UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Gabriel Arthur Gerber Andrade

Test generation for shared-memory verification of multicore chips

Florianópolis
2021

Gabriel Arthur Gerber Andrade

Test generation for shared-memory verification of multicore chips

Tese submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas para a obtenção do título de doutor em Engenharia de Automação e Sistemas.
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.

Andrade, Gabriel Arthur Gerber
Test generation for shared-memory verification of multicore chips / Gabriel Arthur Gerber Andrade ; orientador, Luiz Cláudio Villar dos Santos , 2021.
112 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia de Automação e Sistemas, Florianópolis, 2021.

Inclui referências.

1. Engenharia de Automação e Sistemas. 2. Multicore. 3. Memória compartilhada coerente. 4. Verificação. 5. Geração de testes. I. , Luiz Cláudio Villar dos Santos. II. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Engenharia de Automação e Sistemas. III. Título.

Gabriel Arthur Gerber Andrade

Test generation for shared-memory verification of multicore chips

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

Prof. Rodolfo Jardim Azevedo, Dr.
Universidade Estadual de Campinas

Prof. Rômulo Silva de Oliveira, Dr.
Universidade Federal de Santa Catarina

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

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de doutor em Engenharia de Automação e Sistemas.

Prof. Werner Kraus Junior, Dr
Coordenador do Programa de Pós-Graduação

Prof. Luiz Cláudio Villar dos Santos, Dr.
Orientador

Florianópolis, 2021.

Eu dedico este trabalho ao passado.

ACKNOWLEDGEMENTS

I would like to thank all colleagues, professors, and fellow researchers that contributed, direct or indirectly, to the development of this work, in special my advisor, Luiz C. V. dos Santos, and coworkers, Marleson Graf and Nícolas Pfeifer, and seniors, Eberle Rambo and Leandro Freitas and Olav Henschel. Also, I would like to thank my family and friends for all support and encouragement provided over the years.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and by the Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq) - Finance Code GD.

RESUMO

Multiprocessadores consistem de processadores fortemente acoplados que compartilham um espaço de endereçamento de memória e são construídos com um ou múltiplos *multicore chips*. O susbsistema de memória compartilhada envolve componentes de hardware complexos que implementam um sofisticado protocolo de coerência cujo projeto em RTL é propenso a erros. Esta tese contribui para a verificação funcional do comportamento da memória compartilhada durante o projeto de *multicore chips*. Uma vez que o comportamento não determinístico é chave para expor erros de memória compartilhada, programas paralelos não sincronizados são frequentemente utilizados para a verificação do projeto e para o teste do protótipo. No entanto, durante a etapa de verificação, a lenta execução em um simulador requer o uso de técnicas não convencionais para expor erros e prover alta cobertura com programas mais curtos. Neste contexto, esta tese faz três contribuições. A primeira contribuição consiste em duas técnicas que impõem restrições à geração aleatória convencional de programas de teste para viabilizar a verificação eficiente da memória compartilhada. Uma delas explora cadeias canônicas de dependências para restringir a geração aleatória de sequências de instruções (de forma a aumentar a cobertura de transições entre estados que são induzidas por eventos de conflito em um mesmo endereço), outra explora restrições sobre o espaço de endereçamento para restringir o assinalamento aleatório de endereços distintos (de forma a aumentar a cobertura de transições entre estados devidas a eventos de evicção). Quando comparada a um gerador convencional, a combinação destas técnicas reduziu o esforço médio de verificação em uma ordem de magnitude em vários casos. A segunda contribuição é um novo mecanismo de geração dirigida por modelo de cobertura para melhorar a qualidade de testes não determinísticos. O mecanismo explora propriedades gerais de protocolos de coerência e de memórias *cache* para melhor controlar a cobertura de transições entre estados (a qual serve como *proxy* para aumentar a cobertura de acordo com a métrica que vier a ser adotada em um dado ambiente de verificação). Por ser independente de métrica de cobertura, de protocolo de coerência, e de parâmetros de *cache*, o mecanismo é reusável em projetos e ambientes de verificação bem diferentes. Este gerador baseado em modelo foi mais rápido para atingir valores de cobertura similares aos de um gerador dirigido por dados baseado em Programação Genética, reportado em trabalho correlato. Por exemplo, ao executar testes com 1K operações para verificar projetos de 32 núcleos, este nosso gerador alcançou 60% da cobertura dez vezes mais rápido. A terceira contribuição é uma abordagem híbrida, a qual não é uma simples combinação de uma técnica dirigida por dados com outra dirigida por modelo. A abordagem reformula a geração dirigida de testes como um problema de otimização com dois objetivos e explora vizinhanças para evitar a enumeração explícita do espaço de estados do protocolo de coerência, sem excluir soluções ótimas do espaço de pesquisa. Comparada com um gerador puramente dirigido por dados e com outro baseado em modelo, a abordagem híbrida levou a uma melhor evolução da cobertura ao longo do tempo, quando avaliada para 32 núcleos e para diferentes protocolos de coerência. Por exemplo, para um protocolo MOESI de dois níveis, a abordagem foi 2,7 vezes mais rápida do que o gerador baseado em modelo e cerca de 5 a 19 vezes mais rápido do que o gerador dirigido por dados. Finalmente, a abordagem híbrida também foi comparada com um gerador dirigido por dados baseado em Aprendizado por Reforço. Os resultados experimentais mostraram que a abordagem híbrida proposta é de 2 a 3 vezes mais rápida para obter a máxima cobertura atingida por aquele gerador.

Palavras-chave: *Multicores*. Memória compartilhada coerente. Verificação. Geração de testes.

RESUMO EXPANDIDO

Introdução

Multiprocessadores são computadores cujos processadores foram acoplados de forma a compartilhar um mesmo espaço de endereçamento de memória sob o controle de um único sistema operacional. Dado que multiprocessadores são construídos com um ou múltiplos *multicore chips*, esta tese investiga a *verificação de projeto* do seu componente básico: o *multicore chip*. A comunicação via memória compartilhada costuma aderir a um protocolo de coerência de cache, frequentemente implementado em *hardware*. Há indicadores de que a coerência tende a continuar importante para *multicore chips* voltados a aplicações de propósitos gerais. Embora o número de estados do protocolo de coerência cresça exponencialmente com o número de núcleos, há indícios de que (através de decisões de projeto adequadas) é possível preservar a escalabilidade do *hardware* de suporte à coerência. Portanto, a viabilidade prática da coerência em *hardware* – ao menos no escopo de um *multicore chip* – acaba resultando em um desafio à verificação de seu projeto, mesmo sob a limitação que a potência térmica impõe ao crescimento do número de núcleos. Além disso, uma vez que a maioria dos programas paralelos são elaborados com base em bibliotecas de sincronização, muitos fabricantes passaram a propor processadores que relaxam a consistência sequencial da memória compartilhada (permitindo o relaxamento da ordem entre leituras e escritas para endereços distintos), sem que o programador precise se preocupar com isso. Ora, esta tendência aumenta o número de comportamentos válidos de um programa paralelo e, consequentemente, contribui para aumentar, ainda mais, a complexidade da verificação do projeto. Nesse contexto, esta tese contribui para a verificação funcional do comportamento de operações em memória compartilhada ao se validar um *multicore chip* em tempo de projeto. A verificação funcional baseia-se na simulação da execução de programas não-sincronizados (uma vez que o comportamento não determinístico é chave para expor erros em memória compartilhada). Ela requer um gerador de programas de teste e um *checker* capaz de diagnosticar comportamentos anômalos em tempo de execução, a fim de sinalizar, o mais cedo possível, um erro de projeto.

Objetivos

Os objetivos usuais para a verificação funcional são: (1) aumentar a probabilidade de descoberta de erros de projeto, (2) aumentar a cobertura, e (3) reduzir o esforço necessário para encontrar erros ou para alcançar uma cobertura apropriada. O escopo desta tese é a geração de programas paralelos não sincronizados. Ela aborda a geração de testes (aleatórios e dirigidos) de maneira a atingir esses três objetivos usuais, além de um quarto objetivo: a reusabilidade de um gerador em diferentes projetos, diferentes variantes de protocolos de coerência, e diferentes ambientes de verificação.

Contribuições

Esta tese relata três contribuições científicas. A primeira contribuição consiste em duas novas técnicas que impõem restrições à geração aleatória convencional de programas de teste para viabilizar a verificação eficiente da memória compartilhada. Uma delas, denominada *chaining*, explora cadeias canônicas de dependências para restringir a geração aleatória de sequências de instruções (de forma a aumentar a cobertura de transições entre estados que são induzidas por eventos de conflito em um mesmo endereço); outra, denominada *biasing*, explora restrições sobre o espaço de endereçamento para restringir o assinalamento aleatório de endereços distintos (de forma a aumentar a cobertura de transições entre estados devidas a eventos de

evicção). A segunda contribuição é um novo mecanismo de geração dirigida por modelo de cobertura para melhorar a qualidade de testes não determinísticos. O mecanismo, denominado CTG (*coverage-driven test generation*), explora propriedades gerais de protocolos de coerência e de memórias *cache* para melhor controlar a cobertura de transições entre estados (a qual serve como *proxy* para aumentar a cobertura de acordo com a métrica que vier a ser adotada em um dado ambiente de verificação). A terceira contribuição é uma abordagem híbrida para a geração dirigida de testes, a qual não é uma simples combinação de uma técnica dirigida por dados com outra dirigida por modelo. A abordagem, denominada HTG (*hybrid test generation*) reformula a geração dirigida de testes como um problema de otimização com dois objetivos e explora vizinhanças para evitar a enumeração explícita do espaço de estados do protocolo de coerência, sem excluir soluções ótimas do espaço de pesquisa. Por serem independentes de métrica de cobertura, de protocolo de coerência, e de parâmetros de *cache*, as contribuições desta tese são reusáveis em diferentes projetos e em ambientes de verificação bem distintos.

Metodologia

Para validação e avaliação experimental das contribuições propostas, foram adotados os seguintes passos metodológicos: (1) uso de geradores pré-existentes (que capturam o estado-da-arte) para servir de base de comparação; (2) construção de um novo gerador para cada uma das contribuições propostas nesta tese; (3) síntese de erros artificiais de projeto para desafiar os geradores; (4) simulação da execução dos programas sintetizados por cada um dos geradores disponíveis (os novos e os pré-existentes) para diferentes representações de projetos (com e sem erros); e (5) comparação dos geradores em termos de eficácia (descoberta de erros), cobertura (fração dos estados e transições visitados), e esforço (tempo para detectar erros ou para atingir uma dada cobertura). Para servir como representações de projeto, adotaram-se modelos de processador, memória e interconexão (*O3*, *Ruby* e *simple*, respectivamente) disponíveis no ambiente de simulação gem5. A escolha desse ambiente se deu por razões pragmáticas, em especial pela disponibilidade em domínio público de uma variedade de complexos protocolos de coerência, mas também por viabilizar o uso de *checkers* localmente disponíves, cujo reuso foi crucial para a viabilidade deste trabalho em tempo. A execução foi simulada para representações de projeto com 8, 16 e 32 *cores* (todos com suporte a escalonamento dinâmico) e para protocolos populares, tais como MESI de três níveis e MOESI de dois níveis. A eficácia dos geradores foi avaliada com a injeção de diferentes arquétipos de erros de projeto, sob diferentes métricas de cobertura e para uma ampla configuração de parâmetros de geração (e.g. número de operações e de variáveis compartilhadas).

Resultados e Discussão

Ao avaliar a primeira contribuição desta tese em comparação com um gerador aleatório de testes convencional, observou-se que a aplicação das técnicas *chaining* e *biasing* tende a ampliar a capacidade de descoberta de erros de projeto, a aumentar a cobertura e a reduzir significativamente o esforço médio de verificação. Por exemplo, para projetos com 32 *cores*, pelo menos 50% do espaço de geração tornou-se capaz de expor erros, a mediana da cobertura aumentou em 44% e 83% nos dois níveis mais altos de hierarquia de memória, e o esforço médio de verificação foi reduzido em uma ordem de magnitude em vários casos. Ademais, os resultados experimentais observados – no âmbito de geração *aleatória* de testes – indicaram que uma simplificação da técnica genérica de *biasing* originalmente proposta poderia viabilizar a exploração *dinâmica* de restrições à formação de endereços, beneficiando assim a geração *dirigida* de testes, o que acabou motivando a segunda contribuição desta tese. Ao avaliar a segunda contribuição desta tese em comparação com o *McVersi Test Generator* (MTG) – que é um gerador dirigido por dados,

baseado em Programação Genética – observou-se que a técnica proposta de geração dirigida por modelo de cobertura (CTG) tende a atingir, em menor tempo, níveis de cobertura similares aos obtidos pelo MTG. Por exemplo, para projetos com 32 *cores*, ao executar testes com 1K operações, o CTG obteve a mesma cobertura máxima atingida pelo MTG, mas alcançou a cobertura de 60% em tempo dez vezes menor. Além disso, erros que o MTG levou cerca de uma hora para detectar foram expostos pelo CTG entre 5 a 30 minutos. Ao avaliar a terceira contribuição desta tese em comparação com o gerador MTG (puramente dirigido por dados) e o gerador CTG (puramente dirigido por modelo), observou-se que a abordagem proposta de geração híbrida (HTG) leva a uma melhor evolução da cobertura ao longo do tempo, mesmo quando aferida para diferentes protocolos de coerência. Por exemplo, para projetos com 32 *cores* e um protocolo MOESI de dois níveis, a abordagem HTG foi cerca de 8 a 19 vezes mais rápida para obter a máxima cobertura atingida pelo MTG e 2,7 vezes mais rápida para obter a máxima cobertura obtida pelo CTG. Para um protocolo MESI de 3 níveis, o HTG encontrou em 10 a 15 minutos, alguns erros que o MTG levou de 45 minutos a 7 horas para detectar. Finalmente, avaliou-se a terceira contribuição desta tese também em comparação com com um outro gerador puramente dirigido por dados, baseado em Aprendizado por Reforço, denominado RLG (*Reinforcement Learning Test Generator*). Embora o RLG tenha sido instrumentado para utilizar o mesmo módulo de RTG usado pelo HTG (de forma que ambos se beneficiem das técnicas de *chaining* e *biasing* nele aplicadas), observou-se que a abordagem de geração híbrida (HTG) é superior ao RLG em termos de evolução da cobertura para diferentes protocolos de coerência e métricas distintas de cobertura. Por exemplo, para projetos com 32 *cores* e um protocolo MESI de 3 níveis, o HTG atingiu a cobertura de 95,8% em 3 horas, enquanto o RLG levou 6 horas. Para um protocolo MOESI de 2 níveis, o HTG atingiu a cobertura de 95,26% em 2 horas, enquanto o RLG levou 6 horas.

Considerações Finais

A evidência experimental indica que as técnicas propostas nesta tese reduziram o esforço de verificação, melhoraram a descoberta de erros, e aumentaram a cobertura. Por exemplo, as técnicas *chaining* e *biasing* melhoraram a qualidade dos testes não determinísticos sintetizados através de geração aleatória de testes (Capítulo 3), geração dirigida puramente baseada em modelo (Capítulo 4), geração puramente dirigida por dados (Capítulo 6) e geração dirigida híbrida (Capítulos 5). Os resultados experimentais também indicam que, quando técnicas de aprendizado são usadas para a geração dirigida de testes (como a Programação Genética no MTG e o Aprendizado por Reforço no RLG), é improvável que elas exibam evolução de cobertura superior se: (1) não explorarem dinamicamente restrições que capture propriedades gerais de memória compartilhada (como ocorre com o MTG) e (2) não se basearem em algum modelo para guiar a evolução de cobertura enquanto ainda estiverem aprendendo (como acontece com o RLG e o MTG). Como se espera que a próxima geração de ferramentas de EDA utilize Aprendizado de Máquina para atingir alta cobertura em menor tempo, as descobertas relatadas nesta tese parecem indicar que, em futuras técnicas de geração, a inovação possa efetivamente vir de técnicas avançadas de aprendizado, desde que não se desprezar a herança recebida da longa tradição em geração aleatória de testes nem de modelos de cobertura eficientes.

Palavras-chave: *Multicores*. Memória compartilhada coerente. Verificação. Geração de testes.

