neural networks designed to deal with sequential data. Although regular neural networks can be made to deal with fixed-length sequences, RNNs can learn patterns and generalize through sequences of different lengths, including lengths not seen during training (GOODFELLOW; BENGIO; COURVILLE, 2016).

Figure 4 – An example of a recurrent neural network



Source – the author

The main difference between regular neural networks and recurrent ones is the possibility of cycles in the topology. These recurrent connections implement a type of memory

that allow them to relate the whole history of previous inputs to the output. An example of a simple recurrent neural network is presented in Figure 4. It contains a 2 neuron input layer totally connected to a single self connected hidden layer containing 4 neurons, which is also totally connected to the output layer, with 5 neurons. Other varieties of RNNs have been proposed, such as time delay neural networks (LANG; WAIBEL; HINTON, 1990) and echo state networks (JAEGER, 2001).

2.2.3 Reinforcement Learning

Reinforcement learning is a technique that learns how to maximize a reward obtained by interacting with an environment (SUTTON; BARTO, 2018). Reinforcement learning is a method to solve finite Markov Decision Processes (MDPs).

Markov Decision Processes are comprised of an agent (the learner and decision maker) and an environment (everything that surrounds the agent). The agent selects actions to be executed in the environment, affecting the environment's state. The execution of actions in the environment also presents the agent with a reward signal, a numerical value that represents how well the agent performed according to the goal. This is the value the agent is trying to maximize over time.

Let $\mathtt{S}$ be the set of environment states, let $\mathtt{A}$ be the set of available actions, and let $\mathtt{R}$ be the set of valid reward values. The interactions between agent and environment happen in a sequence of discrete time steps, $t = 0, 1, 2, ...$ At each time step t, the agent receives a representation of the environment's state $S_t \in \mathtt{S}$ and, based on it, selects an action $A_t \in \mathtt{A}$. In the next time step, the agent receives the new state S_{t+1} and a numerical reward $R_{t+1} \in \mathtt{R} \subset \mathbb{R}$.

When the sets of environment states ($\mathtt{S}$), available actions ($\mathtt{A}$), and reward ($\mathtt{R}$) are finite, we have a *finite* MDP. In this setting, the relation between states, actions and rewards can be defined by a discrete probability function, as follows:

$$p(s', r \mid s, a) \doteq Pr(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a),\ \forall s', s \in \mathtt{S}, r \in \mathtt{R}, a \in \mathtt{A}^3$$

The function $p : \mathtt{S} \times \mathtt{R} \times \mathtt{S} \times \mathtt{A} \rightarrow [0, 1]$ denotes the probability of arriving at state s' and receive reward r given the environment is in state s and action a is selected.

The value function of a state represents the reward the agent can expect to obtain over the future when starting from this state. This function represents the long term effect of selecting actions that lead to states and is the function the agent is trying to learn, so that it can obtain the maximum amount of reward in the long run.

States should represent all needed information to predict the environment's behavior. If they do, then the states are said to have the *Markov property*. If they do not, it leads to a partial observability problem. Multiple methods have been devised to handle partial observ-

[3] $Pr(X = x \mid Y = y)$ is used to denote the probability that a random variable X is equal to a particular value x given that the variable Y is equal to the value y.

ability such as partially observable MDPs (ÅSTRÖM, 1965) and predictive state representation (LITTMAN; SUTTON, 2002).

One interesting aspect of reinforcement learning is the trade-off between exploration and exploitation. In order to get reward, the agent has to favor actions that have been shown to produce high reward values. That is, exploit the knowledge learned from previous experience. However, in order to find actions that produce high reward, the agent needs to try new actions. That is, the agent needs to explore the unknown. In order to succeed, it is key that the agent finds a balance between exploiting its prior knowledge and exploring new possibilities.

3 RELATED WORK

This chapter introduces some related work and discusses their methods, achievements, and limitations. We first start by analyzing constrained Random Test Generation legacy methods and which constraints they enforce to improve generation. Then we consider different approaches regarding Directed Test Generation and examine their techniques and drawbacks. Finally, we conclude the chapter by presenting some uses of reinforcement learning for DTG in scopes other than shared memory functional verification and how related work influenced the proposed technique.

3.1 CONSTRAINED RTG LEGACY

Table 2 – RTG Related Work

Work	Generation technique	Exploits:		
		Structural properties of parallel programs	Functional hardware properties	Functional properties of parallel programs
(HANGAL et al., 2004)	Random	✓		
(ADIR et al., 2004)	CSP solver	✓	✓	
(ANDRADE; GRAF; SANTOS, 2016)	Canonical chains	✓	✓	✓

Source – the author

RTG has been used for synthesizing parallel programs for shared memory validation of prototype multicore chips (HANGAL et al., 2004; MANOVIT; HANGAL, 2006). 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 (ADIR et al., 2004) 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 (ANDRADE; GRAF; SANTOS, 2016). 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 (HANGAL et al., 2004). In other words, a chaining constraint enforces *functional properties of parallel programs* for improving test quality.

3.2 DTG FOR SHARED MEMORY VERIFICATION

Table 3 – DTG Related Work

Work	Generation technique
(FINE; ZIV, 2003)	Bayesian network
(WAGNER; BERTACCO, 2008)	Distributed agents
(ELVER; NAGARAJAN, 2016)	Genetic programming
(ANDRADE et al., 2020)	Hybrid generation
This work	Reinforcement learning

Source – the author

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 (FINE; ZIV, 2003). 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 (FINE; FOURNIER; ZIV, 2009).

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 (WAGNER; BERTACCO, 2008) 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, 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.

A more recent approach relied on Genetic Programming for learning how to build new tests from old ones. McVerSi (ELVER; NAGARAJAN, 2016) 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 (WAGNER; BERTACCO, 2008), 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.

The most recently reported method relied on hybrid generation, being both a data-driven and model-based approach. HTG (ANDRADE et al., 2020) consists of a data-driven generation space explorer and a model-based test generation driver. The explorer generates test neighborhoods by selecting tests that contribute to coverage and applying a perturbation function to their parameters. The driver orders the execution of the neighborhood tests so at to favor faster coverage evolution. It was able to both find errors and achieve coverage values faster than McVerSi. As opposed to McVersi, HTG is not limited by a fixed test size and can exploit multiple test sizes to achieve higher coverage values faster.

3.3 REINFORCEMENT LEARNING FOR DTG

Table 4 – RL for DTG Related Work

Work	Scope
(SHAKERI et al., 2010)	Logic level **hardware** verification
(GROCE, 2011)	Test synthesis in **software** validation
(KIM; KWON; YOO, 2018)	Data generation in **software** validation
This work	Shared memory verification

Source – the author

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 (SHAKERI et al., 2010). 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 (GROCE, 2011). 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 (KIM; KWON; YOO, 2018) 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 (SHAKERI et al., 2010; GROCE, 2011; KIM; KWON; YOO, 2018) led to small improvements over random generation/search. RTG approaches that exhibited good results relied on domain-specific properties (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; ADIR et al., 2014; ANDRADE; GRAF; SANTOS, 2016). This indicates that RL should rely on the same types of properties for improving test quality, as did previous learning approaches (FINE; ZIV, 2003; WAGNER; BERTACCO, 2008; FINE; FOURNIER; ZIV, 2009; ELVER; NAGARAJAN, 2016). The next section shows how our proposal bridges that gap.

4 THE REINFORCEMENT LEARNING GENERATOR

This chapter describes our proposed method, the Reinforcement Learning Generator (RLG). First, we present the verification environment the RLG is a part of, followed by the decision process it wants to optimize. Then we report four sets of proposed actions which define how the agent can interact with the parameters in order to control test generation. Finally, we detail the learning technique used for the underlying model and list a few of the hyperparameters used.

4.1 FORMULATION OF THE DECISION PROCESS

The *environment* includes an RTG engine, the simulator, a Timer, 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 can be defined as the difference in the cumulative coverage divided by the time needed to execute the test defined by the action. That is, the reward for an action a is $R_a(v,v') = (c'-c)/(t'-t)$.

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

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

4.2 PROPOSED ACTIONS

The set of actions is largely dependent on the adopted RTG engine. For instance, conventional RTG engines (HANGAL et al., 2004; MANOVIT; HANGAL, 2006) 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 (ANDRADE; GRAF; SANTOS, 2016) employ a third parameter: the number of distinct cache sets to which locations can be mapped (k). Finally, a fourth parameter (f) is employed to either enforce true sharing or enable false sharing, similarly to what is supported in biased generators (ADIR et al., 2004; ANDRADE et al., 2018). Without loss of generality, this dissertation defines actions involving only the parameters mentioned above[2], which cover the majority of reported 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}).

4.2.1 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 function[3] $f(i) = 2^i$, as follows:

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

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

We define the following actions:

- $a_1(n,s) = (2n,s)$
- $a_2(n,s) = (n/2,s)$
- $a_3(n,s) = (n,2s)$
- $a_4(n,s) = (n,s/2)$

4.2.2 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

[2] We define actions that only modify a single parameter at a time in order to limit the combinatory aspect of having multiple parameters, which could hinder learning if a large number of actions was used.

[3] This could be replaced by other functions without loss of generality, as far as actions are accordingly adjusted.

for each allowed value of s, and are constrained to be multiples[4] of that value, as follows:

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

We define the following actions:

- $a_1(n,s,k) = (2n,s,k)$
- $a_2(n,s,k) = (n/2,s,k)$
- $a_3(n,s,k) = (n,2s,k)$
- $a_4(n,s,k) = (n,s/2,k)$
- $a_5(n,s,k) = (n,s,2k)$
- $a_6(n,s,k) = (n,s,k/2)$

4.2.3 Four-parameter actions

In addition to the three parameters in the previous defined actions, these actions also control the occurrence of false sharing. False sharing occurs when processors in a shared-memory system make references to different locations within the same memory block, thereby inducing unnecessary coherence operations (BOLOSKY; SCOTT, 1993).

False sharing is controlled by the use of a new parameter: $f \in \{True, False\}$. If $f = False$, false sharing is activated and the maximum amount of locations is allocated in the same cache block. If $f = True$, no two shared variables address will belong to the same cache block, enforcing true sharing. In order to allow the agent control over sharing, a single new action is added to the previously proposed 6-action model. This new action toggles the value of f while keeping all the other parameters unaltered.

We define the following actions:

- $a_1(n,s,k,f) = (2n,s,k,f)$
- $a_2(n,s,k,f) = (n/2,s,k,f)$
- $a_3(n,s,k,f) = (n,2s,k,f)$
- $a_4(n,s,k,f) = (n,s/2,k,f)$
- $a_5(n,s,k,f) = (n,s,2k,f)$
- $a_6(n,s,k,f) = (n,s,k/2,f)$
- $a_7(n,s,k,f) = (n,s,k,\neg f)$

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

Such control over false sharing could expose new behaviors from the coherence protocol, since this property is not being exploited in any other variant.