ABSTRACT

Multiprocessors consist of tightly coupled processors that share some memory address space and are built with a single or multiple multicore chips. The shared-memory subsystem involves complex hardware components implementing a sophisticated coherence protocol whose RTL design is prone to errors. This thesis contributes to the functional verification of shared-memory behavior during the design of multicore chips. Since non-deterministic behavior is key to exposing shared-memory errors, non-synchronized parallel programs are often used for design verification and prototype test. However, in the verification phase, the slow execution in a simulator requires non-conventional techniques for enabling error exposure and high coverage with shorter programs. In this context, this thesis makes three contributions. The first contribution consists of two techniques that build upon conventional random test generation for efficient shared-memory verification. One technique exploits canonical dependence chains for constraining the random generation of instruction sequences (to raise the coverage of state transitions due to memory events conflicting at a same shared location), another exploits address space constraints for biasing random address assignment (to raise the coverage of state transitions due to eviction events). As compared to a conventional generator, their combination reduced the average verification effort by one order of magnitude in many cases. The second contribution is a new mechanism for directed generation that improves the quality of non-deterministic racy tests. The mechanism exploits general properties of coherence protocols and cache memories for better control on transition coverage (which serves as a proxy for increasing the actual coverage metric adopted in a given verification environment). Being independent of coverage metric, coherence protocol, and cache parameters, the mechanism is reusable across quite different designs and verification environments. Such a model-based generator was faster to reach similar coverage as obtained by a data-driven generator (based on Genetic Programming), reported in a related work. For instance, when executing tests with 1K operations for verifying 32-core designs, our test generator reached 60% coverage ten times faster. The third contribution consists of an hybrid approach that is not a simple combination of data-driven and model-based techniques. It reformulates directed test generation as a double-objective optimization problem, and it explores neighborhoods to avoid explicit enumeration of the coherence state space without excluding optimal solutions from the search space. As compared to purely data-driven and model-based generators, the hybrid approach led to superior coverage evolution with time, when targeting 32-core designs relying on different protocols. For instance, for a MOESI 2-level protocol, the approach was up to 2.7 faster than the model-based generator and around 5 to 19 times faster than the data-driven generator. Finally, the hybrid approach was also compared to a data-driven generator based on Reinforcement Learning. The experimental results showed that the proposed hybrid approach was 2 to 3 times faster to obtain the maximal coverage reached by that generator.

Keywords: Multicores. Coherent shared memory. Verification. Test generation.

LIST OF FIGURES

Figure 1 – An overview of the proposed framework 33

Figure 2 – Structure of the proposed generator 43
Figure 3 – Examples of chain categories 0, 1, 2 and 3 44
Figure 4 – How a canonical chain improves the coverage 46
Figure 5 – Example of address assignment 47
Figure 6 – Meaning of competition pattern 48
Figure 7 – Impact on functional coverage for random test generation 60

Figure 8 – How colliding operations avoid revisiting transitions 70
Figure 9 – How proposed variants traverse a plane of the generation space 73
Figure 10 – Coverage evolution for directed test generation 79

Figure 11 – The anatomy of the proposed Directing Engine 83
Figure 12 – Explorer and Driver at work 86
Figure 13 – Structural coverage evolution for the hybrid generator 94
Figure 14 – Functional coverage evolution for the hybrid generator 96

Figure 15 – Structural coverage evolution against Reinforced Learning 102
Figure 16 – Functional coverage evolution against Reinforced Learning 103

LIST OF TABLES

Table 1 – Relation between distinct terminologies for cache levels 36

Table 2 – Target mixes 50
Table 3 – Types of artificial design errors 52
Table 4 – Median improvement in coverage for random test generation 55
Table 5 – Fractions of generation space with potential for error exposure 56
Table 6 – Average improvement in effectiveness and in effort 59
Table 7 – Impact for SWMR violations 62
Table 8 – Impact for DV violations 62

Table 9 – How related works address test generation 66
Table 10 – Events inducing distinct classes of transitions 68
Table 11 – Studied design errors 77
Table 12 – Effort required for finding errors (I) 81
Table 13 – Effort required for finding errors (II) 81

Table 14 – DTG approaches for functional verification of multicore chips 84
Table 15 – Studied errors for MESI 3-level designs 92
Table 16 – Studied errors for MOESI 2-level designs 93
Table 17 – Time for finding errors in MESI 3-level for the hybrid generator 97
Table 18 – Time for finding errors in MOESI 2-level for the hybrid generator 98

Table 19 – Time for finding errors in MESI 3-level against Reinforced Learning 105
Table 20 – Time for finding errors in MOESI 2-level against Reinforced Learning 105

LIST OF ALGORITHMS

Algorithm 1 – The algorithm underlying the directing engine 75
Algorithm 2 – The algorithm underlying the generalized directing engine 87
Algorithm 3 – The algorithm underlying the Explorer 88
Algorithm 4 – The algorithm underlying the auxiliary routine Reduce Neighbor 88
Algorithm 5 – The algorithm underlying the auxiliary routine Generate Neighbor . . . 88
Algorithm 6 – The algorithm underlying the auxiliary routine Select Neighboring Points 89
Algorithm 7 – The algorithm underlying the Driver 90

LIST OF ABBREVIATIONS AND ACRONYMS

CP Competition Pattern.

CPU Central Processing Unit.

CTG Coverage-driven Test Generator.

DFSM Dichotomic Finite State Machine.

DTG Directed Test Generator.

DV Data Value.

EDA Electronic Design Automation.

FSM Finite State Machine.

HTG Hybrid Test Generator.

MCM Memory Consistency Model.

MESI Modified Exclusive Shared Invalid.

MOESI Modified Owned Exclusive Shared Invalid.

MOSI Modified Owned Shared Invalid.

MTG McVerSi Test Generator.

RLG Reinforcement Learning Test Generator.

RTG Random Test Generation.

RTL Register Transfer Level.

SWMR Single-Writer-Multiple-Reader.

CONTENTS

1 INTRODUCTION . . . 31
1.1 TRENDS IN MULTICORE CHIP DESIGN . . . 31
1.2 CHALLENGES OF SHARED-MEMORY VALIDATION . . . 32
1.3 SHARED-MEMORY FUNCTIONAL VERIFICATION GOALS . . . 33
1.3.1 Main approaches . . . **33**
1.3.2 Main engines and steps . . . **33**
1.3.3 Challenging problems tackled by this thesis . . . **34**
1.4 CONTRIBUTIONS . . . 34
1.5 METHODOLOGY . . . 36
1.6 ORGANIZATION OF THIS THESIS . . . 37
1.7 ACKNOWLEDGEMENTS . . . 37

2 FUNDAMENTAL CONCEPTS . . . 39
2.1 SHARED-MEMORY BEHAVIOR . . . 39
2.2 COLLISIONS AND CONFLICTS . . . 39
2.3 CANONICAL DEPENDENCE CHAINS . . . 40

3 RANDOM TEST GENERATION . . . 41
3.1 RELATED WORK . . . 41
3.2 THE PROPOSED GENERATION FLOW . . . 43
3.3 A CONCEPTUAL RE-ELABORATION . . . 43
3.3.1 Exploitation of chains for thread generation . . . 44
3.3.2 Exploitation of constraints for address assignment . . . 47
3.4 EXPERIMENTAL EVALUATION . . . 49
3.4.1 Experimental setup . . . 49
3.4.2 Metrics . . . 51
3.4.2.1 Metric 1: Potential for error exposure . . . 51
3.4.2.2 Metric 2: Effectiveness in error exposure . . . 53
3.4.2.3 Metric 3: Verification effort . . . 53
3.4.2.4 Metric 4: Functional coverage . . . 54
3.4.3 Broad assessment of impact . . . 55
3.4.3.1 Impact on coverage over the generation space . . . 55
3.4.3.2 Impact of parameter choice on error exposure . . . 55
3.4.3.3 Impact on effectiveness over joint exposure spaces . . . 58
3.4.3.4 Impact on effort over the entire generation space . . . 58
3.4.4 Assessment for a fixed core count . . . 58
3.4.4.1 Impact on functional coverage . . . 58
3.4.4.2 Impact on error exposure and effort . . . 61

3.5 CONCLUSIONS 63

4 DIRECTED TEST GENERATION 65

4.1 RELATED WORK 65

4.2 MAIN IDEAS BEHIND THE CONTRIBUTION 67

4.2.1 Proposed classification of transitions 67

4.2.2 Proposed constraints on RTG 68

4.2.2.1 Constraint 1: alternation between Classes 1 and 2 69

4.2.2.2 Constraint 2: uniform competition 69

4.2.3 Proposed coverage model 71

4.3 DESCRIPTION OF THE DIRECTING ENGINE 72

4.3.1 An example of how it works 72

4.3.2 The proposed algorithm 74

4.4 EXPERIMENTAL EVALUATION 76

4.4.1 Experimental setup 76

4.4.2 Experimental results 76

4.5 CONCLUSIONS 82

5 HYBRID DIRECTED TEST GENERATION 83

5.1 RELATED WORK 84

5.2 DTG FORMULATION AS AN OPTIMIZATION PROBLEM 85

5.3 THE DIRECTING ENGINE AT WORK: AN EXAMPLE 86

5.4 THE DIRECTING ENGINE: ALGORITHMS 87

5.4.1 The Data-Driven Explorer 87

5.4.2 The Model-Based Driver 89

5.5 EXPERIMENTAL EVALUATION 91

5.5.1 Experimental setup 91

5.5.2 Structural coverage evolution 93

5.5.3 Functional coverage evolution 95

5.5.4 Error discovery rate and detection time 97

5.6 CONCLUSIONS 98

6 A COMPARISON OF APPROACHES TO DIRECTED TEST GENERATION 99

6.1 A BRIEFING ON THE KEY IDEAS BEHIND THE RLG 99

6.2 EXPERIMENTAL EVALUATION 100

6.2.1 Structural coverage evolution 100

6.2.2 Functional coverage evolution 101

6.2.3 Error discovery rate and detection time 104

6.3 CONCLUSIONS 106

7 CONCLUSIONS AND PERSPECTIVES 107

BIBLIOGRAPHY 109

1 INTRODUCTION

Multiprocessors are "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, 2019). They are built with a single or multiple multicore chips. This thesis targets the *design verification* of the basic component of multiprocessors: the multicore chip.

This chapter first introduces the main architectural trends that affect multicore chip design and the resulting verification challenges. Then it clarifies the target problems that define the scope of this thesis. Finally, it summarizes the main contributions and the methodology used for their evaluation.

1.1 TRENDS IN MULTICORE CHIP DESIGN

Shared-memory behavior is defined by two complementary aspects: *coherence* defines *which* values can be returned by a load, and consistency defines *when* a written value will be returned by a load (PATTERSON; HENNESSY, 2020). The first aspect is related to the serialization of store operations to a *same* location. The second one is defined by the ordering between load and store operations to distinct locations, and to which extent stores are seen as atomic operations (ADVE; GHARACHORLOO, 1996).

Among the trends in multicore chip design, three aspects are important for defining the relevance and the scope of this thesis:

- **Multicore scaling limited by power.** It has long been shown that "multicore scaling is power limited to a degree not widely appreciated by the computing community" (ESMAEILZADEH et al., 2011; ESMAEILZADEH et al., 2012), as a result of the underutilization of integration capacity (aka dark silicon). However, even under such limited multicore scaling, the projected growth still leads to a major challenge in coherent shared-memory verification, because the protocol state space grows exponentially with the number of cores. Therefore, pragmatic techniques should not enumerate that space.

- **On-chip cache coherence.** Coherent shared-memory is expected in many multicore chips, especially those targeting general-purpose systems (DEVADAS, 2013). Although there is no consensus on the practical limit for coherent cores per chip, the techniques devised for graceful scaling up to tens of cores (MARTIN; HILL; SORIN, 2012) seem to be confirmed by recent examples like IBM Power 9, Xeon E7, and Fujitsu SPARC64X (HENNESSY; PATTERSON, 2019). Since many multicore chips are likely to be coherent, the exploitation of coherence properties to improve test generation seems pragmatic.

- **Relaxed and sequential consistency modes.** Most manufacturers have been building hardware that relax sequential consistency (IBM, 2019; ARM, 2018; WATERMAN;

ASANOVI, 2019). Sequential consistency (ADVE; GHARACHORLOO, 1996) is also often supported by some atomic instructions as an alternative mode. The main challenge in terms of consistency verification is how to efficiently check designs where the *non-multiple copy atomic* behavior of store instructions is architecturally visible (TRIPPEL et al., 2017), such as in Power and ARMv7, or is exposed in aggressive high-performance implementations, such as in the revised ARMv8 and RISC-V. Therefore, albeit the role of consistency is crucial for developing *checkers*, it is less important for building test *generators*. Indeed, to be reusable, a generator should be largely independent of the consistency model adopted by a given design. Although the exploitation of a specific consistency model for generation would limit reusability, any pragmatic generation technique should be able to handle the consistency requirements of sophisticated architectures, instead of being hampered by them. This trend increases the number of valid behaviors of a parallel program and, therefore, increases the complexity of design verification even further.

1.2 CHALLENGES OF SHARED-MEMORY VALIDATION

The practical feasibility of coherence in hardware – at least in the scope of a multicore chip for general-purpose systems – ends up challenging shared-memory validation due to the large protocol state space, even under the limitation imposed by power on core count. The relaxation of sequential consistency further increases the challenges.

The validation of the shared-memory system is challenged at different abstraction levels and distinct phases of the design cycle. The validation effort combines formal verification, e.g. Zhang et al. (2015), and simulation of the coherence protocol at the architectural level, relies on simulation-based functional verification of the design representation at the micro-architectural level (ADIR et al., 2004; FINE; ZIV, 2003; WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012; ELVER; NAGARAJAN, 2016), and finishes with the test of the multicore chip prototype. Shared-memory *test* relies on Random Test Generation (RTG) and post-mortem checking, e.g. Manovit & Hangal (2006). It can exploit long tests with hundreds of thousands of operations to reach high coverage, because the speed of the hardware prototype allows for suitable test throughput. Shared-memory *verification* usually relies on constrained random test generation and runtime checking to stop simulation as soon as a design error is found (ADIR et al., 2004; SHACHAM et al., 2008; FREITAS; RAMBO; SANTOS, 2013; GRAF et al., 2019). It should exploit short tests with tens of thousands of operations (ADIR et al., 2004), because the speed of the simulator would limit test throughput if much larger tests were used.[1]

The scope of this thesis is the *functional verification* of the coherent shared-memory behavior observed in a representation of a multicore chip at design time.

[1] In contrast, post-silicon testing usually relies on tests with tens of millions of operations (MANOVIT; HANGAL, 2006), which are directly executed on the hardware prototype.

1.3 SHARED-MEMORY FUNCTIONAL VERIFICATION GOALS

The focus of this thesis is on *test generation* for functional verification. The usual goals of functional design verification are to increase the probability of *bug discovery*, to raise *coverage*, and to reduce the *effort* required for finding errors or reaching proper coverage. This thesis addresses test generation towards meeting such usual goals, under the focus of an extra desirable feature: the *reusability* across distinct designs, different protocol variants, and distinct design environments.

1.3.1 Main approaches

To reach proper coverage with short tests, simulation-based functional verification relies on *directed* test generation, which appears under different approaches:

1. *Coverage-directed Random Test Generation (RTG)* targets the full system, and it is driven by metrics defined in test plans (FINE; ZIV, 2003; ADIR et al., 2004).
2. *Coherence-based test generation* targets the memory subsystem, and it is driven by the coverage of the full protocol space (QIN; MISHRA, 2012) or a decomposition of it (WAGNER; BERTACCO, 2008).
3. *Consistency-based test generation* targets memory consistency verification over the full system, and it can be driven by coverage metrics defined in test plans (ELVER; NAGARAJAN, 2016).