4.3 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 networks are a viable option for partially observable Markov Decision Processes, because their ability to handle time and memory makes them suitable for modeling any type of dynamical system (DUELL; UDLUFT; STERZING, 2012).

After comparing multiple network topologies and hyperparameters based on loss evolution (similar to Figure 9), we opted for an RNN with a single 10-neuron recurrent layer between fully-connected input and output layers, and 11 distributional RL atoms[5]. Since an RNN is trained with sequences, we used subsequent RL transitions[6] for training[7]. 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[8] is an adaptation of the Rainbow agent (HESSEL et al., 2018), but the original deep neural network was replaced by our RNN.

[5] 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* (HESSEL et al., 2018).

[6] An RL transition is essentially a transition between environment states that was induced by a given action and was assigned a given reward (HESSEL et al., 2018).

[7] Since RNNs employ hidden states to implement recurrence, it is convenient to correlate such states with environment states for the sake of training. We exploit this feature during our online training.

[8] The Directing Engine used in the proposed generator was implemented using Python 3 and PyTorch.

5 EXPERIMENTAL VALIDATION

This chapter describes the experiments performed in order to validate the proposed technique by comparing it with other state-of-the-art generators. We begin by describing the experimental set up which is comprised of the ranges of generation parameters, compared generators, processor simulator, coherence protocols, coverage metrics, and definition of the studied errors used in the detection and effort experiments. Then we compare the generators under coverage evolution, error detection and effort in four different combinations of coherence protocols and coverage metrics. Next, we discuss the impact of test generation constraints in the learning process by analyzing the loss (prediction error) of all four of the proposed generator variants. Finally, we present experiments on the statistical significance of the final coverage values, in addition to an analysis of the impact of test size step and time quantization on coverage evolution.

5.1 EXPERIMENTAL SET UP

Four generators were built under the proposed approach. To build each Reinforcement Learning Generator, we used the same Directing Engine and selected a distinct RTG engine[1]. We selected four 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 (HANGAL et al., 2004; MANOVIT; HANGAL, 2006). RLG+ relies on an RTG engine that not only constrains operations and locations, but also employs *biasing* constraints for controlling cache evictions, similarly to (FINE; ZIV, 2003). RLG* relies on an RTG engine that enforces the same constraints as the previous ones, but imposes extra *chaining* constraints, similarly to (ANDRADE; GRAF; SANTOS, 2016). RLG** relies on an RTG engine that enforces the same constraints as RLG*'s, but controls sharing, either enforcing true sharing or enabling false sharing. We set the same ranges for their common parameters: $n_{min} = 1Ki$, $n_{max} = 64Ki$, $s_{min} = 4$, and $s_{max} = 128$. The parameter f is always $True$ for all generators, except for RLG**.

We compared the proposed RLGs with two state-of-the-art directed test generators, McVerSi (ELVER; NAGARAJAN, 2016) Test Generator (MTG), which is available in the public domain (ELVER, 2016) and the Hybrid Test Generator (ANDRADE et al., 2020) (HTG). We preserved all MTG's genetic parameters exactly as they were originally set in (ELVER; NAGARAJAN, 2016). 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

[1] The Directing Engine and RTG Engine modules are defined in the verification framework depicted in Figure 1

in (ELVER; NAGARAJAN, 2016)[2]. HTG was executed using the same generation space and RTG as RLG*, which has 3 generating parameters and imposes chaining constraints, as HTG's original RTG[3]. Table 5 compares the selected DTG techniques being evaluated.

Table 5 – Comparison between DTG techniques being evaluated

DTG	Dynamically exploits:				
	Test size	Location amount	Biasing	Chaining	False sharing
MTG		✓			
HTG	✓	✓	✓	✓	
RLG-	✓	✓			
RLG+	✓	✓	✓		
RLG*	✓	✓	✓	✓	
RLG**	✓	✓	✓	✓	✓

Source – the author

We relied on gem5's infrastructure (BINKERT et al., 2011) 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) MOESI or a 3-level (L0, L1, L2) MESI directory protocol with 4KB (direct-mapped) private caches at L0, 64KB (2-way) private caches at L1, and a 2MB (8-way) shared L2 cache, all with the same block size (64 bytes). We relied on a checker similar to (FREITAS; RAMBO; SANTOS, 2013).

Since the usual goals of functional hardware verification are to improve error discovery and achieved coverage, as well as reducing the effort needed to detect errors and reach coverage values, we evaluated the generators under these criteria.

To capture coverage evolution, we relied on two complementary coverage metrics: structural and functional. The structural metric is defined in (ELVER; NAGARAJAN, 2016), it 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. The functional coverage metric (defined in (ANDRADE et al., 2020)), on the other hand, tracks each transition with respect to the core that executed the instruction that covered it. It avoids counting the whole exponential state space by correlating FSM transitions and cores, limiting the amount of total tracked events but still allowing representation of interactions between distinct cores. We report a coverage value as the fraction of events (FSM transitions for structural and transition core pairs for functional) covered after the execution of a sequence of tests. The agent relies on

[2] The MTG was built by replacing both components of the DTG module (the Directing and the RTG engines) within the verification framework depicted in Figure 1, all other components remained the same.

[3] The HTG was built by replacing both components of the Directing Engine within the verification framework depicted in Figure 1, all other components remained the same.

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. In order to establish the most appropriate amount of time quantization levels, we performed an experimental comparison (reported in section 5.7) and concluded that using T=5000 was most appropriate for our setup. We arbitrarily selected C=100 levels for coverage quantization.

Table 6 – Studied errors for MESI 3-level designs

ID	State	Input event	Next state	Precluded output action
e1	IS_I	Data_all_Acks	I	writeDataFromL2Response
	IS	Inv	IS instead of IS_I	(preserved)
e2	SM	Data_all_Acks	M	(preserved as in (IM, M))
	SM	Data	SM	(preserved as in (IM, SM))
e3	IS_I	DataS_fromL1	I	writeDataFromL2Response
e4	S	L0_Invalidate_Own	SS instead of S_IL0	forward_eviction_to_L0
e5	E_IL0	WriteBack	MM_IL0	writeDataFromL0Request

Source – adapted from Andrade et al. (2020)

Table 7 – Studied errors for MOESI 2-level designs

ID	State	Input event	Next state	Precluded output action
E1	SI	Writeback_Ack_Data	I	Data block in sendData
E2	ILXW	L1_WBDIRTYDATA	M	writeDataToCache
E3	OM	Fwd_GETS	OM	Data block in sendData
E4	ILOXW	L1_WBCLEANDATA	M	writeDataToCache
E5	SM	Fwd_GETS	S	Data block in sendData
E6	M	Fwd_GETX	I	Data block in sendDataExclusive

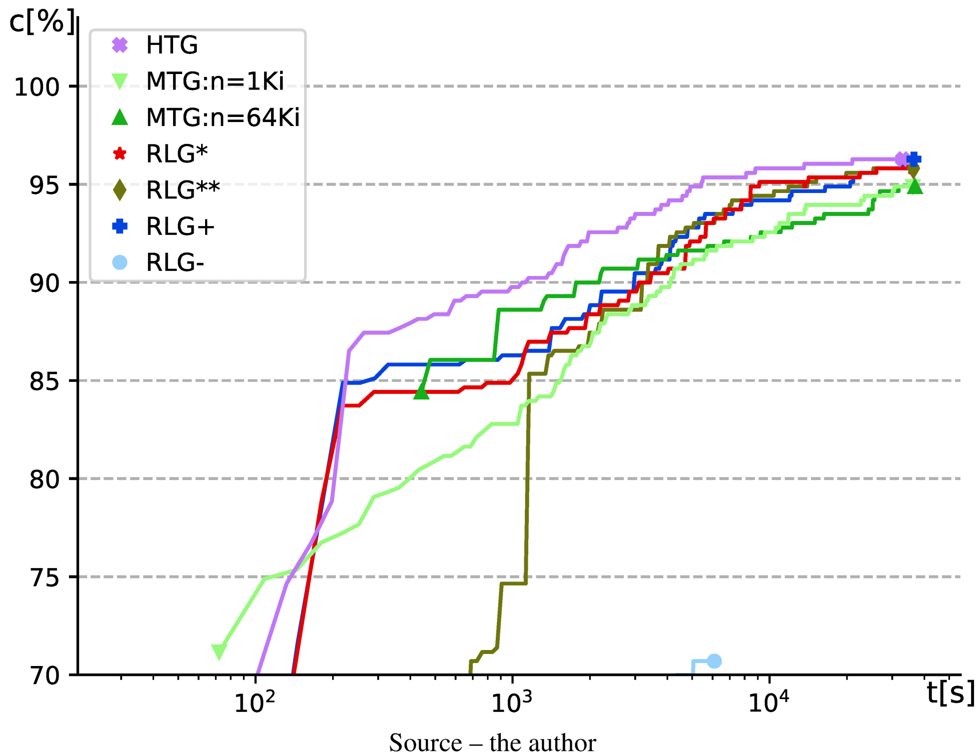
Source – adapted from Andrade et al. (2020)

For effort experiments, we injected artificial errors into the cache controller's FSM (either changing the next state of a transition or precluding an output action). The faulty designs are described in Tables 6 and 7. To determine the effort each generator needs to expose each error, we ran the generators in designs with a single injected error until the error was exposed, the generation space was exhausted or a 10-hour time limit was reached. Each generated test was executed 5 times under different simulation states (not related to the test itself) in such a way that the distinct executions of each test are all perturbed differently (ELVER; NAGARAJAN, 2016). To obtain the reported coverage values, we executed each generator with 10 distinct

random seeds and computed the median value of the resulting distributions. All runtimes were measured on an HP xw8600 Workstation (Intel Xeon E5430 2.66 GHz, 8GB RAM).

5.2 IMPACT OF LEARNING ON COVERAGE EVOLUTION

Figure 5 – Coverage evolution for 3-level MESI using the structural coverage metric



Source – the author

Since the initial verification phase usually reflects the coverage of events that are easy to stimulate using random generation, we focus our analysis on the intermediate and final phases. Consequently, we only show coverage values higher than 70% and trim the graphs accordingly.

Figure 5 shows the coverage evolution for a 3-level MESI design using the structural coverage metric. It represents the median coverage obtained among 10 random seeds. 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 5.3). RLG+, RLG*, and RLG** are not only superior to RLG-, but also competitive with MTG and HTG. This indicates that the exploitation of hardware properties and functional properties of parallel programs is able to improve the quality of the generated tests under RL.