1.3.2 Main engines and steps

This thesis proposes novel techniques within a framework that relies on coverage-directed test generation for the functional verification of coherent shared memory. Figure 1 depicts the proposed framework. It includes a simulator for a full-system design representation of a multicore chip. The RTG engine relies on three parameters to constrain the building of a test program: the number of memory operations (n), the number of shared memory locations

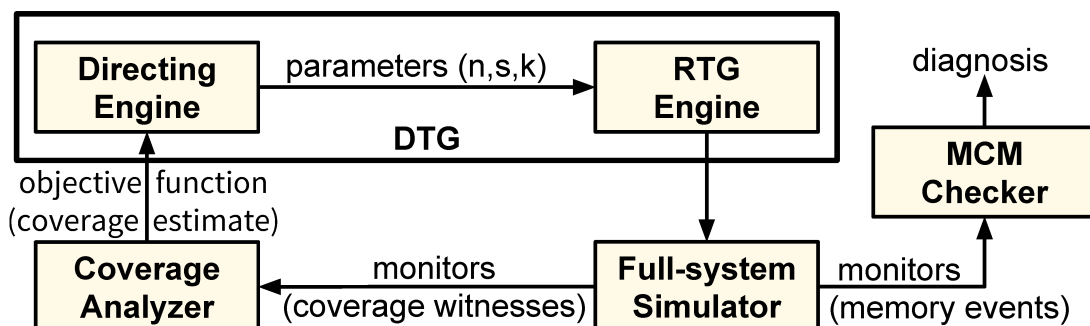


Figure 1 – An overview of the proposed framework.

(s), and the number of distinct cache sets to which the locations can be mapped (k). While the simulator executes a test program, monitors observe memory events at relevant points of each core domain. A checker analyzes the monitored events at runtime according to the axioms of the target Memory Consistency Model (MCM) (GHARACHORLOO, 1995), and it issues a positive diagnosis as soon as it detects an inconsistency. Besides, other monitors observe events that serve as coverage witnesses from which an analyzer computes the cumulative coverage of all tests executed so far. The directing engine takes that coverage value into account before selecting the next setting of parameters for RTG.

1.3.3 Challenging problems tackled by this thesis

To help building the envisaged verification framework, this thesis addresses three challenging problems:

1. Given a program size and a number of shared locations as parameters, how to improve the quality of non-deterministic tests by means of constrained random test generation so as to reduce the effort, raise bug discovery, and increase coverage?
2. Given a *range* of program sizes and a *range* for the number of shared locations, how to dynamically select proper generation parameters for increasing coverage in less time?
3. Given a *space* of generation parameters, how to find a *subspace* that maximizes coverage in minimum?

1.4 CONTRIBUTIONS

This thesis describes three contributions to test generation for shared-memory verification, each solving one of the mentioned challenging problems.

1. **The enforcement of non-conventional constraints for improving error discovery and coverage during *random* test generation.** It consists of a conceptual re-elaboration on two techniques proposed in previous works (ANDRADE; GRAF; SANTOS, 2016; ANDRADE, 2017), along with a more extensive experimental re-evaluation. In such early works, the conceptual focus was on exploiting memory *consistency* properties for reducing the effort of finding design errors. Instead, in the scope of this thesis, the conceptual focus is on exploiting cache *coherence* properties for increasing *coverage*. This shift in focus allowed a better comprehension of the impact of the techniques on coherent shared-memory verification, and it paved the way towards a *reusable* verification framework. Besides, it prompted the exploitation of techniques originally developed for improving the quality of *random* test generation for *directed* test generation. The first technique, called *chaining*, exploits canonical dependence chains for constraining the random generation of instruction sequences in such a way that the races induced at runtime are likely

to raise the coverage of state transitions due to collision events, i.e. due to memory operations colliding at a same location. The second technique, called *biasing*, exploits address space constraints for biasing random address assignment in such a way that the competition of distinct shared locations for a same cache set can be controlled for raising the coverage of state transitions due to cache eviction events. We built test generators relying on each of the proposed techniques, as well as on their combination, and we compared them to a conventional constrained random test generator for 8, 16, and 32-core architectures. Each of the four generators synthesized 1200 distinct test programs for verifying 10 faulty designs derived from each of the three architectures (144000 verification runs in total). For 32-core designs, the combination of the proposed techniques made at least 50% of the generation space capable of exposing errors, improved the median functional coverage by 44% and 83% at the two highest hierarchical levels, and reduced the average verification effort by one order of magnitude in many cases.

2. **The exploitation of non-conventional constraints for increasing coverage faster during *directed* test generation.** We propose a new mechanism that relies on a coverage model to improve the quality of non-deterministic tests. The technique dynamically exploits constraints capturing general properties of coherence protocols and cache memories for better control on transition coverage, which serves as a proxy for increasing the actual coverage metric adopted in a given verification environment. Being independent of coverage metric, coherence protocol, and cache parameters, the proposed technique is reusable across quite different designs and verification environments. As compared to a state-of-the-art generator of racy tests, the proposed technique reached a similar coverage much faster. For instance, when executing tests with 1K operations for verifying 32-core designs, the former reached 60% coverage around ten times faster than the latter. Besides, we identified challenging errors that could hardly be found by the latter within one hour, but were exposed by our technique in 5 to 30 minutes.

3. **The exploration of the generation space for increasing coverage even faster during *directed* test generation.** We propose a new approach that relies on a reformulation of directed test generation as a double-objective optimization problem. It explores neighborhoods without excluding optimal solutions from the search space while dynamically exploiting non-conventional constraints. This leads to a hybrid directed test generator that is not a simple combination of data-driven and model-based techniques. As compared to purely model-based and purely data-driven generators, the proposed technique leads to a better evolution of coverage with time. For instance, when targeting 32-core designs and a two-level MOESI protocol, the proposed approach was around 5 to 19 times faster to obtain the maximal coverage reached by a generator based on Genetic Programming, and it was 3 times faster to obtain the maximum coverage reached by generator based on Reinforcement Learning.

The second and the third contributions were reported in conference papers (ANDRADE et al., 2018; PFEIFER et al., 2020). The first and the third contributions were reported in journal articles (ANDRADE; GRAF; SANTOS, 2020; ANDRADE et al., 2020).

1.5 METHODOLOGY

For the experimental validation and evaluation of the proposed contributions, the following methodological steps were adopted: (1) use of pre-existing generators (capturing the state of the art) to serve as a basis for comparison; (2) building of a new generator for each of the proposed contributions; (3) synthesis of artificial design errors to challenge the generators; (4) simulation of the execution of the programs synthesized by each generator (pre-existing and new ones) for different design representions (with and without errors); (5) comparison of the generators in terms of effectivess (error discovery), coverage (fraction of all states or transitions visited), and effort (time for detecting errors or reaching proper coverage).

To serve as design representation, we adopted models for processor, memory, and interconnect (*O3*, *Ruby*, and *simple*, respectively) that are available in the gem5 simulation environment. The choice of such environment is due to pragmatic reasons, especially the availability (in the public domain) of a variety of complex coherence protocols, but also to allow the use of pre-existing checkers (locally available), whose role was crucial for making this work feasible in time. Simulations were performed for design representations with 8, 16, and 32 cores (each allowing out-of-order execution, as it is often the case for general-purpose applications) and for popular protocols such as 3-level MESI and 2-level MOESI. The effectiveness of the generators was evaluated with the injection of different archetypes of design errors, under different coverage metrics, and for a wide set of generation parameters.

To specify the distinct levels of cache, we adopted the terminology used in the gem5 environment, which does not always match the standard terminology used in computer architecture. To avoid confusion, Table 1 maps the relation between gem5's terminology and the standard one.

Table 1 – Relation between distinct terminologies for cache levels

Protocol	gem5	standard
MESI 3-level	L0 L1 L2	L1 L2 L3
MOESI 2-level	L1 L2	L1 L2

Coherence protocols are functionally specified by Finite State Machines (FSMs). A transition between two states is launched by an input event, and it induces an output event. The hardware implementation of the FSM is subject to mistakes when a functional specification is translated into a Register Transfer Level (RTL) description, which is often performed manually.

Let us illustrate this by means of a simple example. When the write-back policy is adopted by a coherence protocol (say MESI), the update of shared memory occurs either when a modified block is about to be replaced in a private cache (similarly to what happens in a single core) or when a memory block becomes shared by the private caches of two cores (HENNESSY; PATTERSON, 2019). This second case is detected when some core owns a block and a second core requests the same block after a miss in its private cache. Assume that a designer uses the miss event as the criterion to launch the write-back update of a modified block into shared memory. Although this criterion is sufficient for launching write-back, it is not a necessary condition: a block becomes shared when one core writes to it and a second core *reads* it, but it is not shared when the second core *writes* to the block (before reading it), because the second one invalidates the block in the cache of the first core. Therefore, the necessary and sufficient condition for detecting the sharing of a block between two cores is a read miss of a modified block (and not any miss). As a result of the mistake, two distinct transitions of the specified FSM (one with write-back as output event and another without it) would appear to be the same transition in the implemented FSM. Therefore, both transitions of the specified FSM should be covered by the execution of a test program in order to expose that design error. That is why our methodology evaluates the *coverage* of FSM transitions, and it relies on wrong transitions (or wrong output events associated with transitions) to implement artificial design errors for challenging the generators and evaluate their impact on *bug discovery*.

1.6 ORGANIZATION OF THIS THESIS

The remainder of this thesis is organized as follows. Chapter 2 summarizes the shared-memory concepts required for understanding the techniques proposed in later chapters. Chapter 3 describes the contribution to constrained random test generation (*chaining* and *biasing*). Chapter 4 reports the first contribution to directed test generation (*coverage-model-based test generation*). Chapter 5 describes the second contribution to directed test generation (*hybrid test generation*). Chapter 6 reports a comparison of model-based, data-driven, and hybrid approaches under the same level of constraint exploitation. Finally, Chapter 7 draws the overall conclusions and puts the work into perspective.

1.7 ACKNOWLEDGEMENTS

To implement the techniques described in this thesis and to perform the respective experiments, the author counted on the technical help and conceptual discussions with his fellow graduate students, especially Marleson Graf and Nícolas Pfeifer, which are co-authors of the publications that originally reported the contributions.

Thanks also to Eberle Rambo, Leandro Freitas, and Olav Henschel for the legacy infrastructure enabling this work.

2 FUNDAMENTAL CONCEPTS

This chapter defines concepts required to understand the key aspects of the techniques proposed in Chapters 3 and 4. First, it introduces the order relations required to specify shared-memory behavior. Then it presents two relations that can be exploited to improve the quality of directed test generation (Chapter 4). Finally, it reviews canonical multiprocessor dependence chains (GHARACHORLOO, 1995), which were originally proposed for specifying memory consistency, but are exploited as non-conventional constraints for improving the quality of test generation (Chapters 3 and 4).

2.1 SHARED-MEMORY BEHAVIOR

Let $(O_j)_a^i$ denote a memory operation O_j issued by processor i that makes a reference to some memory location a. We replace O by L or S to specify that the operation is either a load or a store, respectively. As shorthand notation, we sometimes drop either a superscript or a subscript. Given two operations O_j and O_m, if the instruction inducing the first operation precedes the instruction inducing the second in some thread, they are in *program order*, written $O_j \prec_{po} O_m$. Two operations are in *execution order*, written $O_j \leq O_m$, if the first one ends execution before the second one starts executing.

A test program may induce many executions with distinct outcomes. An execution induces a memory *behavior*. Every valid behavior must satisfy a partial order $\leq$ on the set of memory operations, which defines the cases in which the execution order is required to comply with the program order and with store serialization requirements. From the program order $\prec_{po}$, a Memory Consistency Model (MCM) specifies the allowed execution order $\leq$. The literature reports axioms formally defining $\leq$ for distinct MCMs (MANOVIT; HANGAL, 2006; FREITAS; RAMBO; SANTOS, 2013; ROY et al., 2006).

2.2 COLLISIONS AND CONFLICTS

We say that two memory operations *collide* if they make reference to the same memory location (ADIR et al., 2004). We say that two colliding operations *conflict* if at least one of them is a store (GHARACHORLOO, 1995). Formally, $(L_j)_a^i \leq (S_m)_a^k$, $(S_j)_a^i \leq (L_m)_a^k$, and $(S_j)_a^i \leq (S_m)_a^k$ denote colliding operations that conflict in execution order, while $(L_j)_a^i \leq (L_m)_a^k$ denotes colliding operations that do not conflict. Colliding operations induce memory events that may trigger transitions of the Finite State Machine (FSM) specifying the behavior of a coherence protocol for a private cache controller. We say that a collision (or conflict) is *intra-processor* or *inter-processor* when the operations involved are both issued by the *same* core ($i = k$) or by *distinct* cores ($i \neq k$), respectively.

2.3 CANONICAL DEPENDENCE CHAINS

To formally define canonical chains, we adopted a description idiom where simple barriers are used for enforcing program order between operations to distinct locations. Despite the idiom's simplicity, chains can be built with more complex fences without loss of generality, as explained in Chapter 3.

Definition 1. *Let MB be a memory barrier, i.e. a mechanism to restore program order between load and store operations whose order is relaxed by the memory model. We say that two operations are in significant program order, written $O_j \prec_{spo} O_m$, iff one of the following holds: $(O_j)_a \prec_{po} (O_m)_{b=a}$ or $(O_j)_a \prec_{po} MB \prec_{po} (O_m)_{b \neq a}$.*

Definition 2. *We say that two operations are in conflict order, written $O_j \leq_{co} O_m$, if and only if $(O_j)^i_a \leq (O_m)^k_{b=a}$ and at least one of them is a store.*

Definition 3. *We say that two operations are in significant conflict order, written $O_j \leq_{sco} O_m$, iff $(L_j)^i_a \leq (S_m)^k_a \vee (S_j)^i_a \leq (L_m)^k_a \vee (S_j)^i_a \leq (S_m)^k_a \vee (L_j)^i_a \leq S^x_a \leq (L_m)^k_a$.*

Definition 4. *A chain is a sequence $X \prec U \prec \cdots \prec V \prec Y$ whose endpoints X and Y are memory operations, but $U, \cdots, V$ may represent either memory operations or memory barriers. The relation $\prec$ between two successive elements denotes one of the relations $\prec_{po}, \prec_{spo}, \leq, \leq_{co}$, or $\leq_{sco}$. Let $\{A \prec B \prec\}_*$ denote zero or more pattern occurrences in the chain, and let $\{A \prec B \prec\}_+$ denote one or more.*

To specify chains, we use L, S, O to denote *types* of operations (respectively, load, store, any). We assume that distinct operation instances of each type will be used for building the chain. Gharachorloo (1995) specified a single category of uniprocessor chain and three categories of multiprocessor chains:

Category 0: $O^i_a \prec_{po} \{O^i_a \prec_{po} O^i_a \prec_{po}\}_* O^i_a$, where two successive elements cannot be of load type.

Category 1: $S^i_a \leq L^j_a \prec_{po} L^j_a$ or $S^i_a \leq L^j_a \prec_{po} S^j_a$, where $i, j \in \{1, \cdots, p\}$ and $i \neq j$.

Category 2: $O^i_a \prec_{spo} \{O^i_b \leq_{sco} O^j_b \prec_{spo}\}_+ O^j_a$, where $i, j \in \{1, \cdots, p\}$, $i \neq j$ and b is arbitrary.

Category 3: $S^i_a \leq_{sco} L^j_a \prec_{spo} \{O^j_b \leq_{sco} O^k_b \prec_{spo}\}_+ L^k_a$, where $i, j, k \in \{1, \cdots, p\}$, $i \neq j$, $j \neq k$ and b is arbitrary.

The next chapter gives examples of such chain categories.

3 RANDOM TEST GENERATION: ENFORCEMENT OF NON-CONVENTIONAL CONSTRAINTS

This chapter summarizes an early contribution to random test generation, which is a re-elaboration on previous work (ANDRADE, 2017; ANDRADE et al., 2018), as reported in a recently submitted article (ANDRADE; GRAF; SANTOS, 2020). It describes the key ideas to exploiting non-conventional constraints for building an efficient and effective random test generator to be used as the kernel of the coverage-directed test generators proposed in Chapters 4 and 5. The techniques described in this chapter are used to build an RTG Engine (for the framework depicted in Figure 1). For the detailed algorithms underlying the proposed random test generator, the reader is referred to previous work (ANDRADE, 2017; ANDRADE; GRAF; SANTOS, 2020).

This chapter is organized as follows. Section 3.1 summarizes related work. Section 3.2 gives an overview of the generation flow required to enforce non-conventional constraints on random test generation. Section 3.3 proposes a conceptual re-elaboration on the two techniques, *chaining* and *biasing* for the new focus on coherence verification. Section 3.4 reports an extensive re-evaluation of those techniques under the intended focus. Section 3.5 draws the chapter's conclusions.

3.1 RELATED WORK

Recall that a Memory Consistency Model (MCM) specifies rules defining not only the degree of program order relaxation, but also the extent of store atomicity (ADVE; GHARACHORLOO, 1996). There are two main shared-memory testing approaches that rely on MCMs: 1) the combination of litmus test generation with checking of valid execution witnesses; 2) the coupling of random test generation and memory-model checking.

The first approach exploits the MCM for automated generation of litmus tests (ALGLAVE et al., 2010; ALGLAVE et al., 2015; LUSTIG et al., 2017), i.e. short concurrent programs designed to stress certain MCM behaviors. The MCM declares which test outcomes are legal and which are not (LUSTIG et al., 2017). Each test is run thousands of times to provoke the behavior that the test characterizes (ALGLAVE et al., 2015). Despite its success in uncovering subtle bugs when testing commercial architectures (ALGLAVE et al., 2010), it has been shown that this approach has limited coverage when applied at design time (ELVER; NAGARAJAN, 2016).

In the second approach, memory-model checkers exploit the MCM for reducing the coupling between testing and implementation details. The paper by Hangal, Vahia, and Manovit (HANGAL et al., 2004) inspired many post-silicon checkers, e.g. Manovit & Hangal (2006), Roy et al. (2006), Chen et al. (2009), Hu et al. (2012), which elaborated on their original idea. This allowed for more reusable checkers and extended post-silicon testing beyond race-free self-checking tests and towards more effective random tests with intensive data races. However,

the claims that post-silicon checkers could be efficiently reused (HANGAL et al., 2004; HU et al., 2012) for pre-silicon verification only hold for their best effort versions, but not for their complete versions (offering verification guarantees) (MANOVIT; HANGAL, 2006; HU et al., 2012), whose poor scalability with growing core counts severely limits their reuse at design time.

The literature reports two classes of pre-silicon *runtime* checkers based on memory models. A relaxed scoreboard (SHACHAM et al., 2008) was proposed for keeping multiple valid events per entry. It employs an update rule that stores a new event after each write and dynamically removes events that become invalid after each read until an entry narrows down to a single memory event. Since it never reconsiders a previous decision, the technique admittedly may raise false negatives for a given test program. In contrast, another work (FREITAS; RAMBO; SANTOS, 2013) proposes the use of multiple verification engines (one per core) and a single global checker to build an axiom-based runtime checker with proven guarantees.

Industrial environments have been relying on random generators for processor verification since the mid-1980's. For instance, Genesys-Pro (IBM's third-generation engine) casts random test generation into a constraint satisfaction problem (ADIR et al., 2004). It offers a unified framework for handling the whole system. Albeit not originally intended to handle non-deterministic behavior, the framework was extended to allow the random generation of collision scenarios (ADIR; SHUREK, 2002), because programs with intensive data races expose bugs faster (HANGAL et al., 2004; SHACHAM et al., 2008). Besides, this observation has also fostered random test generation techniques specifically targeting the memory subsystem through memory-model checking, either for post-silicon test (HANGAL et al., 2004; ROY et al., 2006; MANOVIT; HANGAL, 2006; HU et al., 2012) or pre-silicon verification (SHACHAM et al., 2008; FREITAS; RAMBO; SANTOS, 2013).

As opposed to post-silicon testing, verification cannot afford long tests to achieve coverage goals. To reach similar goals with shorter tests, directed-test generation has been advocated (WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012; ELVER; NAGARAJAN, 2016). In face of the growing number of cores, one of the keys to scalability is the decomposition of the state space. In MCjammer (WAGNER; BERTACCO, 2008), each core is assigned an agent, which sees the coherence protocol in terms of a Dichotomic Finite State Machine (DFSM) (comprising only the states of the local node and the state of the environment). Cooperating agents formulate their coverage goals in terms of the DFSM, not the product FSM. Another technique (QIN; MISHRA, 2012) decomposes the state space into simpler structures such as hypercubes and cliques, which can be traversed (in an Euler tour) to avoid visiting the same transitions many times. It may allow full coverage with tests 50% shorter than a breadth-first traversal. In McVerSi (ELVER; NAGARAJAN, 2016), a genetic programming approach is used to progressively improve the quality of the test suite. It relies on a crossover function that prioritizes memory operations contributing to non-deterministic behavior.

Since directed-test generation often relies on some basic random generation engine, it can also benefit from improvements on constrained random test generation, as follows.

3.2 THE PROPOSED GENERATION FLOW

Figure 2 shows how the generation flow can be decomposed into interacting engines: thread generator, address assigner, and instruction synthesizer.

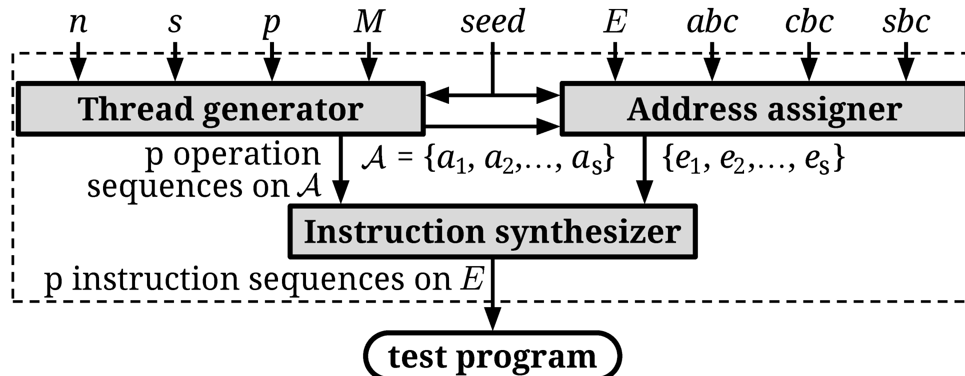


Figure 2 – Structure of the proposed random test generator.

Given the number of processors (p) of the architecture, the target numbers of memory operations (n) and shared locations (s) for the test program, a target mix of canonical dependence chains (M), and a random seed, the *thread generator* builds p random sequences (threads) containing each n/p operations with references to locations in the set $\mathscr{A} = \{a_1, a_2, \cdots, a_s\}$. To build the sequences, the generator exploits canonical chains (GHARACHORLOO, 1995) to increase the probability of error exposure and to foster higher coverage.

Given the address space (E), the *address assigner* maps *locations* $a_1, a_2, \cdots, a_s$ to effective *addresses* $e_1, e_2, \cdots, e_s$. The mapping relies on three types of biasing constraints to enforce desirable properties: 1) the *alignment biasing constraint* (*abc*) specifies the intended address stride, 2) the *sharing biasing constraint* (*sbc*) specifies whether true sharing must be enforced or not, 3) the *competition biasing constraint* (*cbc*) specifies how shared locations compete for cache sets.

The *instruction synthesizer* simply converts p sequences of memory *operations* referencing locations into p sequences of memory *instructions* referencing effective addresses. Besides, it also defines unique values to be written by store instructions, as required by most memory checkers (FREITAS; RAMBO; SANTOS, 2013; HANGAL et al., 2004; MANOVIT; HANGAL, 2006; HU et al., 2012).

3.3 A CONCEPTUAL RE-ELABORATION

This section informally address two previous contributions for improving the quality of random test generation:

1. A technique called *chaining* that exploits canonical multiprocessor dependence chains for constraining thread generation;

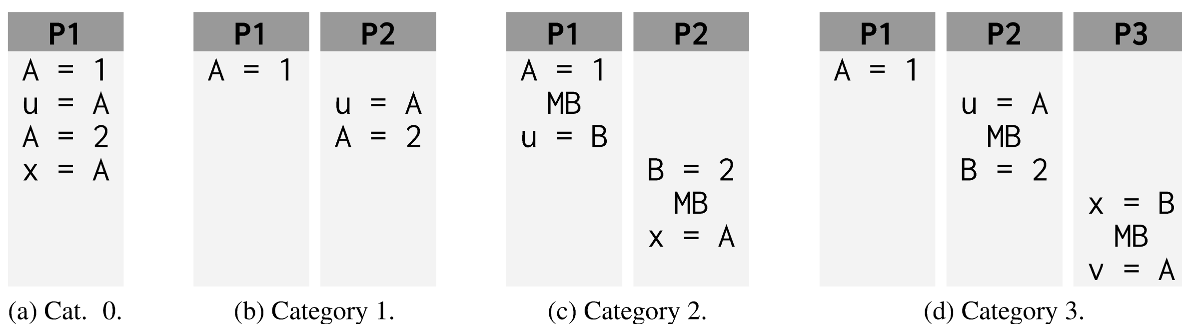
2. A technique called *biasing* that exploits a partitioning of the shared locations for constraining their effective addresses.

This section shows how and why chaining and biasing are likely to increase coverage and error detection.

3.3.1 Exploitation of chains for thread generation

The main idea behind thread generation is the exploitation of uniprocessor and multiprocessor dependence chains for stimulating as many distinct transitions as possible in the FSMs tracking the state of a block in multiple private caches. Let us illustrate that idea by means of examples. The examples assume that the program order between loads and stores to distinct locations can be relaxed, but it is certainly preserved when a memory barrier (MB) is inserted between them.

Let us first illustrate a few notions from Gharachorloo (1995), which are required for the examples. Two operations *conflict* if they collide at the same location and at least one of them is a store. Two operations from the same thread are in *significant program order* either if they conflict or if they are ordered by a memory barrier. Two operations from distinct threads are in *significant conflict order* either if they are in conflict order or if they are colliding loads with an intervening store in conflict order with them. Different *categories* of canonical chains can be defined by such significant orderings, as illustrated in Figure 3.



(a) Cat. 0. (b) Category 1. (c) Category 2. (d) Category 3.

Figure 3 – Examples of chain categories 0, 1, 2 and 3. Upper case letters denote variables in memory; lowercase letters, variables in registers.

Figure 3a shows a uniprocessor dependence chain with all operations in significant program order. Since non-deterministic behavior is key to the exposure of shared-memory errors (HANGAL et al., 2004; SHACHAM et al., 2008; ELVER; NAGARAJAN, 2016), Figures 3b, 3c, and 3d illustrate multiprocessor dependence chains that form data races between threads, but where operations in a same thread must execute in significant program order. Races are formed when operations from distinct threads are in conflict order.

In Figure 3b, the chain constrains the first two operations and its endpoints to form data races for location A, and the last two operations to significant program ordering. In Figure

3c, the chain constrains the first two and the last two operations to be in significant program order (by exploiting memory barriers). In Figure 3d, if the value 1 is observed for A in P2 and the value 2 is observed for B in P3, then the value 1 must be observed for A also in P3. If a multiprocessor chain is formed in execution time (as shown), the outcome of the data race involving its endpoints is deterministic, otherwise it is non-deterministic. Since each scenario induces distinct state transitions, their exploitation in different test runs tends to benefit coverage. Chains from categories 1, 2, and 3 not only drive the generator to form data races, but they also favor significant orderings. Such orderings tend to reduce the number of valid execution witnesses that do *not* lead to coherence events while the races increase the chances of detecting invalid ones. Both concur to raise the probability of error exposure and to improve coverage. That is why we exploit a mix of such categories.

Our technique exploits canonical chains not for enforcing specific *consistency* rules, but for favoring proper *coherence* events instead. Figure 4 shows the conceptual connection between a canonical chain and coherence events for different protocols. Note that, as the operations in the chain are executed, an intra-core conflict leads to local requests that induce *distinct* transitions in the local Finite State Machine (FSM), while an inter-core conflict leads to local and remote requests that also induce *distinct* transitions. Thus, the chain's *alternation* between intra- and inter-core conflicts tends to induce different transitions, which favors transition coverage. Since distinct protocols have similar responses for the same coherence transactions (except for a few transitions and write-back actions), this general property of a canonical chain makes the impact of our technique largely *independent* of the protocol implemented in a given design.

Chaining assumes the relaxation of program order for accesses to distinct locations only when it builds a *given* chain. Its focus on coherence does not restrain its applicability for four reasons:

1. the random selection of locations in distinct chains ensures that the stimulation of stronger orderings is not excluded from the generation space (albeit not favored in the scope of a given chain);

2. the checker is the guardian of memory model semantics (it may have to check orderings that are stronger than those relaxed for improving test quality);

3. albeit the use of barriers does not improve coherence coverage, chaining preserves them for defining *general* significant orderings (instead of enforcing *stricter* conflict orderings, as in Figure 4), because they enable the detection of consistency errors not tied to coherence mechanisms;

4. when complex fences (as in ARMv8 and Power8) replace simple barriers in canonical chains, our technique can handle sophisticated MCMs.

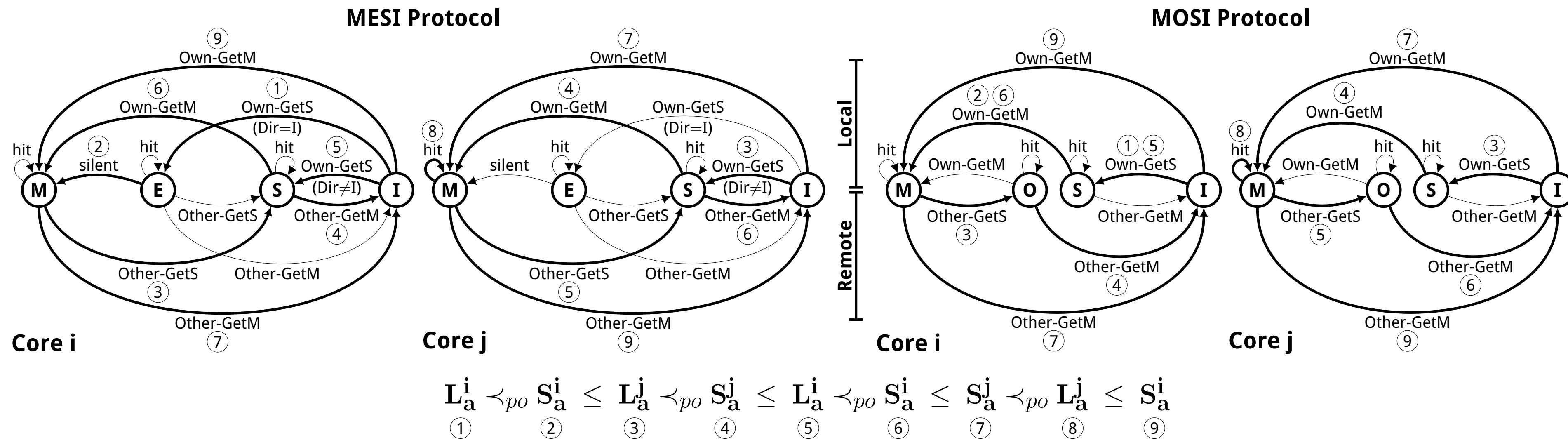


Figure 4 – How a canonical chain improves the coverage of coherence events independently of the protocol implemented in a given design. For each protocol, it shows the FSMs of the private cache controllers of two cores i and j. The coherence events are classified as local (Own) and remote (Other), according to the origin of the request (same or distinct core). Finally, numbers match each operation with the transition it triggers. The example considers that no cache initially holds a valid copy of the block. For the MESI protocol, the example assumes that the directory controller keeps the aggregate state of a block in all private caches and knows whether some cache holds a valid copy of a block ($Dir \neq I$) or none ($Dir = I$). For instance, the first operation in the chain (L^i_a) induces a local event (Own-GetS) in core-i's controller, and triggers a transition (I,S) for the MOSI protocol. For the MESI protocol, that local event triggers a transition (I,E), because no cache is initially holding a valid copy of the block. Note that the execution of that chain induces a sequence where transitions are rarely revisited (with the exception of transitions (I,S) and (S,M) for the MOSI protocol).

3.3.2 Exploitation of constraints for address assignment

The main idea behind address assignment is the competition biasing constraint (*cbc*). Let us illustrate that idea by means of an example. Figure 5 shows a layout corresponding to a 32-bit address space, but where 26-bit block addresses are actually represented, because it assumes blocks with 64 bytes.