Notice that RLG** reached the same final coverage that RLG* reached (95.81%). Therefore, controlling sharing did not enable the discovery of new coverage events in this verification environment. Also, RLG**'s efficiency throughout the experiment was lower than both RLG*'s and RLG+'s, taking longer to achieve the same coverage values. This can be ex-

plained by how adding an extra parameter expanded the generation space. RLG**'s generation space has two valid parameters settings for each setting in the 3-parameter space: one with false sharing enabled, another with true sharing enforced. The expanded generation space makes the decision process harder to solve because of the higher amount of possibilities available. This indicates that, although more parameters gives the agent more control over generation, it also makes the task of finding better tests harder. Albeit this experiment might suggest that true sharing should always be enforced, this can preclude the detection of design errors that require false sharing to be detected, as can be seen in section 5.4.

Note that the behavior of RLG* and RLG+ is similar up to around the 2-hour mark. From around 2 hours to 6 hours into the experiment, the tighter constraints exploited by RLG* allowed it to obtain a higher coverage than all the other RLG variants. However, after that period, their behavior was again similar up to 9 hours, when RLG+ was able to achieve a final coverage higher than RLG*'s, 96.28% against RLG*'s 95.81%. This indicates that, even though the tighter constraints helped to increase RLG*'s efficiency, they did not enable the discovery of new transitions in this verification environment.

Observe that, for 64Ki, MTG is better than its 1Ki counterpart until around the 3-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. Although shorter tests allowed a higher efficiency for some time, both MTG variants achieved the same final coverage of 94.88% for 3-level MESI and structural coverage.

RLG*'s final coverage was 95.81%, while MTG's was 94.88%. However, MTG (with $n = 1Ki$) took around 30.500 seconds to reach its highest coverage, while RLG* took around 8400 seconds to reach that same coverage, i.e. 3.6 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 a higher final coverage[4].

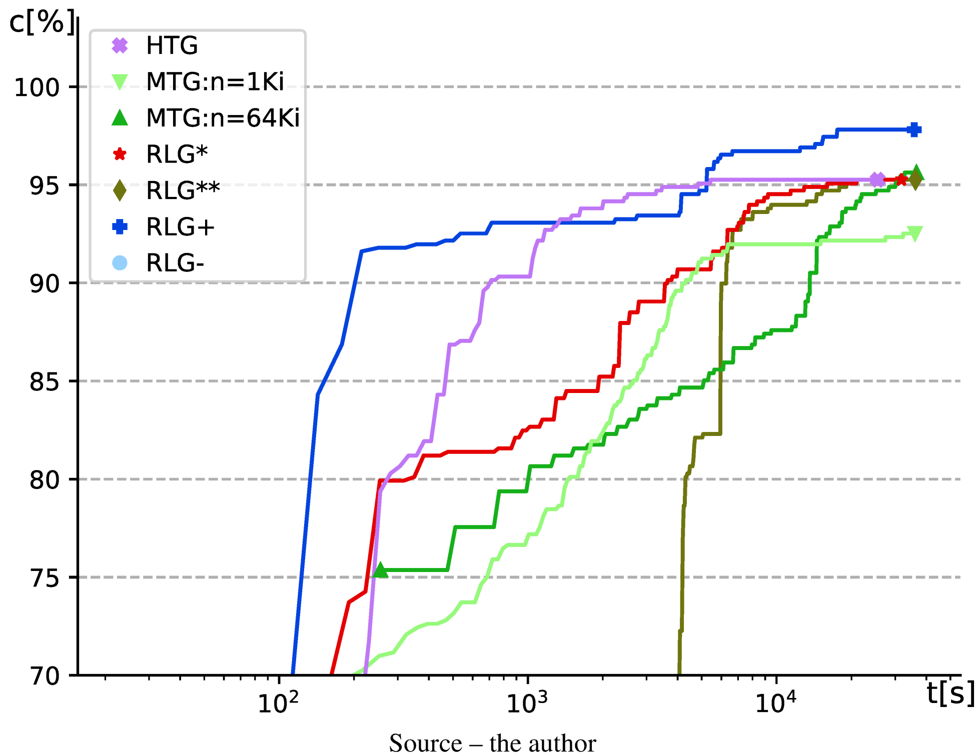
HTG achieved its final coverage 60% faster than RLG+, however, they still achieved the same final value. Therefore, HTG's hibrid generation approach did not allow it to discover coverage events that were not discovered by RLG.

Figure 6 represents the coverage evolution of the selected generators for a design using a 2-level MOESI coherence protocol and the structural coverage metric. When comparing to 3-level MESI, the generators behaved very differently. The difference of evolution speed among generators is more noticeable, as well as more distinct final coverage values.

Let us first consider the RLG variants. RLG-, for instance, reached the lowest final coverage value (46.35%) and does not appear in the zoomed-in plot. This can be attributed to the same aspect that led it to also achieve the lowest coverage in 3-level MESI, namely, unsuitable generation constraints. Additionally, as MOESI is a more complex protocol, it is even harder to cover events. This indicates that generators that only exploit structural properties

[4] 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.

Figure 6 – Coverage evolution for 2-level MOESI and the structural coverage metric



Source – the author

of parallel programs are not suitable for verifying complex multicore processors, especially with intricate protocols such as MOESI. Concerning the two best performing variants, RLG+ and RLG*, their behavior was very different, especially when comparing with how closely they behaved in 3-level MESI. This can be explained by their underlying test generators. In order to enforce the chaining constraint, the RLG*'s generator uses memory barriers. Even though barriers increase the chance of executing the intended operation chains, they could also limit the amount of possible execution witnesses, which could prevent the coverage of some harder to reach events, especially in more complex protocols like MOESI.

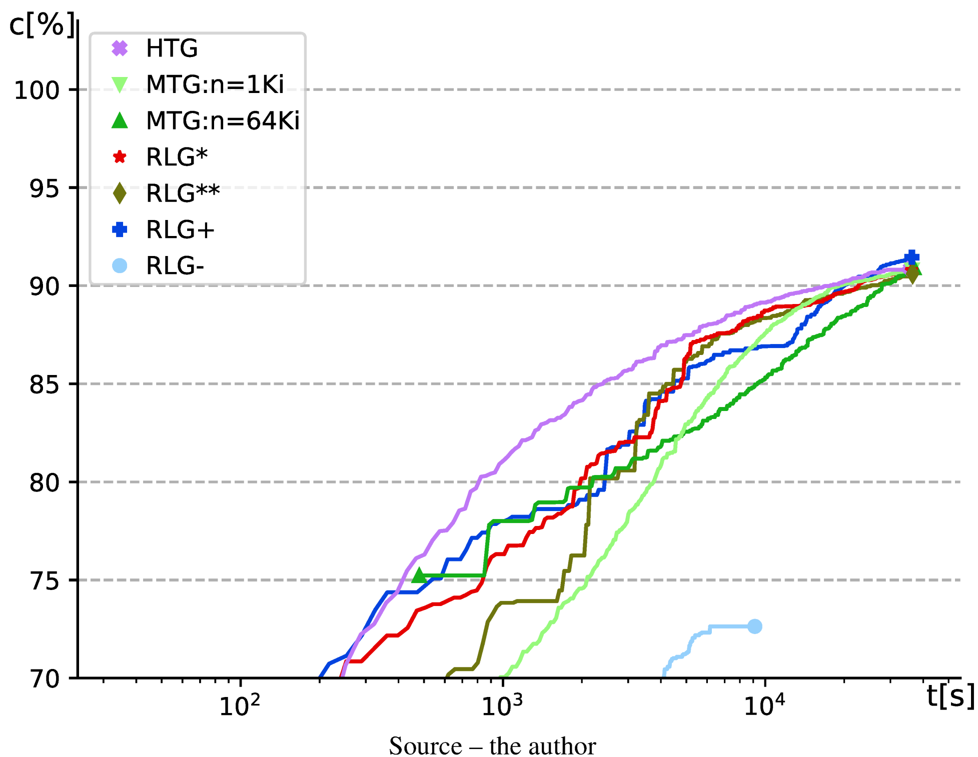
Regarding RLG**, it behaved similarly in the designs using both protocols. It had a poorer coverage evolution than RLG* until around 2 hours into the experiment, then it tied with RLG* until the end of the experiment. Therefore, the control over sharing hampered coverage evolution in the initial verification phase when compared to the other variants (which do not control sharing).

Let us now compare the MTG behavior with different test sizes. Similar to what was observed for MESI, MTG with shorter tests (1Ki) prevailed over its longer-test counterpart for a period of the verification. However, it stagnated after around 2 hours and was surpassed by its 64Ki version around the 5-hour mark. This can be explained by the higher complexity of the MOESI protocol when compared to MESI. Even though the shorter tests allowed MTG to maximize the amount of non-determinism in short tests through its genetic algorithm, it soon hit the maximum non-determinism possible using this amount of instructions. A longer test is required to increase it further, and since the MTG uses a fixed test size, it could not adapt to

this scenario. This indicates that, as the complexity of the coherence protocol increases, test generators need to pay increasing attention to test size and be able to use multiple values to allow for fast coverage evolution and achieve high final coverages.

Albeit both RLG and MTG rely on learning, their different approaches lead to different results. Even though MTG (with $n = 64Ki$) achieved a slightly higher final coverage, 95.62%, against RLG*, RLG** and HTG's 95.25%, it had a slower coverage evolution until the very end of the experiment. One main difference between these generators and MTG is the dynamic test size selection. The lack of a dynamic selection hampered MTG's coverage evolution (for MTG with $n = 64Ki$) and limited the final coverage obtained (for MTG with $n = 1Ki$).

Figure 7 – Coverage evolution for 3-level MESI using the functional coverage metric



Source – the author

Let us now analyze how the generators fared under the more descriptive functional coverage. Figure 7 represents the coverage evolution of the selected generators for a design using a 3-level MESI coherence protocol and the functional coverage metric.

The functional coverage metric is a more expressive metric than the simpler structural one because of how it tracks, alongside covered transitions, the processor responsible for executing the instruction that triggered its coverage. Hence, it is harder to achieve high coverage in this metric because of the higher amount of tracked events. This can be observed in the coverage graphs as a lower final value combined with a slower coverage evolution for all generators. When comparing the generators, it is possible to notice that their behaviors differed less than on previous experiments, their final values and coverage evolutions were similar. This can be explained by the lower amount of saturation achieved by the generators in this experiment when compared to the structural ones. Since the amount of total coverage events increased, but the

overall ratio between easier and harder-to-stimulate transitions was maintained, generators need more time to cover all easier-to-stimulate transitions, needing more time to reach post-saturation coverage levels. Differences between generators behavior is better seen after saturation, since distinct techniques achieve distinct saturation levels.