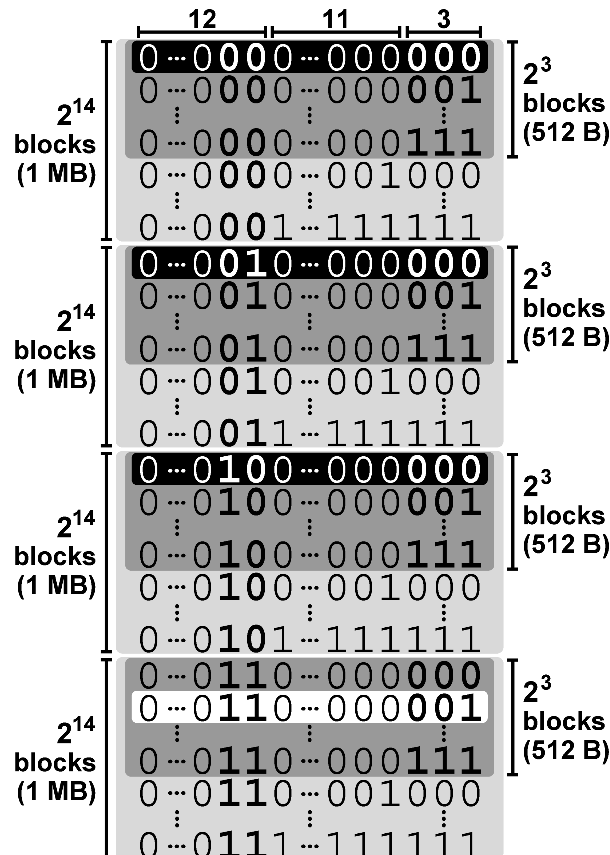


Figure 5 – An address assignment for four shared locations under address space constraints (assuming blocks with 64 bytes). A block address consists of a 12-bit tag and a 14-bit index where 11 bits are fixed but 3 may vary.

Suppose that the address range is limited to 4 MB, and it is partitioned into segments with 1MB each, as depicted by the light gray boxes. Suppose, however, that the address space to be exploited by the generator is constrained to only 2 KB in total, and it is uniformly distributed over the partitions in *useful* sub-ranges with 512 B each, as depicted by the dark gray boxes. Note that, due to the 1 MB partitioning, the block addresses can accommodate indices for identifying up to 2^{14} distinct cache sets. In spite of that, the constraint imposed on useful sub-ranges leads to a bound of 8 for the number of different index identifiers. Besides, the constraint imposed on the amount of partitions of the full address range leads to a bound of four for the number of distinct tags identifiers. This way of restricting the address space into a few useful chunks (as depicted by the dark gray boxes inside the light gray ones) is sometimes exploited, e.g. Elver & Nagarajan (2016), as a static address space constraint for fostering replacement events.

Instead, we propose a biasing technique that dynamically exploits address space constraints to foster replacement events *without* the need for restricting addresses to useful subranges, as explained next. Consider the black and white boxes lying inside the dark gray boxes. They represent the assignment of effective addresses to four distinct shared locations. Such assignment can be seen as an instance of a general pattern such that three locations compete for the same cache set, and a single location that does not compete with the others, because it is mapped to a distinct cache set. Similar assignment instances could be induced by such same *competition pattern* within the useful address space.

For a given number of shared locations, a *competition biasing constraint* $cbc = (\kappa, \chi)$ specifies competition patterns where all locations map to one out of κ cache sets, and at most χ locations map to the same cache set. For instance, the address assignment illustrated in Figure 5 was induced by $cbc = (2,3)$.

For a given *cbc*, there are different ways in which locations may compete for cache blocks, and they can be seen as patterns, each representing distinct ways of partitioning the set of locations, as follows. Let $set(a_i, a_j)$ denote that a_i and a_j compete for the same cache set. Let C be the set of locations competing for a same cache set, i.e. $C = \{(a_i, a_j) : a_i, a_j \in \mathscr{A} \wedge set(a_i, a_j)\}$. Since C is an equivalence relation, a *competition pattern* is a collection $CP = \{A_x\}$ forming a partition of the set $\mathscr{A}$ where each set A_x is an equivalent class induced by C.

Figure 6 illustrates the notion of Competition Pattern (CP) for a scenario with four locations and a cache with 2^I sets. For convenience of illustration, each equivalent class is represented as a clique of an undirected graph. Therefore, given a $cbc = (\kappa, \chi)$ inducing a CP, κ and χ could be seen as clique cover and chromatic numbers, respectively.

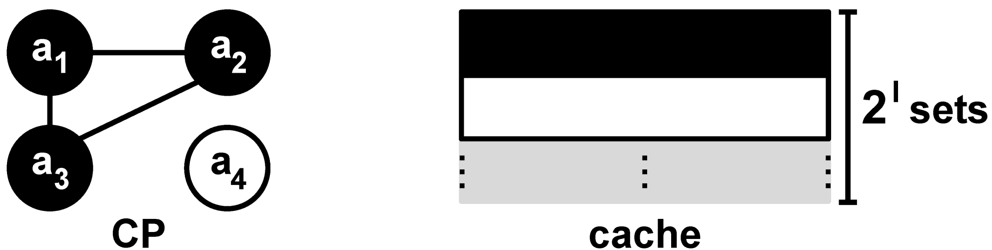


Figure 6 – Meaning of a competition pattern induced by $cbc = (2,3)$.

For a given number s of locations, a same *cbc* may induce a collection of CPs. Thus, to avoid limiting the verification space, the address assigner must be able to randomly select one among them. Notice, however, that not all pairs (κ, χ) represent feasible constraints. Fortunately, the collection of all feasible pairs for a given s can be easily precomputed.

Note that *cbc* constraints can be exploited for inducing cache evictions. Given an n-way cache, a block is evicted iff $n+1$ distinct and successive addresses compete for the same set; therefore, $\chi \geq n+1$ is a necessary condition for cache eviction. Besides, κ defines the number of distinct cache sets accessed by a test program.

Another desirable property of an address assignment is specified by the *sharing biasing constraint* (*sbc*). The address assignment in Figure 5 can be used to illustrate this notion. Note

that the locations competing for the same cache set have distinct block addresses: despite the same index, their tags are all different. Since unrelated shared variables are not stored in the same memory block, such assignment precludes false sharing. Actually, the *sbc* is a Boolean value specifying if true sharing must be enforced or not.

Yet another desirable property is specified by the *alignment biasing constraint* (*abc*). The *abc* is a natural number specifying that all effective addresses must be aligned to 2^{abc}-byte boundaries. For instance, if we enforce the alignment to 2^6 bytes, the six offset bits implicit in Figure 5 must be zero for all effective addresses to be exploited by the generator.

The motivation for mapping locations to effective addresses lies in the control of replacement events. For instance, the alternation between *cbc*s enabling and disabling block replacement tends to avoid revisiting the same state transition, which favors coverage and the probability of exposing design errors.

When multiple data races for distinct locations do not interfere with each other, the state transitions induced by each of them are quite similar. However, when they are coupled due to address space constraints (e.g. if two distinct locations are allocated in the same memory block or if they map to the same cache set), the induced state transitions tend to be rather different. That is why the combination of chaining and biasing is likely to lead to higher coverage than the application of each technique alone, as reported next.

3.4 EXPERIMENTAL EVALUATION

This section compares generators built with the proposed techniques to a conventional random test generator. It first describes the experimental conditions and defines the metrics adopted for comparison. Then it evaluates the generators according to each metric.

3.4.1 Experimental setup

Designs were derived for 8, 16, and 32-core architectures[1]. We relied on the pseudocode described in (FREITAS; RAMBO; SANTOS, 2013) to implement the checker and on the gem5 infrastructure (BINKERT et al., 2011) for simulation (*O3*, *Ruby*, and *simple* as CPU, memory, and interconnect models). We selected a 3-level MESI directory protocol that defines requests for read-only (GETS) and read-write (GETX, UPG) permissions, as well as eviction notifications for dirty blocks (PUTX). The hierarchy consisted of private L0 (split) caches, private L1 (unified) caches, and shared L2 cache, with 4KB (directed-mapped), 64KB (2-way), and 2MB (8-way), respectively, all operating with the same block size (64 bytes) and the same replacement policy (LRU).

We compared four generators. 1) PLAIN- is a conventional generator, which is similar to the ones used for memory-model checking, e.g. Hangal et al. (2004), Shacham et al. (2008). The sequence of operations forming a thread is obtained by randomly selecting the location

[1] In all designs, the adopted architecture was SPARC.

and the type of each operation independently of the choice made for other operations belonging to the same or some other thread. Address assignment is unconstrained (except for the obvious injection requirement), i.e. binary patterns are randomly selected from a specified address subspace. Since we could not find an implementation in the public domain, we relied on the pseudo-code reported by Rambo, Henschel & Santos (2011) to implement our own prototype. 2) PLAIN+ is similar to PLAIN-, but our biasing technique replaces the conventional unconstrained assigner. 3) CHAIN- exploits our chaining technique, but relies on a conventional unconstrained address assigner. 4) CHAIN+ combines chaining and biasing.

All generators have *common program parameters* that enforce general properties of the test to be generated: number of threads (p), number of memory operations (n), and number of shared locations (s), and a *common parameter for random generation* (*seed*). However, each generator has specific parameters tied to its inner mechanism. PLAIN- and PLAIN+ rely on *instruction* mixes specifying the target proportions of load, store, and membars (whose values were inspired by a related work (RAMBO; HENSCHEL; SANTOS, 2012). CHAIN- and CHAIN+ rely on *category* mixes specifying target proportions of chain categories (whose values[2] were obtained empirically). Table 2 shows the target mixes. Finally, PLAIN+ and CHAIN+ have common parameters for specifying biasing constraints.

Table 2 – Target mixes

Instruction mix			Category mix			
Load	Store	Membar	$\mathscr{C}_0$	$\mathscr{C}_1$	$\mathscr{C}_2$	$\mathscr{C}_3$
0.30	0.66	0.04	0.4	0.6	0	0
0.48	0.48	0.04	0	1	0	0
0.66	0.30	0.04	0	0.8	0.2	0
0.80	0.16	0.04	0	0.8	0	0.2

To verify each design, a distinct test suite was synthesized with each generator. We compared the generators for a same setting of the common program parameters by letting the others vary within pre-defined ranges. We call a *verification scenario* the collection of all random tests induced by a same setting of parameters (p, n, s) when distinct mixes and different seeds are explored.

To select ranges for the parameters, we relied on values reported from industrial environments (ADIR et al., 2004; MANOVIT; HANGAL, 2006; HU et al., 2012). Tests for post-silicon usage have hundreds of thousands of operations (MANOVIT; HANGAL, 2006; HU et al., 2012) and a few hundreds of shared locations (MANOVIT; HANGAL, 2006). Tests for pre-silicon usage typically have tens of thousands of operations (ADIR et al., 2004). Thus, since intensive data races are key to error exposure, the number of shared locations should be kept in the order of a few tens for reaching the same level of operation conflict required by

[2] When randomly selecting an operation for a chain, the generator employed the following probabilities: 0.75 for loads and 0.25 for stores.

the best post-silicon practices. Each generator synthesized a distinct suite with 1200 tests per architecture by exploring 15 random seeds $(1,\cdots,15)$, 4 target mixes, 4 amounts of shared locations ($s = 4, 8, 16, 32$), and 5 program sizes ($n =$ 1K, 2K, 4K, 8K, 16K), with K standing for 2^{10} operations. We constrained the shared locations within a 2^{25} address subspace.

For the generators PLAIN+ and CHAIN+, each verification scenario was constrained by a single *cbc*, exactly the same for both. For the experiments reported in this chapter, we selected $cbc = (1, s)$, because it fosters eviction as much as possible for the adopted range of s, allowing an evaluation of the proposed techniques that is largely independent of the associativity adopted at each hierarchical level. Besides, all addresses were aligned to the block ($abc = 2^6$) and true sharing was always enforced ($sbc = true$).

Since design errors are accidental, there is no standard collection of typical bugs that could be used as benchmark. Unfortunately, the errors reported in the literature are often tied to specific consistency models or are not described in sufficient detail to be properly reproduced. Therefore, as in many related works, we also had to rely on artificially injected errors to challenge the generators. We deliberately adopted coherence errors that are largely independent of consistency model and are easily reproducible. To build faulty designs, we used ten types of errors affecting cache controllers at levels L0, L1, and L2, as described in Table 3, which indicates modified (white) and new (gray) transitions. Errors were classified according to the violation of either Data Value (DV) or Single-Writer-Multiple-Reader (SWMR). We assumed that an error is systemic, i.e. it results in replicas in all controllers at the same level.[3] From a correct design, we derived ten *distinct* designs, each containing a *single* type of error.

For every architecture and every generator, we applied each test to ten designs, each containing an error from a different type (144000 runs in total). Then we determined whether or not an error was exposed for each of them. We say that a test program *exposes* a design error if it leads the checker to detect a *violation* of the memory model. To avoid that false negatives (positives) could underestimate (overestimate) error exposure, we used a presilicon runtime checker with verification guarantees (FREITAS; RAMBO; SANTOS, 2013).

Run times were measured on an HP xw8600 Workstation (Intel Xeon E5430, 2.66 GHz) with 8 GB of main memory.

3.4.2 Metrics

3.4.2.1 Metric 1: Potential for error exposure

To evaluate the *potential for error exposure* as measured by our experimental results, we determined whether or not a generator is likely to synthesize a test exposing the error under each (p, n, s) setting, and we obtained the fraction of all settings with potential for exposure (the higher the fraction, the smaller the sensitivity to parameter choice). Given two generators, to correlate their potentials for error exposure, we determined whether both, one, or none is

[3] Since most of the functionality of the protocol lies in the L1 controller, Table 3 focuses on errors at that level.

Table 3 – Types of artificial design errors (for reproducibility, the names of states, events, and actions match those in gem5's infrastructure).

	ID	Level	Current state	Input event	Next state	Output action
SWMR	D0	L1	IS	Data_Exclusive	EE instead of E	preserved
			I	WriteBack	I	popL0RequestQueue
			SS	WriteBack	SS	popL0RequestQueue
			M_I	WriteBack	M_I	popL0RequestQueue
			SINK_WB_ACK	WriteBack	SINK_WB_ACK	popL0RequestQueue
			EE	WriteBack	MM	same as in (E, MM)
	D1	L1	E	L0_Invalidate_when_GETX instead of L0_Invalidate_Else	EE instead of E_ILO	none
			E	L0_Invalidate_when_INV_UPG_GETS instead of L0_Invalidate_Else	E_ILO	preserved
			I	WriteBack	MM	same as in (E, MM) plus allocateCacheBlock
			S	WriteBack	S	popL0RequestQueue
	D2	L1	E	L0_Invalidate_when_GETX instead of L0_Invalidate_Else	EE instead of E_ILO	none
			E	L0_Invalidate_when_INV_UPG_GETS instead of L0_Invalidate_Else	E_ILO	preserved
			I	WriteBack	I	popL0RequestQueue
			S	WriteBack	S	popL0RequestQueue
	D3	L1	E	L0_Invalidate_when_GETS instead of L0_Invalidate_Else	EE instead of E_ILO	none
			E	L0_Invalidate_when_INV_UPG_GETX instead of L0_Invalidate_Else	E_ILO	preserved
			SS	WriteBack	SS	popL0RequestQueue
			S	WriteBack	S	popL0RequestQueue
DV	D4	L0	E	Store	M	dirty=1 precluded in store_hit
	D5	L1	M_ILO	L0_DataAck	EE instead of MM	preserved
	D6	L1	MM	Load	E instead of M	preserved
	D7	L2	MT	L2_Replacement	MCT_I instead of MT_I	preserved
	D8	L1	E_ILO	L0_DataAck	MM	writeDataFromL0Response precluded
	D9	L1	IS_I	DataS_fromL1	I	writeDataFromL2Response precluded

likely to expose an error under the *same* (p,n,s). We adopted the following procedure for each type of error and every scenario. For a given (p,n,s), we generated multiple random tests by exploring distinct seeds and mixes. Then we applied the multiple tests to a design containing a given error. If *at least one* test led to the detection of that error, we marked that scenario as "exposing". To correlate the potentials of a pair of generators, say G and g, we adopted the following procedure for each type of error. For each scenario (p,n,s), we checked whether or not both generators have marked it as "exposing" for a given error. If so, we labeled (p,n,s) as a scenario of *joint exposure*[4] (written $G.g$). If not, we labeled it as a scenario of *mutually-exclusive exposure*, depending on whether it was an "exposing" scenario either for G (written $G.\overline{g}$) or for g (written $g.\overline{G}$). Otherwise, it was labeled as a *joint non-exposure* scenario (written $\overline{G}.\overline{g}$). Finally, for a given type of error, we computed the fraction of all scenarios corresponding to each label.