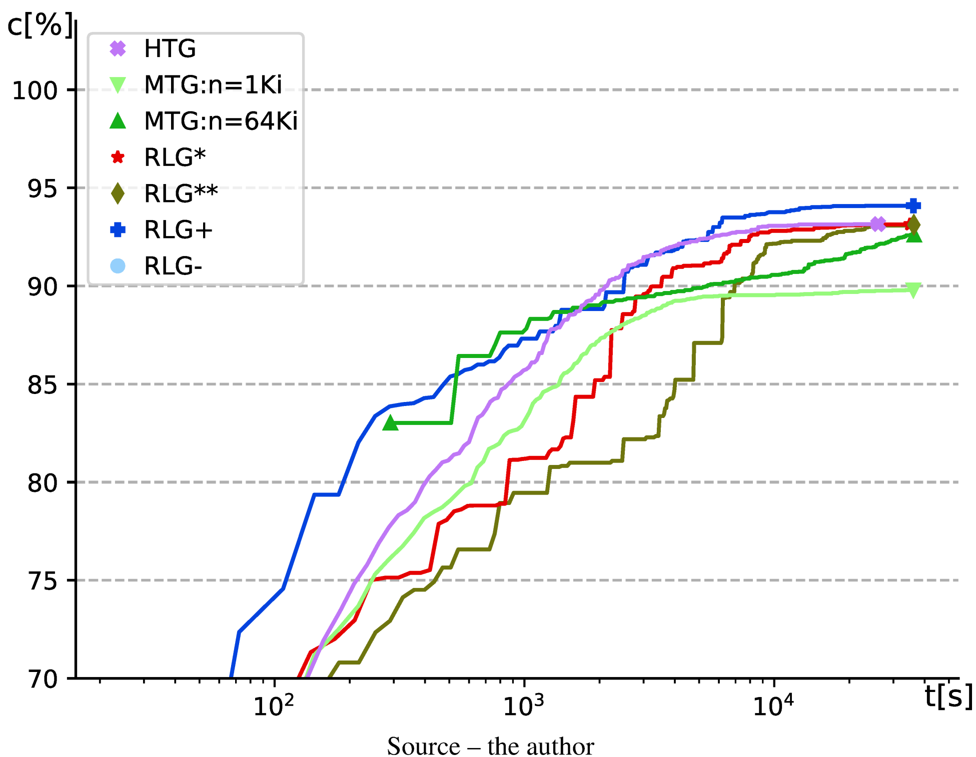
RLG variants performed similarly to other environments, but their difference was not as pronounced as when the structural coverage metric was used.

MTG behaved similarly to when the structural metric was used. Its longer-test counterpart started at a higher coverage value, it was then surpassed by its 1Ki counterpart. Finally, MTG with $n = 64Ki$ was able to achieve a slightly higher final value (90.92%) than with $n = 1Ki$ (90.80%).

The differences in behavior between MTG and RLG are less pronounced in this metric. RLG variants were usually faster to achieve coverage values during the experiment, but achieved a slightly lower coverage value in the end, except RLG+, which achieved a higher final coverage (91.45%) than both MTG variants.

Let us now compare RLG with HTG. For most of the experiment, HTG achieved coverage values faster than RLG, however, RLG+ was 20% faster to achieve HTG's final coverage (90.86%) and also achieved a higher final coverage (91.45%). Therefore, even though HTG showed good results, in the end, RLG+ was better able to use the more detailed information provided by the functional coverage metric and achieve a higher final coverage value.

Figure 8 – Coverage evolution for 2-level MOESI using the functional coverage metric



Source – the author

Finally, let us now analyze the most challenging verification environment available to us: MOESI coherence protocol and functional coverage metric. Figure 8 represents the cover-

age evolution of the selected generators for a design using 2-level MOESI and the functional coverage metric. Among the selected coherence protocols and coverage metrics, this combination is the hardest to cover, since it uses the most complex protocol with the most detailed metric. Surprisingly, the maximum coverage achieved with this combination was higher than the one obtained using the same metric with MESI 3. This happens because of the lower amount of transitions used to implement MOESI in Gem5, due to its fewer amount of cache levels. This difference is amplified by the way functional tracks the transitions from private cache levels in relation to each processor, multiplying their total amount. MESI 3 with the functional metric has about 10 times more coverage events than MOESI using the same metric.

When comparing this environment with MOESI 2 using structural coverage, two main differences can be noted: final coverage and coverage evolution. The final coverage value obtained by all generators was lower and the coverage evolution was smoother when the functional metric was used. Both of these effects can be explained by the higher amount of coverage events tracked by this metric.

For this environment, RLG- achieved a final coverage of 60.04%. RLG*, RLG** and HTG tied at around 93.13%. MTG's final coverage with $n = 64Ki$ was 92.60%, while it was 89.79% with $n = 1Ki$. RLG+ achieved the highest coverage (94.09%) and needed 3 times less time than HTG to achieve the latter's maximal coverage, as well as 6 times less time to achieve MTG's. This indicates that RLG was better able to use the more detailed information provided by the functional coverage metric and achieve higher coverage values faster in MOESI.

MTG with $n = 1Ki$ in all previous environments surpassed its longer test counterpart at some point. In this environment, however, it was never able to. This confirms our previous conclusion that more complicated environments require control over test size and longer tests to be able to be verified and even advanced techniques cannot surpass this hindrance.

By comparing the generators in all four verification environment, two coherence protocols (MESI and MOESI) and two coverage metrics (structural and functional), we conclude that RLG is a viable approach to directed test generation. RLG was able to outperform other generators in most cases, especially when paired with properly constrained RTGs, even when verifying designs with complex coherence protocols, such as MOESI.

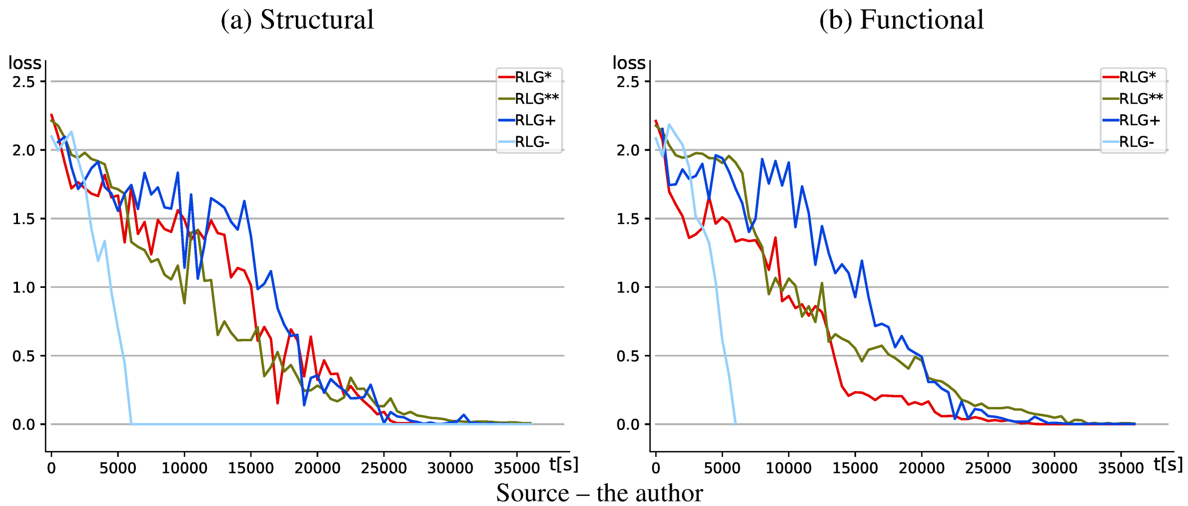
5.3 IMPACT OF PROBLEM-SPECIFIC INFORMATION ON LEARNING

RLG learning and its performance may be correlated. An RLG variant that quickly learns how to better predict the reward obtained from executing a test may perform better than a slower-learning variant. Even though RLG variants employ the same learning agent, their underlying RTGs differ. The goal of this section is to analyze how the underlying RTGs impact learning performance, more specifically, how distinct test generation constraints can influence learning.

Figures 9, and 10 show the learning evolution of all RLG variants for MESI and MOESI designs using structural and functional coverage metrics. They plot the loss as a func-

tion of time. Each loss value is the median of 500 second intervals and 10 random seeds[5]. The loss essentially represents the difference between the prediction of the agent 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 the agent is learning faster.

Figure 9 – Impact of constraints on learning for 3-level MESI and given coverage metric



Source – the author

Figure 9a shows the learning evolution for a MESI 3-level design and structural coverage. Albeit all RLG variants are clearly learning with time, 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 2-hour mark explains why it got stuck at practically the same coverage after that time (as seen in Figures 5 to 8). This happens because the two-dimensional generation space of the underlying RTG engine was exhausted. Additionally, the fewer amount of parameters the agent has do deal with simplifies learning.

In contrast, the RTG engines underlying RLG+ and RLG* have much larger three-dimensional generation spaces. That is why they can keep learning longer. Thus, RTG engines with more parameters or larger ranges of parameters have higher potential for coverage control under RL. There is, however, a limit to the advantage gained from larger generation spaces. RLG**'s four-dimensional generation space proved too big to be fully exercised in the 10-hour time frame selected for our experiments, leading to slower coverage evolution in some scenarios.

For the first 2 hours, RLG+, RLG*, and RLG** seemed to behave quite similarly, learning at a similar pace. From the 2-hour mark, up to the 6-hours mark, RLG* and RLG** were able to get lower loss values than RLG+. After that, all the three generators continued

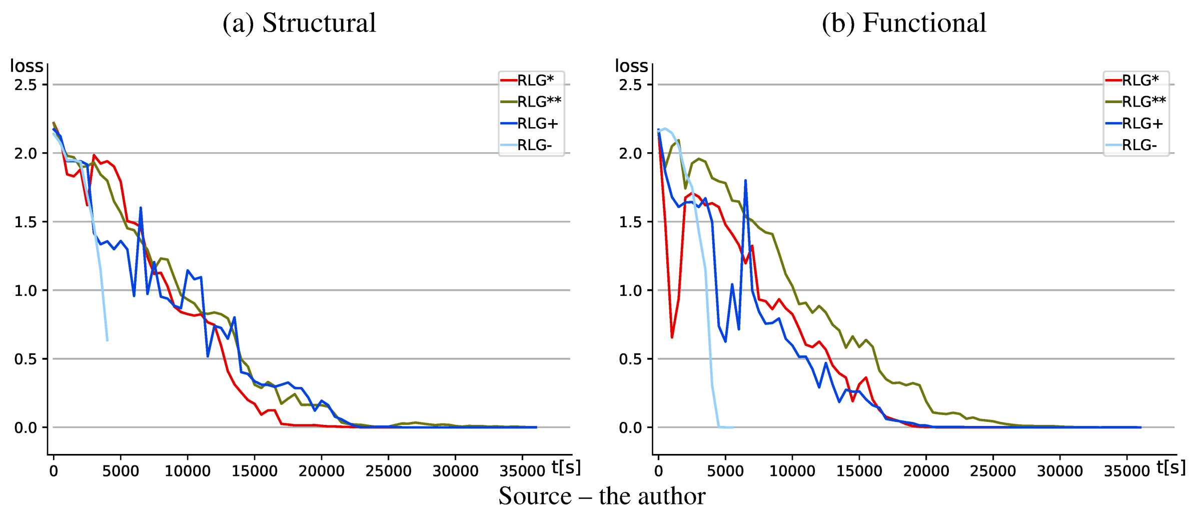
[5] The median loss was obtained as follows: First, for each random seed, the 10-hour runtime was divided into 72 intervals, each representing 500 seconds of the experiment. Then, for each interval, we calculated the median value among all loss values inside its time range. Giving us a loss function with 72 points for each random seed. Finally, we calculated the median value among all random seeds.

learning similarly again. These times match the period when RLG* and RLG** achieved higher coverage values than RLG+. This indicates that the higher constraints exploited by their test generators enabled RLG* and RLG** to learn how to increase coverage faster than RLG+ for a subtancial period of the experiment.