3.4.2.2 *Metric 2: Effectiveness in error exposure*

To evaluate the *effectiveness* of a generator in exposing a given type of error under a scenario $v=(p,n,s)$, we measured the fraction $\varepsilon(v)$ of all tests induced by v for which violations were detected. Assuming sufficient sampling, this fraction can be interpreted as the probability of a generator to expose that type of error when the generation parameters are set to $v=(p,n,s)$. To calculate the average effectiveness for designs containing the same type of error, we took the arithmetic mean over the collection of all scenarios.

3.4.2.3 *Metric 3: Verification effort*

To estimate the *verification effort* of running the random tests induced by a scenario $v=(p,n,s)$ in an attempt to expose an error, we combined the effectiveness and the average test run times measured for a given design in scenario v. Let $T=\{T_i\}$ be the collection of tests induced by v and let t_i be the measured runtime for test T_i. Let $T^0 \cup T^1$ be a partition of T, where $T^0=\{T_j{}^0\}$ and $T^1=\{T_i{}^1\}$ are, respectively, the tests that do not expose an error and those that do. Let ε be the effectiveness measured in that scenario. Let us assume that all tests that do not expose an error (under a same scenario) take essentially the same time $\widehat{t^0}$ and let $\widehat{t^1}=\sum_{T_i^1 \in T^1} t_i^1/|T^1|$. As ε estimates the probability of a collection of random tests to expose an error, $1/\varepsilon$ serves as an estimation for the average number of tests required to expose that error. Since execution is stopped as soon as a test hits the error, the time to expose that error corresponds, on average, to the execution of a sequence of $\lceil 1/\varepsilon \rceil$ tests in which the first $\lceil 1/\varepsilon \rceil - 1$ tests do not expose the error, but the last one does, i.e. $(\lceil 1/\varepsilon \rceil - 1)\widehat{t^0} + t_j^1$. As there are exactly $|T^1|$ such sequences, by taking the arithmetic mean over them, we obtain the average

[4] Note that those under joint exposure can be interpreted as the collection of parameter settings for which the generators are correlated with respect to the potential for error exposure.

effort in scenario $v = (p, n, s)$:

$$EF(v) = \begin{cases} (\lceil 1/\varepsilon(v) \rceil - 1)\, \widehat{t^0}(v) + \widehat{t^1}(v) & \text{if } \varepsilon(v) \neq 0 \\ |T(v)|\, \widehat{t^0}(v) & \text{if } \varepsilon(v) = 0 \end{cases} \tag{3.1}$$

where $\widehat{t^1}(v) = \sum_{T_i^1 \in T^1(v)} t_i^1 / |T^1(v)|$ denotes the average run time of all tests that expose the error in scenario v.

Given two generators G and g, we determined the relative effort of G with respect to g as the ratio $EF_g(v)/EF_G(v)$. We obtained the average improvement for designs containing a same type of error by taking the geometric mean over the set of all v. Similarly, when both generators expose the error in the same scenario v, we determine the relative effectiveness of G with respect to g as the ratio $\varepsilon_G(v)/\varepsilon_g(v)$ [5].

Since the values obtained for the metrics above largely vary from one error type to another, we report them on a per-error-type basis, and we deliberately avoid taking the average over the collection of error types.

3.4.2.4 *Metric 4: Functional coverage*

We measured the *functional coverage* as the fraction of transitions covered in the state machine of each cache controller. For a design containing no errors, we tracked all memory blocks referenced by the collection of random tests induced by a given scenario $v = (p, n, s)$. We counted the number of *different* transitions taken in the machine tracking the state of the block corresponding to a given location a from the perspective of the cache owned by a given processor i at level L, written $TRAN_a^{i,L}(v)$. Then we computed the transition coverage, written $TC_a^{i,L}(v) = TRAN_a^{i,L}(v)/total(L)$, where $total(L)$ is the number of transitions of the state machine at level L. Next, we obtained the distribution of the transition coverage induced by v at level L, written $TC(v, L)$, i.e. the distribution of $TC_a^{i,L}(v)$ over all processors (i) and all locations (a). We also determined a similar distribution at the (shared) last-level cache, written $TC(v, L2)$. Finally, for each level L, we took the *median* over the collection of all scenarios v, written $\widehat{TC}(L)$. Given two generators G and g, we determined the relative coverage of G with respect to g as the ratio $\widehat{TC}_G(L)/\widehat{TC}_g(L)$.

Next, we report two distinct evaluation approaches: the former shows the *relative* improvement with respect to the baseline over the whole generation space; the latter, the *absolute* values for designs with a fixed core count.

[5] In this way, a ratio larger than one, either in effort or in effectiveness, can always be interpreted as an *improvement* of G with respect to g or, otherwise, as a degradation.

3.4.3 Broad assessment of impact

3.4.3.1 Impact on coverage over the generation space

Table 4 reports the relative coverage at all levels. It indicates that pure chaining does not improve the typical coverage. However, it shows that pure biasing improves the coverage most significantly at lower hierarchical levels. The fact that the improvement increases from the highest to the lowest level shows how ineffective random address assignment is in face of the progressively larger associativities towards the lowest level. When combined, the techniques led to the highest improvement at all levels and every core count. Biasing was the largest contributor to the combined improvement at L2; chaining, the largest contributor at L0. This is a first evidence of the complementary nature of the proposed techniques, as it will be explained in Section 3.4.4.

Table 4 – Median improvement in coverage over the entire generation space

p	Chaining			Biasing			Chaining and Biasing		
	CHAIN- w.r.t **PLAIN-**			**PLAIN+** w.r.t **PLAIN-**			**CHAIN+** w.r.t **PLAIN-**		
	L0	L1	L2	L0	L1	L2	L0	L1	L2
8	1.05	0.96	1.02	1.11	1.68	3.39	1.21	1.86	3.46
16	1.00	0.96	1.00	1.12	1.72	3.29	1.35	1.92	3.33
32	1.00	0.96	1.00	1.19	1.67	3.67	1.44	1.83	3.67

The significant improvement in coverage observed for the combination of the proposed techniques is a general evidence of higher chances of exposing design errors in less time. To provide further support to that claim, we measured the potential, the effectiveness, and the effort to expose actual errors over a collection of faulty designs, as follows.

3.4.3.2 Impact of parameter choice on error exposure

Table 5 shows the fractions of verification scenarios with potential for error exposure. Note that a value in that table represents an optimistic estimate for the actual effectiveness. For instance, an entry containing a unit value does not mean that all tests are able to expose a given type of error. Instead, it means that all *verification scenarios* have potential of exposing it, but the actual detection is largely independent of the (p,n,s) parameters: either it may depend on the probabilistic parameters (seed and mix) or on the specific features of each generator. For each core count, the table has two segments: one reporting the correlation of potential exposure for pairs of generators, another showing the overall exposure of each generator individually.

For each core count, the first row captures the fraction of the generation space where the features common to the baseline and the proposed generators were sufficient for error exposure. Such common features narrow down to the random choice of operations and addresses, as well as the exploitation of conventional constraints (e.g. target operation mix). The results

Table 5 – Fractions of generation space with potential for error exposure (P-, C-, P+, and C+ are acronyms for PLAIN-, CHAIN-, PLAIN+, and CHAIN+).

Chaining

Cores	Exposure	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
8 cores	P− . C−	0.55	0.40	0.25	0.50	0.00	0.00	0.00	0.00	0.15	0.15
	P− . $\overline{\text{C}}$−	0.05	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.05	0.10
	$\overline{\text{P}}$− . C−	0.05	0.25	0.25	0.05	0.00	0.00	0.00	0.00	0.30	0.05
	$\overline{\text{P}}$− . $\overline{\text{C}}$−	0.35	0.35	0.50	0.35	1.00	1.00	1.00	1.00	0.50	0.70
	PLAIN−	0.60	0.40	0.25	0.60	0.00	0.00	0.00	0.00	0.20	0.25
	CHAIN−	0.60	0.65	0.50	0.55	0.00	0.00	0.00	0.00	0.45	0.20
16 cores	P− . C−	0.65	0.40	0.30	0.55	0.00	0.00	0.00	0.00	0.15	0.15
	P− . $\overline{\text{C}}$−	0.05	0.05	0.10	0.05	0.00	0.00	0.00	0.00	0.05	0.20
	$\overline{\text{P}}$− . C−	0.05	0.10	0.15	0.10	0.00	0.00	0.00	0.00	0.35	0.10
	$\overline{\text{P}}$− . $\overline{\text{C}}$−	0.25	0.45	0.45	0.30	1.00	1.00	1.00	1.00	0.45	0.55
	PLAIN−	0.70	0.45	0.40	0.60	0.00	0.00	0.00	0.00	0.20	0.35
	CHAIN−	0.70	0.50	0.45	0.65	0.00	0.00	0.00	0.00	0.50	0.25
32 cores	P− . C−	0.60	0.45	0.35	0.35	0.00	0.00	0.00	0.00	0.00	0.10
	P− . $\overline{\text{C}}$−	0.20	0.20	0.20	0.25	0.00	0.00	0.00	0.00	0.05	0.10
	$\overline{\text{P}}$− . C−	0.05	0.10	0.20	0.15	0.00	0.00	0.00	0.00	0.45	0.10
	$\overline{\text{P}}$− . $\overline{\text{C}}$−	0.15	0.25	0.25	0.25	1.00	1.00	1.00	1.00	0.50	0.70
	PLAIN−	0.80	0.65	0.55	0.60	0.00	0.00	0.00	0.00	0.05	0.20
	CHAIN−	0.65	0.55	0.55	0.50	0.00	0.00	0.00	0.00	0.45	0.20

Biasing

Cores	Exposure	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
8 cores	P− . P+	0.60	0.40	0.15	0.35	0.00	0.00	0.00	0.00	0.20	0.20
	P− . $\overline{\text{P}}$+	0.00	0.00	0.10	0.25	0.00	0.00	0.00	0.00	0.00	0.05
	$\overline{\text{P}}$− . P+	0.40	0.60	0.65	0.40	0.60	0.65	0.50	0.60	0.80	0.65
	$\overline{\text{P}}$− . $\overline{\text{P}}$+	0.00	0.00	0.10	0.00	0.40	0.35	0.50	0.40	0.00	0.10
	PLAIN−	0.60	0.40	0.25	0.60	0.00	0.00	0.00	0.00	0.20	0.25
	PLAIN+	1.00	1.00	0.80	0.75	0.60	0.65	0.50	0.60	1.00	0.85
16 cores	P− . P+	0.70	0.45	0.25	0.45	0.00	0.00	0.00	0.00	0.20	0.25
	P− . $\overline{\text{P}}$+	0.00	0.00	0.15	0.15	0.00	0.00	0.00	0.00	0.00	0.10
	$\overline{\text{P}}$− . P+	0.30	0.50	0.35	0.40	0.50	0.75	0.50	0.70	0.80	0.55
	$\overline{\text{P}}$− . $\overline{\text{P}}$+	0.00	0.05	0.25	0.00	0.50	0.25	0.50	0.30	0.00	0.10
	PLAIN−	0.70	0.45	0.40	0.60	0.00	0.00	0.00	0.00	0.20	0.35
	PLAIN+	1.00	0.95	0.60	0.85	0.50	0.75	0.50	0.70	1.00	0.80
32 cores	P− . P+	0.80	0.65	0.30	0.35	0.00	0.00	0.00	0.00	0.05	0.15
	P− . $\overline{\text{P}}$+	0.00	0.00	0.25	0.25	0.00	0.00	0.00	0.00	0.00	0.05
	$\overline{\text{P}}$− . P+	0.20	0.35	0.15	0.40	0.50	0.75	0.50	0.60	0.95	0.70
	$\overline{\text{P}}$− . $\overline{\text{P}}$+	0.00	0.00	0.30	0.00	0.50	0.25	0.50	0.40	0.00	0.10
	PLAIN−	0.80	0.65	0.55	0.60	0.00	0.00	0.00	0.00	0.05	0.20
	PLAIN+	1.00	1.00	0.45	0.75	0.50	0.75	0.50	0.60	1.00	0.85

Chaining and Biasing

Cores	Exposure	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
8 cores	P− . C+	0.60	0.40	0.25	0.60	0.00	0.00	0.00	0.00	0.20	0.25
	P− . $\overline{\text{C}}$+	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\overline{\text{P}}$− . C+	0.40	0.60	0.75	0.40	0.65	0.70	0.60	0.70	0.80	0.65
	$\overline{\text{P}}$− . $\overline{\text{C}}$+	0.00	0.00	0.00	0.00	0.35	0.30	0.40	0.30	0.00	0.10
	PLAIN−	0.60	0.40	0.25	0.60	0.00	0.00	0.00	0.00	0.20	0.25
	CHAIN+	1.00	1.00	1.00	1.00	0.65	0.70	0.60	0.70	1.00	0.90
16 cores	P− . C+	0.70	0.45	0.40	0.60	0.00	0.00	0.00	0.00	0.20	0.35
	P− . $\overline{\text{C}}$+	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\overline{\text{P}}$− . C+	0.30	0.55	0.60	0.40	0.60	0.75	0.55	0.70	0.80	0.55
	$\overline{\text{P}}$− . $\overline{\text{C}}$+	0.00	0.00	0.00	0.00	0.40	0.25	0.45	0.30	0.00	0.10
	PLAIN−	0.70	0.45	0.40	0.60	0.00	0.00	0.00	0.00	0.20	0.35
	CHAIN+	1.00	1.00	1.00	1.00	0.60	0.75	0.55	0.70	1.00	0.90
32 cores	P− . C+	0.80	0.65	0.55	0.60	0.00	0.00	0.00	0.00	0.05	0.20
	P− . $\overline{\text{C}}$+	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\overline{\text{P}}$− . C+	0.20	0.35	0.45	0.40	0.50	0.75	0.50	0.60	0.95	0.75
	$\overline{\text{P}}$− . $\overline{\text{C}}$+	0.00	0.00	0.00	0.00	0.50	0.25	0.50	0.40	0.00	0.05
	PLAIN−	0.80	0.65	0.55	0.60	0.00	0.00	0.00	0.00	0.05	0.20
	CHAIN+	1.00	1.00	1.00	1.00	0.50	0.75	0.50	0.60	1.00	0.95

indicate that the role of such features vary among designs: it might be significant for some (D0-D3, D8, D9), but negligible for others (D4-D7).

The second row shows the fraction of the generation space where the baseline generator exposes an error that is not exposed by one of the proposed generators. Under pure chaining, that fraction was not negligible for a few designs (32-core D0-D3, 16-core D9). Under pure biasing, that fraction was zero or negligible for all designs but three (32-core D2-D3, 8-core D3). When the techniques were combined, that fraction was zero for all designs. Thus, the baseline generator is rarely superior to a biased generator, especially when it also exploits chaining.

The third row shows the fraction of the generation space in which one of the proposed generators exposed an error that was not exposed by the baseline generator. Under pure chaining that fraction was significant for a few designs (D1, D2, D3, and D8). In contrast, biasing significantly increased that fraction (whether pure or combined). The combination of biasing and chaining was superior over at least 20% and at most 95% of the generation space, as compared to the baseline. Thus, in general, a biased generator is superior for a significant fraction of the generation space, especially when it also exploits chaining.

The fourth row shows the fraction of the space where neither the features common to all generators nor the proposed complementary features were able to foster exposure. Biasing significantly reduced that fraction as compared to pure chaining, except for one design (32-core D2). The combination of biasing and chaining completely ruled out the joint non-exposure subspace for half of the designs (D0-D3 and D8). The other half (D4-D7 and D9) gives evidence of challenging errors that can only be exposed if extra constraints (such as the proposed ones) prune random test generation.

The last two rows show the overall impact of each generator. Pure biasing increased the fraction of the space leading to the exposure of errors in all designs but one (32-core D2). This indicates that biasing, in general, makes error detection less dependent on the choice of parameters, i.e. it makes constrained random choice useful over a larger fraction of the generation space. Indeed, within the ranges adopted for the parameters, pure biasing made detection practically independent of the choice of n and s for three designs (D0, D1, D8), regardless of core count. When biasing was combined with chaining, the fraction leading to exposure increased in *all* designs as compared to the baseline, and at least 50% of all scenarios became suitable for exposing errors. The combination made detection practically independent of the choice of n and s for six designs (D0, D1, D2, D3, D8, D9) instead of three. This shows one of the contributions of chaining: it may help in extending the potential for error exposure over a larger number of scenarios. Another contribution of chaining (which cannot be inferred from Table 5) comes into play when a given scenario is selected for a test. Chaining tends to increase the probability of error detection, making constrained random choice more effective and efficient, as shown next.