Figure 9b shows the learning evolution for a MESI 3-level design and functional coverage. RLG- was, again, able to learn faster, but was not able to achieve high coverage values because of its unsuitable generation space. Its loss curve also stops at around 6000 seconds into the experiment. This happens because all RLG-'s executions finished at, or before, that time, so it did not continue learning after that period.

Note that, albeit the learning behavior of RLG+ and RLG* are quite 'noisy' up to the 4-hour mark, this changes afterwards. In the intermediate phase of the verification process, specifically between 1 hour and 5 hours into the experiment, RLG* and RLG** were able to learn faster than RLG+. In this period, the generators were also able to achieve higher coverage values. In the final phase of the verification process, however, RLG+ was able to catch up, achieve low loss values and the highest final coverage. This indicates that there exists a correlation between learning and coverage evolution, however hard to identify because of the noisy aspect of the loss function.

Figure 10 – Impact of constraints on learning for 2-level MOESI and given coverage metric



Source – the author

Figure 10a shows learning evolution for a MOESI design and structural coverage metric. Even though RLG variants behaved quite differently coverage evolution-wise, their learning pattern was quite similar (except RLG-). This indicates that the difference in results of distinct variants come, in this environment, mainly from improved test quality due to constraint exploitation instead of from better learning allowed by exploiting a specific constraint. This may indicate that the structural coverage metric is not descriptive enough to inform the agents throughout the learning process when verifying a more complex coherence protocol, such as MOESI.

Figure 10b shows how a complex protocol, such as MOESI, requires a more descriptive coverage metric, such as functional, to better inform the agents in the learning process and visualize the difference between the RLG variants.

Figure 10b shows learning evolution for a MOESI design and functional coverage metric. In this verification environment, learning speed seems to be correlated with coverage evolution for RLG+, RLG*, and RLG**. RLG** learned slower throughout the whole experiment and was also the slowest variant to increase coverage. Therefore, the extra constraint exploited by RLG**'s RTG hindered learning. Even though the chaining constraint did not hamper RLG*'s learning significantly, its more frequent use of barriers slowed coverage evolution.

Experiments seem to indicate that there is a correlation between coverage evolution and learning speed. Test generation constraints can influence learning, especially when they influence generation space size. On the one hand, constraints that lead to small space size can limit the agent by not providing it enough tests to achieve high coverage values. On the other hand, generation spaces can also be too big, hindering coverage evolution by overburdening the agent with too many tests, making the task of finding good tests harder. Therefore, the selection of test generation constraints should be done carefully in order to create suitable next generation directed test generators, especially regarding methods that rely on learning.

5.4 ERROR DISCOVERY RATE AND EFFORT

Table 8 – Effort for finding errors in MESI 3-level designs with both structural and functional coverage metric

		MTG		HTG	RLG**	RLG*	RLG+	RLG-
Error	Metric	1Ki	64Ki	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}
e1	structural	98(10)	59(10)	29(10)	**12**(10)	14(10)	**11**(10)	**11**(10)
	functional	101(10)	60(10)	20(10)	**12**(10)	14(10)	**11**(10)	**12**(10)
e2	structural	959(10)	154(10)	69(10)	36(10)	31(10)	**12**(10)	**13**(10)
	functional	604(10)	162(10)	75(10)	36(10)	31(10)	**12**(10)	**12**(10)
e3	structural	994(10)	355(10)	43(10)	59(10)	29(10)	**18**(10)	1893(10)
	functional	525(10)	358(10)	61(10)	59(10)	30(10)	**18**(10)	1579(10)
e4	structural	9022(10)	34146 (5)	821(10)	2335(10)	1371(10)	1482(10)	**530**(10)
	functional	7457(10)	26822 (7)	851(10)	2039(10)	1071(10)	1031(10)	**609**(10)
e5	structural	5434 (9)	2628(10)	**665**(10)	2846(10)	**678**(10)	2858(10)	5586 (1)
	functional	7429 (8)	2741(10)	1204(10)	2198(10)	**677**(10)	2552 (9)	5606 (1)

Source – the author

Table 8 reports the effort required by each generator to expose errors in MESI 3-level 32-core designs using both coverage metrics. Each value represents the median time in seconds the generator needed to expose the error, and, between parentheses, the amount of times it succeeded (out of ten). Values in bold represent the best values achieved.

RLG**, RLG*, RLG+, and HTG were almost always able to expose all errors, while MTG and RLG- failed to expose errors in some trials. RLG required the least effort in all errors, when it was from 5 to 80 times faster than MTG and up to 6 times faster than HTG. HTG also needed the least effort to expose one error for one metric, needing as much time as RLG.

As RLG- got the worst coverage evolution of all evaluated generators, it was expected that it needed the most time to expose all errors. Indeed, it required the most time to expose e3 and e5. Errors e3 and e5 were the two errors RLG- had the most difficulty exposing. It was only able to expose e5 in 1 of the 10 trials. This can be explained by how these errors requires an L0 replacement to happen while an invalidation is being treated by the L1. Since there is a very narrow time window for this to happen, RLG-'s effectiveness was low for e5.

Surprisingly, RLG- was the best at exposing e4, an error the other generators needed from 1.4 to 64 times more time to expose. The error e4 changes L1 behavior so that it does not invalidate its L0 counterpart when the L1 is invalidating or replacing a shared block. This leaves the L0 block in state S while L1 is in state I, breaking the inclusion property. This error can only be exposed if a read to the affected block occurs after the error is set and the block has been written by another core. However, if the affected block is replaced or written, the error disappears. Since L0 is directly mapped, RLG**, RLG*, RLG+, and MTG easily cause L0 replacements. Even though RLG* and RLG+ can also disable replacements, their initial parameter k=1, maximizes replacements. Besides, L1 is two-way, a low associativity level, so RLG- is able to cause replacements in this level, even if fewer than other generators. The lower amount of L0 replacement causes RLG- to not clear e4 as it happens and, as a result, reduces the effort RLG- needs to expose it.

RLG**'s efficiency was inferior to RLG*'s and RLG+'s for most errors. This can be explained by its coverage evolution, which was also inferior to RLG*'s and RLG+'s. RLG* and RLG+ required similar effort to expose errors. The most notable exception is e5, when RLG* required from 3 to 4 times less time than RLG+. Even though RLG+ was able to achieve higher final coverage values, it was not able to detect e5 faster than RLG*. This indicates that tighter generation constraints can help detect harder errors, even when they don't seem to improve coverage evolution. In general, harder-to-find errors require higher coverage to be exposed. Even though these generators behaved similarly regarding coverage evolution, error detection depends not only on how much coverage is reached, but on reaching the transition actually exposing the error. That is, even if two generators have reached the same coverage level at a given time, they could have stimulated different sets of transitions with the same cardinality. The average effort to find an error depends on which transitions are able to expose that error, and on how many of them were covered on average.

Let us now compare MTG effort with both test sizes. MTG with $n = 64Ki$ was faster to expose errors e1, e2, e3, and e5 than with $n = 1Ki$. This is not surprising, since MTG with longer tests was usually faster to attain higher coverage values than its shorter-test counterpart. For e5, MTG with $n = 64Ki$ was not only about 2 faster to detect the error, but was also able to detect it in all trials, while MTG with $n = 1Ki$ was unable to detect it in some of the trials.

MTG with $n = 1Ki$, however, was able to detect e4 about 3.6 faster and was able to detect it in all trials.

Regarding HTG and RLG, HTG was slower than RLG for all errors and metrics except e5 and structural, where it tied with RLG*. This indicates that a better coverage evolution does not always correlates with faster error detection.

Let us now analyze how the coverage metrics affected generators effort. The effort needed by the generators changed with distinct coverage metrics. It was expected that a generator would need less time to detect an error while using a more detailed coverage metric. This effect can be seen in our effort experiments, but it was not always the case. HTG was the generator that better exploited the more detailed data provided by the functional metric. It was up to 2 times faster when using this metric. MTG and RLG were also able to exploit this effect, to a lesser extent, achieving up to 89% and 43% improvement over their structural metric effort, respectively. MTG needed 36% more time to detect e5 with the functional metric. This indicates that advanced DTG techniques may exploit more detailed coverage metrics to not only improve covearge evolution, but also to detect errors faster.

RLG was able to expose all errors in 21 minutes while MTG took as much as 3 hours and HTG, 27 minutes. This indicates that constrained random generation combined with reinforcement learning can be made more efficient in error discovery than genetic programming and at least as efficient as hybrid generation.

Table 9 – Effort for finding errors in MOESI 2-level designs with both structural and functional coverage metric

		MTG		HTG	RLG**	RLG*	RLG+	RLG-
Error	Metric	1Ki	64Ki	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}
E1	structural	**8**(10)	47(10)	**7**(10)	17(10)	17(10)	17(10)	886(10)
	functional	**8**(10)	46(10)	**7**(10)	17(10)	18(10)	17(10)	1320(10)
E2	structural	**9**(10)	47(10)	**7**(10)	18(10)	19(10)	20(10)	912(10)
	functional	**8**(10)	46(10)	**7**(10)	18(10)	19(10)	22(10)	1359(10)
E3	structural	21(10)	49(10)	35(10)	13(10)	11(10)	10(10)	**8**(10)
	functional	42(10)	48(10)	25(10)	13(10)	11(10)	10(10)	**7**(10)
E4	structural	36000(0)	36000(0)	641(10)	4509(10)	590(10)	**147**(10)	4407 (0)
	functional	3962 (8)	3170(10)	1073(10)	6127(10)	349(10)	**173**(10)	4547 (0)
E5	structural	36000 (1)	4680(10)	586(10)	4262(10)	1359(10)	**291**(10)	4432 (0)
	functional	36000 (1)	1478(10)	1044(10)	3739(10)	2348 (9)	**365**(10)	4562 (0)
E6	structural	481(10)	5267(10)	25486(0)	**105**(10)	34434(0)	35446(0)	4332 (1)
	functional	758(10)	5304(10)	25634(0)	**107**(10)	36000(0)	36000(0)	4430 (1)

Source – the author

Table 9 reports the median effort to detect errors for all the selected generators in MOESI 2-level 32-core designs using both coverage metrics. An entry filled in black indicates the error was never found.

RLG** was the only generator able to expose all the errors in every trial, while all the other generators failed to detect an error in one or more trials. MTG was the fastest at exposing

E1 and E2, tied with HTG. RLG-, at exposing E3, RLG+, at exposing E4 and E5, and RLG** at exposing E6. RLG was best at exposing errors E4, E5, and E6, the harder to expose errors among the selected ones.

Let us now analyze how MTG fared. MTG was the best at exposing errors E1 and E2. These errors, however, were detected on the first tests executed by all generators, except by RLG-. This also explains why MTG with tests of size 1Ki instructions was almost 6 times faster than its longer test counterpart. However, MTG with the longer tests was able to always detect E5, while its shorter-test counterpart could only detect it in 10% of the trials.

Most generators were not able to improve detection time when using the functional coverage metric, as opposed to when using the structural one. RLG** and RLG* were able to detect E5 faster (14% and 69% faster, respectively). HTG and RLG- were only able to improve the detection time of E3, an easy error. MTG was not able to detect E4 using the structural coverage, an error the other generators (except RLG-) needed from 2 to 75 minutes to detect. MTG, however, was able to detect E4 in most tries when using the functional metric. This indicates that the more detailed information provided by the functional coverage metric enabled MTG to detect E4, while the structural metric did not provide enough information to enable the detection of the error. Therefore, this indicates that some errors, especially the ones that involve processor interaction, require the use of more detailed coverage metrics to be detected by some techniques.

Let us now analyze HTG. The advantages observed for HTG in coverage evolution did not translate completely to effort for MOESI 2-level designs. HTG was the best at detecting E1 and E2, the easiest errors, while taking longer to detect all other errors. HTG needed from 2 to 6 times more time do detect E4 and E5, two of the harder to detect errors, and it was not able to detect E6.

Now, let us look at the effort needed by the RLG variants. RLG was the best at exposing E3, E4, E5 and E6. RLG- needed the most time to detect all errors, except E3. RLG- could not, however, detect E4 and E5. RLG*, exploiting its tighter generation constraints, could detect E4 and E5, even though it was not the best at exposing them. RLG+ needed the least time, among all generators, to expose E4 and E5, two of the harder to detect errors.

Regarding E6, the error that requires false sharing to be detected, only 4 generators were able to detect it: RLG**, MTG (with $n = 1Ki$ and $n = 64Ki$), and RLG-. RLG** and MTG were able to reliably detect it in all tries, while RLG- only detected it in one try. RLG- was able to detect E6 because of how its unconstrained test generator can create a situation where false sharing is possible with a low probability of occurrence. MTG needed from 4 to 50 times more time to detect E6 than RLG**, but it was still able to detect this error. This was possible because, even though the initial tests generated by MTG allocate one shared variable per cache block, new tests created by the genetic algorithm can create variables in new addresses, enabling false sharing. This also explains why MTG with $n = 1Ki$ needed from 7 to 11 times less time than its longer-test counterpart to detect this error. Since it has a higher test throughput, it is quicker to create a mutation that enables false sharing. RLG** could easily detect E6 because

of its control over sharing. This indicates that, even though the addition of a sharing constraint hindered coverage evolution for RLG** and increased the effort needed to expose other errors, a control over sharing is needed to detect some challenging errors.

RLG+ needed 8 minutes to expose all errors (except E6) under MOESI and the functional metric, while HTG needed 21 and MTG ($n = 64Ki$), 80 minutes. Therefore, we conclude that the proposed generator was superior in most cases, especially in our most challenging verification environment. This indicates that a reinforcement learning based approach, allied with proper constraints, is likely to cope with the verification needs of new, highly complex, multicore processors, especially when paired with more detailed coverage metrics.

5.5 STATISTICAL SIGNIFICANCE OF FINAL COVERAGE VALUES

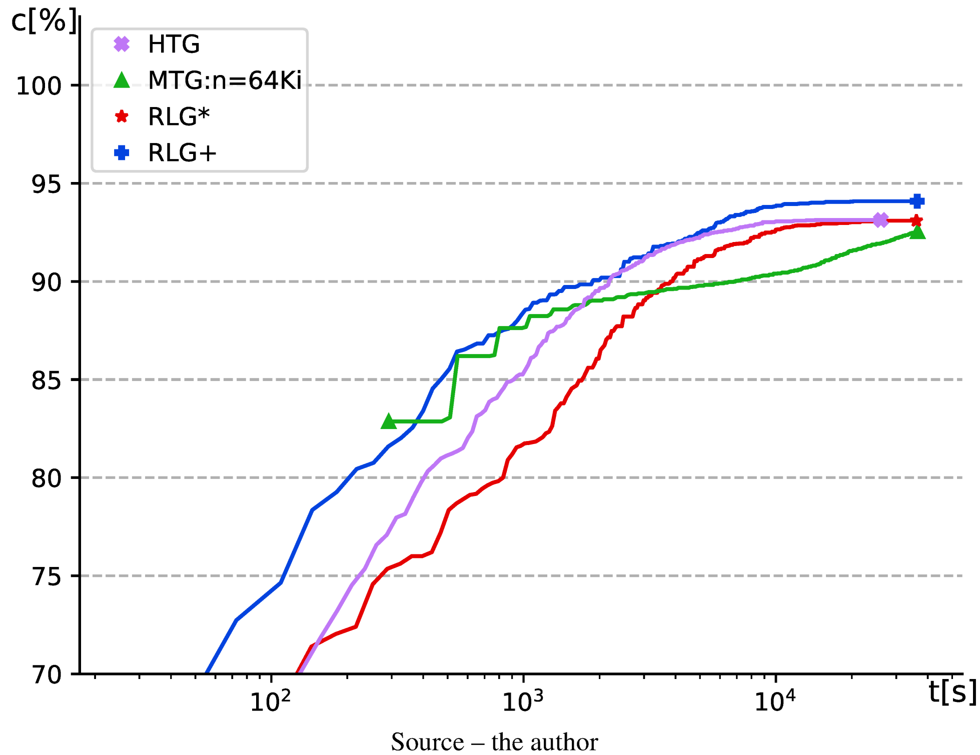
In this section, we executed extra experiments and employed statistical significance testing in order to have better confidence in the results presented in section 5.2, especially regarding final coverages. Since time restrictions limited the amount of seeds used in prior experiments to 10, it was not feasible to use statistical methods to analyze the data. Such methods require a higher amount of samples to express meaningful results. In order to make this experiment feasible but still obtain useful results, we limited the generators to those which achieved the highest coverages observed in the prior experiments, (RLG*, RLG+, HTG, and MTG with $n = 64Ki$), and tested them in the most challenging environment available to us: MOESI coherence protocol and functional coverage metric. The experiments presented in this section were executed 50 times, each with a distinct random seed.

Figure 11 shows the coverage evolution for the selected generators using extra random seeds. Each data point represents the median value among 50 random seeds. It is remarkable how this figure resembles Figure 8, albeit smoother from the higher amount of data samples used to generate it. The final coverage obtained by the generators was very similar to when using fewer seeds, only differing by at most 0.06 percentage points. This indicates that the amount of seeds used in the previous experiments was enough to accurately represent the behavior of the generators, even though insufficient to be strictly statistically significant.

RLG+ achieved the highest final coverage of 94.09%, while HTG, RLG* and MTG achieve 93.13%, 93.10%, and 92.54%, respectively. RLG+ was more than 2 times faster than HTG to achieve its final coverage and 7 times faster than MTG to achieve its final coverage.

In order to make sure the final coverage obtained by the generators was indeed statistically significantly distinct we used statistical hypothesis testing. We selected significance level $\alpha = 0.05$. First, we tested our data, the final coverage obtained for each generator and seed, for normality using 3 normality tests, since each is better at categorizing different types of distributions (YAZICI; YOLACAN, 2007): the Shapiro-Wilk test, D'Agostino's K^2 test, and Anderson-Darling test. All 3 tests were unanimous in their responses, under the same α. RLG+ and RLG* final coverage values do not seem to be normally distributed, while HTG and MTG do. So, since not all our data is normally distributed, we had to opt for a non-parametric hy-

Figure 11 – Coverage evolution for 2-level MOESI using the functional coverage metric with extra random seeds



Source – the author

pothesis test. We chose the Mann-Whitney U test, recommended by McCrum-Gardner (2007). All generators differed significantly from one another ($p < 0.05$ one-tailed) except for two of them: HTG and RLG*. This was expected since their final coverage was very similar (HTG: 93.13%, RLG*: 93.1%).

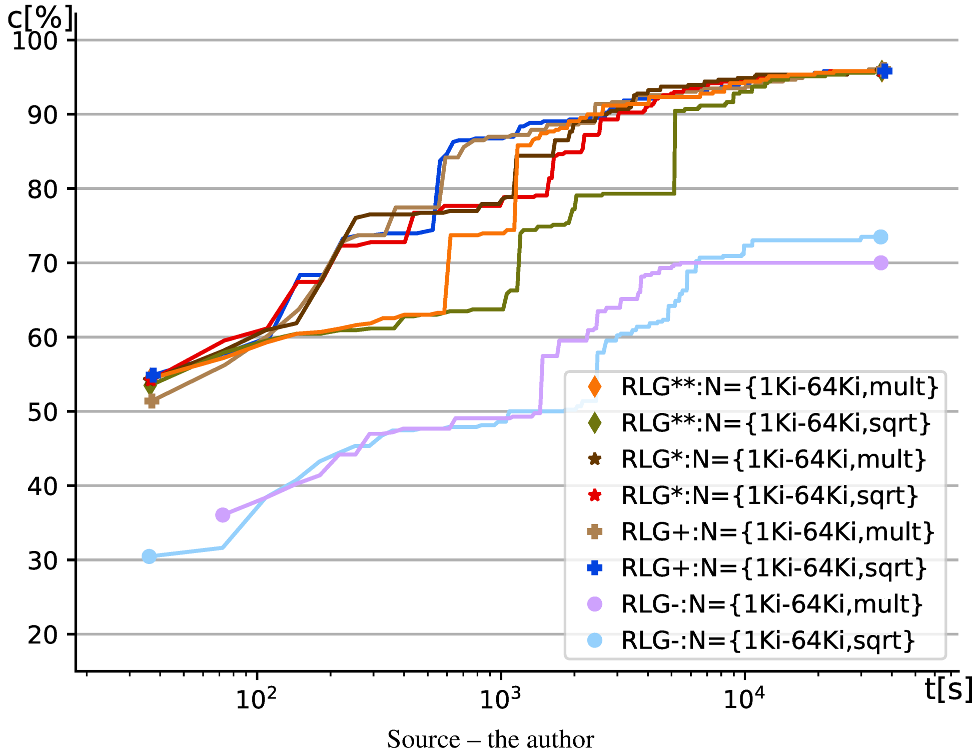
These results increase the confidence in our previous conclusion that RLG was able to achieve a higher final coverage than two state-of-the-art generators under the most complex environment available to us: MOESI protocol with functional coverage metric.