3.4.3.3 Impact on effectiveness over joint exposure spaces

The top of Table 6 reports the relative effectiveness over joint exposure subspaces. For four designs (D4-D7), the baseline generator was not able to expose errors with test lengths between 1K and 16K (empty subspaces are indicated by dashes). Pure chaining kept or slightly improved the effectiveness for all designs but two (16/32-core D9). In contrast, pure biasing significantly improved the effectiveness for all designs but two (16/32-core D2). Finally, the combination of biasing and chaining improved the effectiveness for all designs. Note that the combination of the techniques led to the largest improvements with respect to the baseline for all designs but three (8/16/32-core D9). This can be explained by the fact that the structure of canonical multiprocessor chains foster operation conflict, which is the main mechanism required to expose errors in most designs.

When two generators are compared in *identical* verification scenarios where *both* expose an error, the improvement in effectiveness is directly reflected as an improvement in effort. However, since test suites usually exploit distinct settings of parameters, the impact on effort over the entire generation space provides a better assessment for test throughput. That is why we relaxed the joint exposure requirement to evaluate the impact on effort, as follows.

3.4.3.4 Impact on effort over the entire generation space

The bottom of Table 6 shows the relative effort over the entire generation space, whether both, one, or none of the generators under comparison happened to expose a given type of error. Pure chaining hardly improved the effort. In contrast, pure biasing significantly improved the effort for all but three designs (32-core D3, 16/32-core D2). The combination of biasing and chaining improved the effort for all designs. As compared to pure biasing, the combination led to large reductions in effort, which resulted from the gains in effectiveness under joint exposure (Table 6) and from the higher numbers of scenarios leading to error exposure (Table 5). Albeit the combination worsened the effort for three designs (32-core D7, 16/32-core D9), the maximum degradation (32%) was much smaller than the maximum improvement (15 times).

The experimental evidence indicates that pure chaining does not pay off. However, it indicates that address biasing not only pays off, but it also favors proper operation chaining, which tends to further reduce the verification effort.

3.4.4 Assessment for a fixed core count

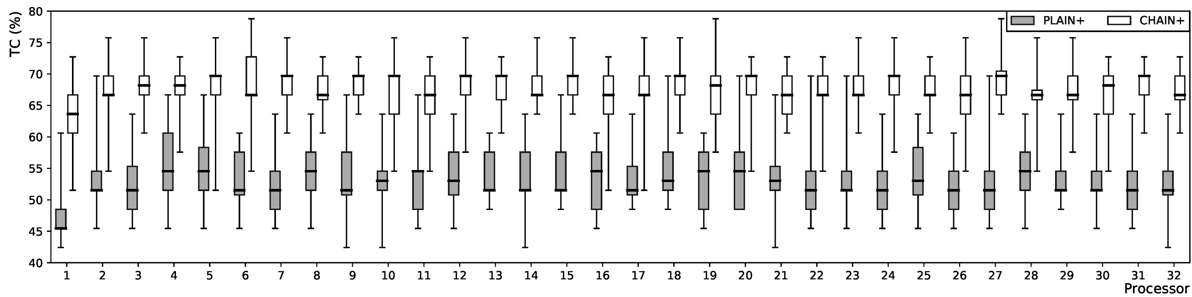
3.4.4.1 Impact on functional coverage

Figure 7 displays the distributions $TC(v, L0)$ and $TC(v, L1)$ resulting from the execution of all random tests induced by the scenario $v = (p, n, s) = (32, 4K, 32)$. Each box represents the

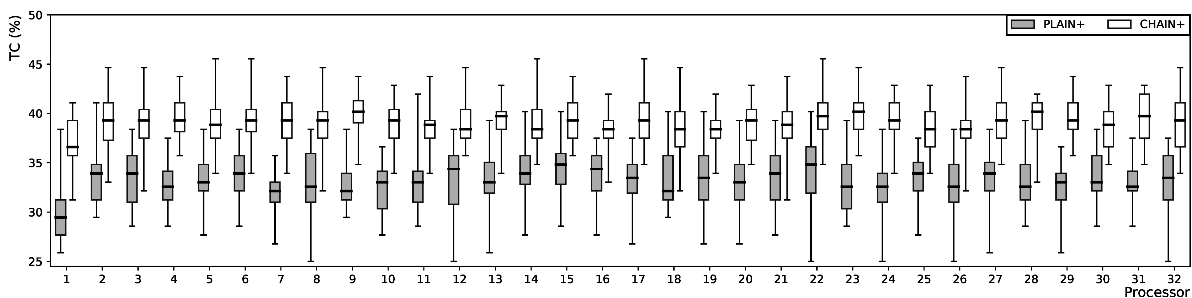
Table 6 – Average improvement in effectiveness (under joint exposure subspaces) and in effort (over the entire generation space).

Metric	p	**Chaining** **CHAIN-** w.r.t **PLAIN-**										**Biasing** **PLAIN+** w.r.t **PLAIN-**										**Chaining and Biasing** **CHAIN+** w.r.t **PLAIN-**									
		D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
$\varepsilon_G/\varepsilon_g$	**8**	1.3	1.1	1.1	1.0	—	—	—	—	1.3	1.0	5.9	10.3	1.7	4.3	—	—	—	—	16.1	18.1	21.9	22.8	18.0	13.5	—	—	—	—	55.0	16.7
	16	1.5	1.1	1.0	1.5	—	—	—	—	1.3	0.6	7.3	10.3	1.0	2.3	—	—	—	—	10.1	7.8	25.7	18.2	13.3	10.1	—	—	—	—	41.8	4.4
	32	1.0	1.4	1.2	1.1	—	—	—	—	—	0.4	2.4	9.6	0.8	1.2	—	—	—	—	2.0	12.1	15.5	19.5	9.8	5.7	—	—	—	—	23.0	2.7
EF_g/EF_G	**8**	1.2	1.1	1.1	1.1	1.0	1.0	1.0	1.0	1.3	1.0	6.7	11.8	1.9	2.5	3.9	7.1	1.5	6.0	7.8	12.5	24.9	29.2	21.1	15.8	8.6	11.7	2.5	6.7	29.7	14.9
	16	1.3	1.0	1.0	1.4	1.0	1.0	1.0	1.0	1.2	0.8	4.6	7.3	0.7	1.9	4.0	10.5	1.4	5.3	5.7	8.1	20.4	17.4	10.6	9.4	7.8	15.4	1.9	6.1	26.7	7.5
	32	1.0	1.2	1.1	1.0	1.0	1.0	1.0	1.0	1.2	1.0	2.1	6.4	0.6	1.0	2.8	12.0	1.8	5.9	3.2	9.8	12.2	12.2	8.5	6.7	6.0	15.4	1.9	5.7	19.4	6.7

distribution from the perspective of each private cache. Since it is clear from Section 3.4.3 that only biased generators can significantly improve coverage, Figure 7 omits the plots for PLAIN- and CHAIN-, whose typical transition coverage was around 19-21% at L1 and 45-49% at L0. As compared to those values, Figure 7 shows that biasing significantly improved coverage whether alone (PLAIN+) or combined with chaining (CHAIN+). Indeed, the combination led to the highest coverage for all private caches at the same level[6].



(a) Transition coverage distribution at level L0



(b) Transition coverage distribution at level L1

Figure 7 – Impact on the functional coverage of private cache controllers for 32-core designs (for tests with 4K operations and 32 locations).

The higher coverage of the combination of chaining with biasing can be explained as follows. Chaining and biasing foster, respectively, conflict and eviction events. Such different classes of events often induce *distinct* transitions in the state machine controlling a private cache, which contributes to raising the coverage. At all levels, biasing fosters eviction-induced transitions. At L0, biasing also fosters transitions from state I, whereas chaining fosters transitions from states E, S, and M either through intra-processor conflict (i.e. (S,M) and (E, M)) or through inter-processor conflict (i.e. (M,I), (E,I), and (S,I)). At L1, biasing fosters transitions induced by GETS and GETX requests from L0, while chaining fosters transitions induced not only by UPG requests from L0, but also requests from other cores (Invalidate, FWD_GETS, FWD_GETX). Note that the impact of combining biasing and chaining is higher at L0 than at L1. One reason for that results from the exploitation of inclusive caches: operation conflicts raising distinct types of requests (Invalidate, UPG, FWD_GETS, FWD_GETX) often trigger the

[6] Notice also that the distribution over the processors is more uniform for CHAIN+ than for PLAIN+, and the former typically exhibits smaller dispersion. Figure 7b shows a similar behavior at L1. Figure 7a shows that, while the typical coverage at L0 was around 52% for PLAIN+, it became approximately 67% for CHAIN+. Albeit the typical coverage was around 33% for PLAIN+, it became approximately 40% for CHAIN+.

same transition at L1. Another reason is the smaller frequency of evictions due to the higher associativity at L1.

At L2 (where the shared cache also plays the role of directory), the median coverage was 15% for both PLAIN- and CHAIN- and 53% for *both* PLAIN+ and CHAIN+. Chaining had marginal impact, because most requests resulting from inter-processor conflict (Invalidate, FWD_GETS, FWD_GETX) do not induce transitions at L2 (they are actually outputs actions). Only 4 out 148 transitions are induced by intra (UPG) or inter-processor conflict (GETX, GETS). Thus, most of the transitions at L2 are fostered by biasing.

3.4.4.2 Impact on error exposure and effort

Tables 7 and 8 convey three complementary pieces of information. Each table shows whether or not a generator was able to expose an error with a given test length when targeting 32-core designs (an entry filled in black indicates that the error was undetected). It also shows the required effort (expressed in seconds) to expose an error or the effort wasted in trying to expose it. Values in bold indicate the generator leading to the minimum effort for each verification scenario that exposed a given type of error. Finally, it shows the effectiveness of each generator to expose a given type of error in every scenario (as indicated between parentheses).

Let us first focus on random tests with 1K operations. In Table 8, note that no unbiased generator ever exposed errors for designs D4-D7 and D9, but biased generators exposed all errors except for D9. Table 7 shows that the probability of a random test to expose the error in design D2 is 1/50 for PLAIN+ and 1/5 for CHAIN+, i.e. PLAIN+ requires 50 tests on average to expose the error, while CHAIN+ requires only 5. This explains why CHAIN+ requires one order of magnitude less effort to expose the error in that design.

Let us now analyze the impact over the range of test lengths from 1K to 16K. For errors leading to SWMR violations (Table 7), neither pure chaining nor pure biasing significantly improved the effort. However, their combination led to the smallest effort for almost all designs and test lengths. Errors leading to DV violations (Table 8) were much harder to find. Pure chaining improved exposure for a single design (D8), whereas biasing significantly improved error detection for all designs. Their combination exposed the errors in every design for all cases but two. Although CHAIN+ found practically as many errors as PLAIN+, the former required the smallest effort in most cases.

Finally, we measured the time to *generate* tests for 32-core designs with 16K operations and 32 locations. On average, PLAIN- and PLAIN+ took 0.3 seconds, while CHAIN- and CHAIN+ took 0.8 seconds (i.e. one to four orders of magnitude smaller than the *verification* efforts in Tables 7 and 8).

Table 7 – Impact on effort (and error exposure) for 32-core designs containing SWMR violations (when exploiting 32 shared locations).

n	Unbiased								Biased							
	PLAIN-				CHAIN-				PLAIN+				CHAIN+			
	D0	**D1**	**D2**	**D3**	**D0**	**D1**	**D2**	**D3**	**D0**	**D1**	**D2**	**D3**	**D0**	**D1**	**D2**	**D3**
1K	177 (0.07)	702 (0.02)	701 (0.02)	237 (0.05)	232 (0.05)	685 (0.02)	685 (0.02)	349 (0.03)	92 (0.18)	25 (0.63)	966 (0.02)	986 (0.00)	**25** (0.82)	**25** (0.78)	**70** (0.20)	**229** (0.07)
2K	387 (0.03)	759 (0.02)	763 (0.00)	776 (0.02)	754 (0.02)	182 (0.07)	737 (0.02)	755 (0.00)	94 (0.22)	30 (0.82)	1324 (0.02)	1252 (0.00)	**29** (0.95)	**28** (0.92)	**67** (0.27)	**225** (0.08)
4K	81 (0.17)	135 (0.10)	163 (0.08)	138 (0.10)	133 (0.10)	197 (0.07)	264 (0.05)	269 (0.05)	91 (0.25)	35 (0.77)	1588 (0.00)	1610 (0.00)	**36** (0.92)	**35** (0.92)	**61** (0.43)	**191** (0.13)
8K	236 (0.07)	472 (0.03)	947 (0.02)	317 (0.05)	182 (0.08)	302 (0.05)	302 (0.05)	**308** (0.05)	86 (0.33)	42 (0.87)	2039 (0.02)	2072 (0.00)	**10** (1.00)	**42** (0.93)	**47** (0.57)	342 (0.10)
16K	129 (0.15)	166 (0.12)	224 (0.08)	287 (0.07)	90 (0.23)	129 (0.15)	168 (0.12)	**169** (0.12)	118 (0.33)	61 (0.77)	1463 (0.03)	3031 (0.00)	**55** (0.97)	**57** (0.97)	**66** (0.73)	285 (0.18)

Table 8 – Impact on effort (and error exposure) for 32-core designs containing DV violations (when exploiting 32 shared locations).

n	Unbiased												Biased											
	PLAIN-						CHAIN-						PLAIN+						CHAIN+					
	D4	**D5**	**D6**	**D7**	**D8**	**D9**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**
1K	658 (0.00)	645 (0.00)	720 (0.00)	710 (0.00)	645 (0.00)	658 (0.00)	642 (0.00)	628 (0.00)	703 (0.00)	693 (0.00)	312 (0.03)	641 (0.00)	56 (0.28)	**8** (1.00)	**202** (0.08)	**27** (0.53)	131 (0.12)	926 (0.00)	**23** (0.97)	**8** (1.00)	485 (0.03)	73 (0.23)	**23** (0.88)	876 (0.00)
2K	718 (0.00)	705 (0.00)	780 (0.00)	769 (0.00)	704 (0.00)	717 (0.00)	694 (0.00)	680 (0.00)	756 (0.00)	745 (0.00)	681 (0.00)	694 (0.00)	50 (0.42)	**8** (1.00)	**255** (0.08)	32 (0.73)	89 (0.23)	**290** (0.07)	**9** (1.00)	**8** (1.00)	1246 (0.00)	**32** (0.65)	**27** (0.92)	1119 (0.02)
4K	790 (0.00)	778 (0.00)	855 (0.00)	843 (0.00)	778 (0.00)	154 (0.08)	758 (0.00)	746 (0.00)	821 (0.00)	810 (0.00)	741 (0.02)	756 (0.02)	63 (0.45)	**8** (1.00)	210 (0.13)	**39** (0.85)	86 (0.28)	1541 (0.00)	**34** (0.97)	**8** (1.00)	**122** (0.20)	42 (0.83)	**33** (0.87)	**1510** (0.02)
8K	907 (0.00)	897 (0.00)	975 (0.00)	960 (0.00)	895 (0.00)	908 (0.00)	876 (0.00)	869 (0.00)	942 (0.00)	929 (0.00)	859 (0.02)	876 (0.00)	46 (0.53)	**8** (1.00)	525 (0.07)	49 (0.97)	214 (0.15)	2010 (0.00)	**9** (1.00)	**8** (1.00)	**368** (0.10)	**48** (0.97)	**42** (0.97)	**518** (0.07)
16K	1116 (0.00)	1106 (0.00)	1191 (0.00)	1172 (0.00)	1106 (0.00)	1109 (0.02)	1128 (0.00)	1116 (0.00)	1200 (0.00)	1182 (0.00)	212 (0.08)	1127 (0.02)	64 (0.63)	**8** (1.00)	1052 (0.05)	**12** (1.00)	162 (0.30)	**1451** (0.03)	**9** (1.00)	**8** (1.00)	**267** (0.20)	16 (1.00)	**53** (0.97)	3229 (0.02)

3.5 CONCLUSIONS

The experimental results indicate that the proposed techniques are complementary and improve constrained random test generation when a reusable memory-model checker is exploited for verification. Address biasing makes error detection less dependent on generation parameters, while operation chaining raises the probability of error detection. We have shown design cases for which the combination of biasing and chaining largely reduced the effort. In half of the 32-core designs, the baseline generator was unable to expose the same errors detected by our techniques with tests 16 times shorter. Albeit we had to limit the analysis to designs containing ten types of errors, the observed improvement in coverage indicates that the adequacy of the techniques is not limited to the evaluated design cases.