5.6 IMPACT OF TEST SIZE STEP ON COVERAGE EVOLUTION

This section evaluates how test length step affects coverage evolution. Figure 12 shows coverage evolution for 3-level MESI using two distinct functions for inducing multiple test lengths: the original function $f(i) = 2^i$ described in section 4 and the additional function $f'(i) = \lceil 2^{i/2} \rceil$. The second function induces a shorter step than the first when increasing or decreasing test length. They allow us to evaluate the impact on coverage of different generation spaces with the same bounds, but distinct granularities for the parameter n, which has the largest impact on test throughput. To distinguish the induced generation spaces, we denote them by different acronyms for the above functions: *mult* and *sqrt*, respectively[6].

[6] Note that, for the adopted ranges, $|N|$ is 13 and 7, for *sqrt* and *mult*, respectively

Figure 12 – Impact of test size granularity on coverage evolution for 3-level MESI



Source – the author

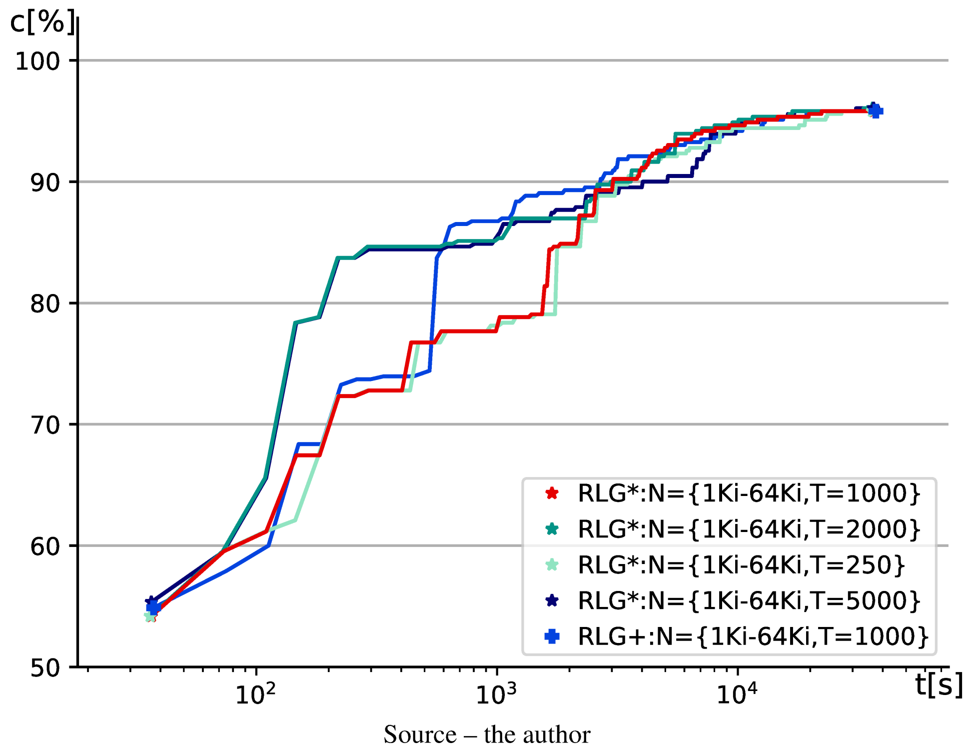
Note that both RLG* and RLG+ generally performed better than (or at least similar to) their additional function counterparts using the *mult* function. This can be explained as follows. At each action, the agent can only apply the induction function once to the generation parameters. Therefore, a function that returns a larger increase per action leads to larger tests being executed, leading to faster coverage evolution in this experiment. Also note that RLG- reached a lower final coverage using the *mult* function than the original *sqrt* function counterpart. This happens because a smaller $|N|$ leads to a smaller generation space, which resulted in fewer tests being executed. Since RLG- does not exploit functional hardware properties or functional properties of parallel programs, it relies heavily in test quantity to reach higher coverage values.

This experiment indicates that, although *sqrt* provides a tighter control over test throughput, its smaller step harms coverage evolution. Therefore, *mult* seems to be a more suitable induction function in our experimental conditions.

5.7 IMPACT OF TIME QUANTIZATION ON COVERAGE EVOLUTION

In order to evaluate the impact of environment representation on RLG's coverage evolution, we executed an experiment using RLG's most efficient variant (RLG*) with different values of time quantization levels T. Since our original RLG evaluation was performed with T=1000, we selected three extra values: one smaller than 1000, and two larger. We performed experiments using four values for T: 250, 1000, 2000, and 5000. As a result, we evaluated

Figure 13 – Impact of time quantization on coverage evolution of 3-level MESI



Source – the author

three extra representations, one where time was coarser-grained, and two where time was finer-grained. When the environment is represented using a smaller amount of time quantization levels, the agent has a coarser-grain view of time. That is, it sees time less precisely. When the environment is represented using a larger amount of time quantization levels, the agent has a finer-grain view of time. That is, it sees time more precisely.

Figure 13 shows the coverage evolution for MESI 3-level of the two most efficient variants of RLG with the default value for time quantization levels (T=1000) and RLG* with T=250, T=2000, and T=5000 amount of levels.

Notice that, RLG* with T=250 takes slightly more time to reach each coverage value, when compared to the other generators, this indicates that coarser-grain time quantization harmed the generator's efficiency. Note that RLG* with both T=2000 and T=5000 reached the highest final coverage (96.05%) observed in this experiment and were 30% faster than their T=1000 counterpart to reach its highest coverage (95.81%). However, RLG* was almost 10% faster to achieve the higher final coverage when using the higher value of T. This indicates that finer-grain time quantization improves generator efficiency, although we get diminishing returns as T approaches higher values.

6 CONCLUSIONS AND PERSPECTIVES

Slow multicore simulation and short verification budgets require the use of Directed Test Generation for efficient coverage control during pre-silicon verification. This dissertation proposes casting DTG as coverage-driven RTG. Based on estimated coverage and elapsed time, our directing engine selects a few RTG parameters in order to maximize coverage under a time constraint.

Our generator harnesses the power of reinforcement learning to learn how to select RTG's parameters in order to maximize coverage evolution. The next test parameters are selected through the use of customizable actions, making the agent adaptable to distinct RTGs. Since the agent only uses the percentage of total coverage through a coverage metric, it is also reusable across distinct coherence protocols. We proposed four variants of our agent, each with tighter constraints based on test generation legacy constraints.

Experiments were performed to evaluate how our generator fared against two state-of-the-art generators under three aspects: coverage evolution, error detection, and effort. The generators were analyzed under two coherence protocols, MESI and MOESI, and under two complementary coverage metrics, structural and functional. Under the most challenging environment available to us (MOESI and functional), our generator was capable of achieving a final coverage of 94.09%, while HTG was able to achieve 93.13% and MTG, 92.54%. RLG was also more than 2 times faster to achieve HTG's max coverage and 7 times faster to achieve MTG's. The final coverage exhibited by the proposed generator was also shown to be statistically significantly (for $\alpha = 0.05$) greater than the compared generators. Regarding error detection, the proposed generator needed 8 minutes to expose all errors (except one) under the most challenging environment, while the other generators needed from 21 to 80 minutes.

We also analyzed the agent prediction error (loss) of the four variants of the proposed generator in order to establish how test constraints impact learning. Experiments seem to indicate that there is a correlation between coverage evolution and learning speed, however, this may be difficult to identify because of the noisy aspect of the loss function. They also seem to indicate that test generation constraints can influence learning, especially when they influence generation space size. Generation space can be too small, leading to a low final coverage, as well as too big, leading to poorer coverage evolution because of how it makes the task of finding good tests harder. Therefore, the selection of test generation constraints should be done carefully in order to create suitable next generation directed test generators, especially regarding methods that rely on learning.

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 tools for verification.

We believe that the proposed technique could be easily adapted for usage in industrial verification environments. The tailoring of the Directing Engine to a new framework seems straightforward. However, such tailoring would be insufficient to obtain the same impact observed in the results reported in this dissertation. The effectiveness of reinforcement learning actions largely depends on properly constraining RTG. To produce the results reported in this thesis, we have relied on two techniques that exploit non-conventional constraints, which are called *chaining* and *biasing* (ANDRADE et al., 2018). That is why we advise their use (or of similar techniques) in combination with the Directing Engine proposed in this dissertation.

6.1 FUTURE WORK

Limited time prevented us from further investigating a few research aspects. To better assess extra aspects of this research, the experimental evaluation could be extended as follows:

- **Impact of composed actions**: All the actions proposed in this work were limited to modifying a single parameter at a time. Composed actions (those simultaneously modifying multiple parameters), could be exploited by the agent in order to increase the number of distinct test in sequence, which would probably lead to better coverage evolution, since they are likely to expose different coverage events.

- **Impact of online training**: This work employed online training, that is, the RNN training was done between test executions. There is a trade-off between training time and coverage evolution: more training can lead to better decision making, but it takes time from test execution. Extra experiments could pave the way towards a technique able to dynamically find the best trade-off.

Finally, we envisage a couple of new artifacts:

- **A new constraint-oriented directing engine based on genetic programming**: The McVerSi technique (ELVER; NAGARAJAN, 2016), which we encapsulated in our framework as MTG, has shown that genetic programming can lead to an adequate DTG technique. However, it is unable to exploit generation constraints explicitly. Since chaining and biasing (ANDRADE et al., 2018) were beneficial to RLG, they are likely to be beneficial to a new generator where the Directing Engine would use genetic programming to *select* parameters for an RTG engine based on chaining and biasing.

- **A generalized agent**: The proposed agent could be generalized so as to have a more fine-grained control over the generation process as perhaps rely on alternative network types and architectures that could be better targeted to the application domain of test generation.

6.2 PUBLICATIONS

The work described in this dissertation was partially reported in the proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE). The author also contributed to the publication of an article in the journal IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) and of a paper in the proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD).

- ICCAD 2018 paper (ANDRADE et al., 2018)
- DATE 2020 paper (PFEIFER et al., 2020)
- TCAD 2020 article (ANDRADE et al., 2020)

BIBLIOGRAPHY

ADIR, A. et al. Genesys-pro: innovations in test program generation for functional processor verification. **IEEE Design Test of Computers**, v. 21, n. 2, p. 84–93, Mar 2004. ISSN 0740-7475.

ADIR, A. et al. Verification of transactional memory in power8. In: **2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2014. p. 1–6. ISSN 0738-100X.

ADVE, S. V.; GHARACHORLOO, K. Shared memory consistency models: a tutorial. **Computer**, IEEE, v. 29, n. 12, p. 66–76, Dec 1996. ISSN 0018-9162.

ALGLAVE, J. et al. Fences in weak memory models. In: TOUILI, T.; COOK, B.; JACKSON, P. (Ed.). **Computer Aided Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 258–272. ISBN 978-3-642-14295-6.

ANDRADE, G. A. G. et al. Steep Coverage-ascent Directed Test Generation for Shared-memory Verification of Multicore Chips. In: **Proceedings of the International Conference on Computer-Aided Design**. [S.l.]: ACM, 2018. p. 29:1–29:8. ISBN 978-1-4503-5950-4.

ANDRADE, G. A. G. et al. A Directed Test Generator for Shared-Memory Verification of Multicore Chip Designs. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, 2020.

ANDRADE, G. A. G.; GRAF, M.; SANTOS, L. C. V. dos. Chain-Based Pseudorandom Tests for Pre-Silicon Verification of CMP Memory Systems. In: **34th IEEE International Conference on Computer Design (ICCD)**. [S.l.: s.n.], 2016. p. 552–559.

BINKERT, N. et al. The Gem5 Simulator. **SIGARCH Computer Architecture News**, Association for Computing Machinery, New York, NY, USA, v. 39, n. 2, p. 1–7, aug. 2011. ISSN 0163-5964.

BOLOSKY, W. J.; SCOTT, M. L. False sharing and its effect on shared memory performance. In: **USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4**. Berkeley, CA, USA: USENIX Association, 1993. (Sedms'93), p. 3–3. Disponível em: http://dl.acm.org/citation.cfm?id=1295480.1295483.

DENNARD, R. H. et al. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. **IEEE Journal of Solid-State Circuits**, v. 9, n. 5, p. 256–268, oct. 1974. ISSN 1558-173X.

DEVADAS, S. Toward a coherent multicore memory model. **Computer**, IEEE, v. 46, n. 10, p. 30–31, 2013.

DINECHIN, B. D. de et al. A clustered manycore processor architecture for embedded and accelerated applications. In: **2013 IEEE High Performance Extreme Computing Conference (HPEC)**. [S.l.: s.n.], 2013. p. 1–6.

DUELL, S.; UDLUFT, S.; STERZING, V. Solving partially observable reinforcement learning problems with recurrent neural networks. In: **Neural Networks: Tricks of the Trade**. Springer, Berlin, Heidelberg, 2012. p. 709–733. Disponível em: https://doi.org/10.1007/978-3-642-35289-8_38.

ELVER, M. **McVerSi Framework**. [S.l.]: GitHub, 2016. https://github.com/melver/mc2lib.

ELVER, M.; NAGARAJAN, V. McVerSi: A test generation framework for fast memory consistency verification in simulation. In: **IEEE Int. Symp. on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2016. p. 618–630.

ESMAEILZADEH, H. et al. Dark silicon and the end of multicore scaling. In: . [S.l.: s.n.], 2011. v. 32, p. 365–376.

FINE, S.; FOURNIER, L.; ZIV, A. Using bayesian networks and virtual coverage to hit hard-to-reach events. **International Journal on Software Tools for Technology Transfer**, v. 11, n. 4, p. 291–305, 10 2009. ISSN 1433-2787.

FINE, S.; ZIV, A. Coverage directed test generation for functional verification using bayesian networks. In: **Proceedings of the 40th Annual Design Automation Conference**. New York, NY, USA: ACM, 2003. (DAC '03), p. 286–291. ISBN 1-58113-688-9.

FLOOD, M. M. The traveling-salesman problem. **Operations Research**, v. 4, n. 1, p. 61–75, 1956.

FRANCESQUINI, E. et al. On the energy efficiency and performance of irregular application executions on multicore, numa and manycore platforms. **Journal of Parallel and Distributed Computing**, v. 76, p. 32–48, 2015. ISSN 0743-7315. Special Issue on Architecture and Algorithms for Irregular Applications. Disponível em: https://www.sciencedirect.com/science/article/pii/S0743731514002093.

FREITAS, L. S.; RAMBO, E. A.; SANTOS, L. C. V. dos. On-the-fly verification of memory consistency with concurrent relaxed scoreboards. In: **Design, Automation, and Test in Europe (DATE)**. [S.l.: s.n.], 2013. p. 631–636. ISBN 978-1-4503-2153-2.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. http://www.deeplearningbook.org.

GRAVES, A. **Supervised Sequence Labelling with Recurrent Neural Networks**. [S.l.]: Springer, 2012. v. 385. 1-131 p. (Studies in Computational Intelligence, v. 385). ISBN 978-3-642-24796-5.

GROCE, A. Coverage rewarded: Test input generation via adaptation-based programming. In: **2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)**. [S.l.: s.n.], 2011. p. 380–383. ISSN 1938-4300.

HANGAL, S. et al. TSOtool: A program for verifying memory systems using the memory consistency model. **ACM SIGARCH Comp. Arch. News**, ACM, New York, NY, USA, v. 32, n. 2, p. 114–123, Mar 2004. ISSN 0163-5964.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 6th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055.

HESSEL, M. et al. Rainbow: Combining improvements in deep reinforcement learning. In: **AAAI Conference on Artificial Intelligence**. [s.n.], 2018. Disponível em: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17204/16680.

HU, W. et al. Linear Time Memory Consistency Verification. **IEEE Transactions on Computers**, v. 61, n. 4, p. 502–516, Apr 2012. ISSN 0018-9340.

JAEGER, H. The" echo state" approach to analysing and training recurrent neural networks-with an erratum note’. **Bonn, Germany: German National Research Center for Information Technology GMD Technical Report**, v. 148, 01 2001.

KIM, J.; KWON, M.; YOO, S. Generating test input with deep reinforcement learning. In: **Proceedings of the 11th International Workshop on Search-Based Software Testing**. New York, NY, USA: ACM, 2018. (SBST ’18), p. 51–58. ISBN 978-1-4503-5741-8. Disponível em: http://doi.acm.org/10.1145/3194718.3194720.

KOZA, J. R. **Genetic programming: on the programming of computers by means of natural selection**. 1. ed. [S.l.]: MIT Press, 1992. (Complex adaptive systems). ISBN 0262111705,9780262111706.

LAMPORT, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. **IEEE Transactions on Computers**, IEEE Computer Society, Washington, DC, USA, v. 28, n. 9, p. 690–691, sep. 1979. ISSN 0018-9340.

LANG, K. J.; WAIBEL, A. H.; HINTON, G. E. A time-delay neural network architecture for isolated word recognition. **Neural Networks**, v. 3, n. 1, p. 23–43, 1990. ISSN 0893-6080.

LITTMAN, M.; SUTTON, R. S. Predictive representations of state. In: **Advances in Neural Information Processing Systems**. [S.l.]: MIT Press, 2002. v. 14.

LUSTIG, D.; PELLAUER, M.; MARTONOSI, M. Pipe check: Specifying and verifying microarchitectural enforcement of memory consistency models. In: **Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2014. (MICRO-47), p. 635–646. ISBN 978-1-4799-6998-2.

MANOVIT, C.; HANGAL, S. Completely verifying memory consistency of test program executions. In: **IEEE Int. Symposium on High-Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2006. p. 166–175.

MARTIN, M. M.; HILL, M. D.; SORIN, D. J. Why on-chip cache coherence is here to stay. **Communications of the ACM**, ACM, v. 55, n. 7, p. 78–89, June 2012.

MCCRUM-GARDNER, E. Which is the correct statistical test to use? **British Journal of Oral and Maxillofacial Surgery**, v. 46, n. 1, p. 38 – 41, 2008. ISSN 0266-4356. Disponível em: http://www.sciencedirect.com/science/article/pii/S0266435607004378.

MOORE, G. E. Cramming More Components onto Integrated Circuits. **Electronics**, v. 38, n. 8, p. 114–117, apr. 1965.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design, Fifth Edition: The Hardware/Software Interface**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 0124077269.

PFEIFER, N. et al. A reinforcement learning approach to directed test generation for shared memory verification. In: **2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2020. p. 538–543.

QIN, X.; MISHRA, P. Automated generation of directed tests for transition coverage in cache coherence protocols. In: **Design, Automation, and Test in Europe (DATE)**. [S.l.: s.n.], 2012. p. 3–8. ISSN 1530-1591.

SHAKERI, N. et al. Near optimal machine learning based random test generation. In: **2010 East-West Design Test Symposium (EWDTS)**. [S.l.: s.n.], 2010. p. 420–424.

SORIN, D. J.; HILL, M. D.; WOOD, D. A. **A Primer on Memory Consistency and Cache Coherence**. 1st. ed. [S.l.]: Morgan & Claypool Publishers, 2011. ISBN 1608455645.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. 2. ed. [S.l.]: MIT Press, 2018. ISBN 9780262039246.

WAGNER, I.; BERTACCO, V. MCjammer: Adaptive Verification for Multi-core Designs. In: **Design, Automation, and Test in Europe (DATE)**. [S.l.: s.n.], 2008. p. 670–675. ISSN 1530-1591.

WAGNER, I.; BERTACCO, V. **Post-Silicon and Runtime Verification for Modern Processors**. [S.l.]: Springer US, 2010. (SpringerLink : Bücher). ISBN 9781441980342.

YAZICI, B.; YOLACAN, S. A comparison of various tests of normality. **Journal of Statistical Computation and Simulation**, Taylor |& Francis, v. 77, n. 2, p. 175–183, 2007.

ZHANG, M. et al. PVCoherence: Designing Flat Coherence Protocols for Scalable Verification. **IEEE Micro**, v. 35, n. 3, p. 84–91, May 2015. ISSN 0272-1732.

ÅSTRÖM, K. J. Optimal control of markov processes with incomplete state information I. Elsevier, v. 10, p. 174–205, 1965. ISSN 0022-247X.