The techniques re-evaluated in this chapter under the new focus on coherence can be used to implement the Random Test Generation (RTG) engine for building a coverage-directed test generation, as depicted in the framework proposed in Figure 1. To do so, we realized that we had to find a proper way of dynamically exploiting biasing constraints.

During a few preliminary experiments for defining an adequate setup for the experiments reported in Section 3.4, we realized that the impact of *abc* and *sbc* on coverage and effort were much less important than the *cbc*. For this reason, we concluded that we should simplify biasing when used for directed test generation: for such purpose, we should keep both *abc* and *sbc* fixed.

More importantly, during those preliminary experiments, we also realized that if we enforced a uniform competition of locations for cache sets, the probability of inducing cache evictions was raised, thereby increasing coverage. For this reason, we concluded that we should simplify dynamic *cbc* exploitation by selecting a value of χ dependent on κ, instead of independently setting the two parameters of a $cbc = (\kappa, \chi)$. Therefore, given a selected number k of cache sets to which all locations must be mapped, we concluded that we should choose $cbc = (k, s/k)$ for uniform competition. That is how the parameters k and s are used to enforce competition biasing constraints. These two parameters, along with the parameter n, are sufficient for properly driving the RTG engine, as depicted in Figure 1.

Assuming these simplifications, the next chapter shows how chaining and biasing can be used (inside the RTG engine) for building a novel coverage-directed test generator.

4 MODEL-BASED DIRECTED TEST GENERATION

This chapter describes a first contribution on directed test generation that exploits the constrained random test generator described in Chapter 3. The technique described in this chapter is used to build a model-based directing engine (for the framework depicted in Figure 1), and it is also used inside the hybrid directing engine to be described in the next chapter. Since that contribution was previously reported in proceedings (ANDRADE et al., 2018), this chapter reproduces most of that text.

The features of our model-based Directed Test Generator (DTG) are:

1. The casting of *general* properties of coherence protocols and cache memories as non-conventional constraints on Random Test Generation (RTG).

2. A novel coverage model resulting from such constraints, which serves as a proxy for whatever coverage metric is adopted in the design environment.

3. A novel, steep coverage-ascent algorithm for the directing engine, which relies on the proposed coverage model.

This chapter is organized as follows. Section 4.1 shows the similarities and differences of the proposed approach as compared to related works. Section 4.2 presents the main ideas behind the contribution. Section 4.3 describes the proposed Directing Engine. Section 4.4.2 reports its experimental evaluation as compared to a state-of-the-art generator. Finally, Section 4.5 draws the chapter's conclusions.

4.1 RELATED WORK

This section employs the engines from Figure 1 (Section 1.3.2, p. 31) as a reference for discussing the similarities and differences of the proposed approach with respect to related works, as summarized in Table 9.

Qin and Mishra (QIN; MISHRA, 2012) proposed a simulation-based technique for protocol verification. Their technique does not require any RTG engine. Its directing engine builds a graph representation for the product Finite State Machine (FSM) that explicitly enumerates the entire protocol space. It exploits topological properties of that graph (clique or hypergraph) for inducing an efficient Euler tour of its edges. As a result, no transition is ever visited more than once, and tests leading to full transition coverage can be generated. Albeit the technique does not scale with growing core counts, the reported results indicate that the explicit enumeration of the full state space keeps viable for single-level protocols up to 16 cores.

MCjammer (WAGNER; BERTACCO, 2008) is a scalable scheme that avoids the enumeration of the full protocol space. Instead of a centralized directing engine, it relies on distributed intelligent agents that formulate their coverage goals according to Dichotomic Finite State Machines (DFSMs) capturing the protocol behavior from the perspective of each core

Table 9 – How related works address test generation and behavior verification.

Approach	Verification Scope	Generator			Checker Mechanism	Objective Function	Reusability restrictions	
		Directing engine	**RTG engine**	**Parameters**			**Generator**	**Checker**
Qin & Mishra (2012)	memory in isolation	Euler tour on hypergraph/clique	none	none	(unspecified)	Product FSM transition coverage	Protocol compliance	(unspecified)
Wagner & Bertacco (2008)	memory in isolation	multiple intelligent agents	purely random streams	variable: n, s	Data tagging	DFSM transition coverage	Protocol compliance	Partial MCM compliance
Fine & Ziv (2003), Adir et al. (2004)	full system	Bayesian network	constrained random under biasing constraints	variable: n, s	Golden Model	External coverage metric	none	Architecture compliance
Elver & Nagarajan (2016)	memory within full system	genetic programming	constrained random	fixed: n variable: s	MCM	External coverage metric	none	MCM compliance
This chapter's	memory within full system	steep coverage ascent	constrained random under coherence & biasing constraints	variable: n, s, k	MCM	External coverage metric	none	MCM compliance

domain. Given a processor domain, a state in the DFSM captures the state of a block in a 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 a same protocol, because the DFSM must be modified for porting the generator to a protocol variant.

Genesys-Pro is an approach to functional processor verification based on the solution of a constraint satisfaction problem (ADIR et al., 2004). It provides support for verifying the entire system, including the memory subsystem, and it relies on coverage-based RTG, where the directing engine can be, for instance, a Bayesian network (FINE; ZIV, 2003).

McVersi (ELVER; NAGARAJAN, 2016) proposes a Genetic Programming approach, where an RTG engine is used only for the creation of an initial population of tests. Further generations of tests are obtained from a pre-existing population by a directing engine that uses as objective function the fitness of a test, which is obtained from some coverage metric defined by the verification environment.[1] To obtain a new population from the fittest tests, the directing engine 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 the directing engine exploits non-determinism. As opposed to McVerSi, the proposed approach exploits general properties of coherence protocols and cache memories, and it balances their exploitation in separate engines. Like McVerSi's (ELVER; NAGARAJAN, 2016), our directing engine distinguishes the externally measured coverage from the inner mechanism for fostering coverage improvement, as opposed to MCjammer (WAGNER; BERTACCO, 2008), whose mechanism is tied to its inner coverage metric.

4.2 MAIN IDEAS BEHIND THE CONTRIBUTION

We constrain the RTG engine to produce tests leading to an effective coverage model for decision making in the directing engine.

The proposed coverage model relies on FSMs that specify the behavior of a coherence protocol for the cache controllers at each hierarchical level. It distinguishes three classes of transitions, assumes non-conventional constraints on the building of each test, and relies on three parameters that can drive the tests towards higher coverage, as explained in the following subsections.

4.2.1 Proposed classification of transitions

Our approach distinguishes three classes of transitions for the FSMs governing the private caches owned by a given processor:

[1] For instance, in Elver & Nagarajan (2016), the experimental evaluation employs a structural coverage metric of the logic implementing a FSM for the coherence protocol.

- *Class 1*: transitions induced by local events triggered by the processor or by another private cache controller lying on the same processor domain at the immediate higher hierarchical level. Such events result from intra-processor collisions.
- *Class 2*: transitions induced by local events triggered by requests from remote processors. Such events result from inter-processor collisions.
- *Class 3*: transitions induced by replacement events triggered by the cache controller itself.

Table 10 shows examples of events inducing transitions of such classes for a 3-level MESI protocol available in gem5's infrastructure (BINKERT et al., 2011).
[2]

Table 10 – Events inducing distinct classes of transitions

		Events inducing transitions from each class		
Level	**Cache**	**Class 1**	**Class 2**	**Class 3**
L0	private	Load, Store	Invalidate	Replacement
L1	private	L0_UPG, L0_GETS, L0_GETX	Invalidate, FWD_GETS, FWD_GETX	Replacement
L2	shared	UPG, GETS, GETX		Replacement

Since all processors share the last-level cache, it does not make sense to distinguish Classes 1 and 2 at that level. However, their distinction at the private cache levels is one of the keys to enabling an effective coverage model. The next section shows that, by properly *alternating* between such classes in successive accesses to the same location, transitions are less likely to be revisited, which favors coverage. Besides, it also shows that the controllability of replacement events is key to further improving coverage, because it allows the distinction between Class 3 and the other two classes.

4.2.2 Proposed constraints on RTG

Our approach exploits constraints on random test generation as mechanisms for enabling better control on coverage improvement. The first constraint enforces the alternation between Class 1 and Class 2 transitions, which is exploited by the RTG engine itself. The second one paves the way to the alternation between Class 3 and Class 1/2 transitions, which is exploited by the directing engine.

[2] The events in Table 10 refer to requests for read-only permission (Load, L0_GETS, FWD_GETS, GETS) and requests for read-write permission (Store, L0_UPG, L0_GETX, FWD_GETX, UPG, GETX), as well as invalidate and replacement events. Load and Store are requests from the home processor. L0_UPG, L0_GETS, and L0_GETX are requests from the private L0 cache to the private L1 cache of the same core domain. Fwd_GETS and FWD_GETX are requests from another core domain, which were forwarded by the directory (co-located at the shared L2 cache). Finally, UPG, GETS, and GETX are requests to the shared L2 cache.

4.2.2.1 Constraint 1: enforce alternation between Classes 1 and 2

To increase the chances of raising transition coverage, we enforce the RTG engine to build each test program according to rules that make successive colliding accesses likelier to induce transitions that are *different* from those already covered. Let us illustrate the idea by means of an example[3]. Suppose that processors i and j cooperate to execute a chain of memory operations, and there is no intervening replacement events in their private caches between the execution of the endpoints of such chain. Albeit a chain may contain operations referencing distinct locations, let us focus on the operations of a chain that collide at the same memory location (i.e. operations referencing other locations are abstracted out for simplicity). Figure 8 illustrates which transition of each private cache controller is covered after each operation in the chain is executed (it excludes replacement-induced transitions and assumes that, initially, no private cache contains a valid copy of a referenced block).

The chain used in the example was selected to enforce a pattern that alternates intra and inter-processor *conflicts*. The numbers serving as labels for elements in the chain and transitions in FSMs indicate that every operation tends to induce a *distinct* transition in *each* FSM. Note that class alternation happens after every second transition. Thus, inter-processor conflicts foster transitions of distinct classes, whereas intra-processor conflicts foster distinct transitions of a same class. This illustrates that the exploitation of conflicting events in intra/inter-processor *alternation* benefits coverage. Actually, the chain selected for the example is a particular case of more general canonical dependence chains (all exhibiting a similar pattern) (GHARACHORLOO, 1995). Our RTG engine was designed to build tests according to the rules of such canonical multiprocessor chains.

Albeit McVersi (ELVER; NAGARAJAN, 2016) also exploits conflicts, it does not distinguish between intra and inter-processor events, because its fitness function computes the union of the observed conflicts. On the other hand, our DTG exploits their distinction for coverage improvement. McVersi does not constrain the random generation of conflicting events (but only observes the fittest for learning how to improve further generation of tests), while our approach constrains them during RTG to avoid revisiting transitions.

4.2.2.2 Constraint 2: enforce uniform competition

To enable or disable Class 3 transitions, we constraint the choice of effective addresses in the RTG engine such that it can control the number of shared locations competing for a same cache set (i.e. the number of locations with same index). First, we define a generation parameter k, which represents the number of distinct cache sets for which the s shared locations compete. Then, for each value of s, we constraint the values that can be assigned to k: only those for which s is multiple of k are kept in the generation space. As a result, the RTG engine is constrained to assign effective addresses in such a way that exactly s/k locations compete for

[3] Although MESI is used as an example, the rules of formation of such a chain are protocol independent.

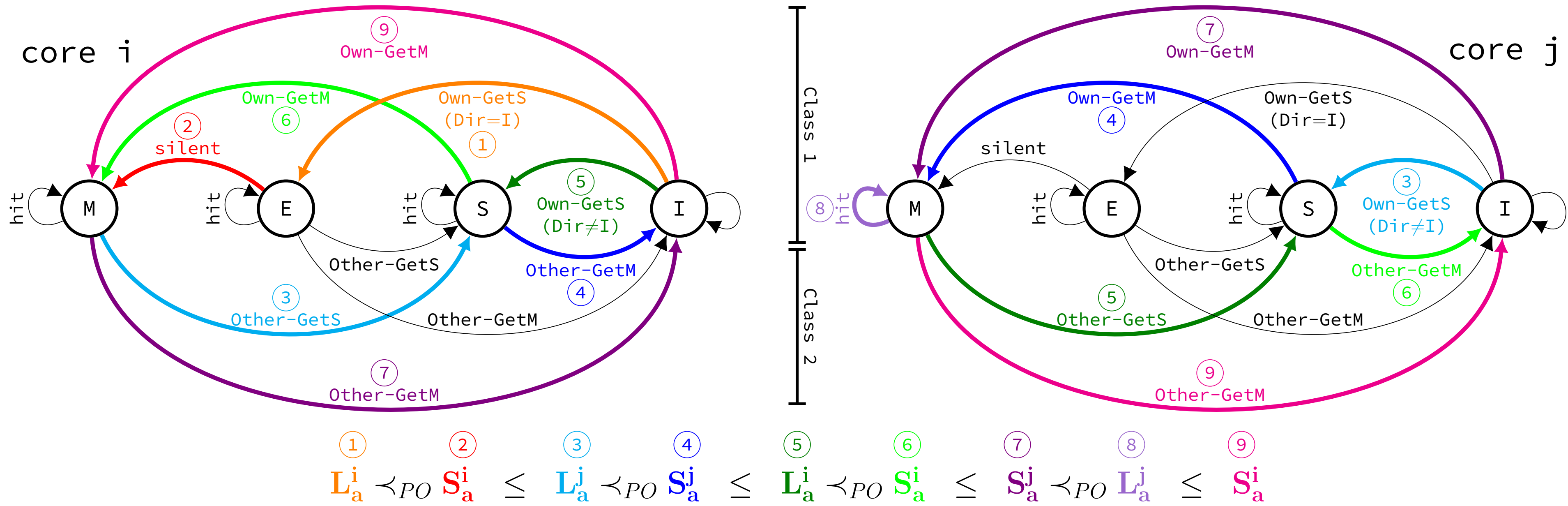


Figure 8 – Example of how colliding operations should be chained to avoid revisiting already covered transitions.

each cache set. Such *uniform* distribution maximizes the probability of inducing replacements in all sets for a given setting of a pair (s,k).

Let α denote the associativity of a cache. For inducing a replacement event in a given cache set, a sequence of at least $\alpha+1$ references to *distinct* locations competing for that set is required. Therefore, a necessary condition for enabling replacement is $s/k \geq \alpha+1$. Conversely, a sufficient condition for disabling replacement in all sets is $s/k < \alpha+1 \Leftrightarrow s/k \leq \alpha$. Thus, there is a threshold s/α for the value of k above which replacement is certainly disabled, but below which it may be enabled depending on the sequence of references that turns out to be generated randomly. Therefore, the RTG engine can stimulate the alternation between Class 3 and Class 1/2 transitions by selecting appropriate values of k.

The proposed use of a generation parameter to deliberately enable or disable replacement events is uncommon in the literature on directed test generation for shared memory verification. It plays a similar role as the biasing constraints employed in the approach that casts verification as a constraint satisfaction problem (ADIR et al., 2004).

4.2.3 Proposed coverage model

We propose a coverage model that pessimistically assumes that Classes 1, 2, and 3 induce a partition of the set of transitions of a FSM. Let $TC = (tran_1 + tran_2 + tran_3)/total = TC_1 + TC_2 + TC_3$ denote the transition coverage of the FSM specifying the protocol behavior for the memory block containing a given location, where $tran_j$ denotes the number of distinct transitions from Class j and $total$ denotes the overall number of transitions in the FSM.

Every collision leads to a transition from Classes 1 or 2, i.e. $TC_{1/2} = TC_1 + TC_2$ denotes the coverage of collision-induced transitions. Let $N_{COL} = n/s$ be the average number of collisions induced by a test. Albeit Constraint 1 increases the probability that successive colliding accesses lead to distinct transitions in the FSM, this might not always be true. Thus, we assume that the number of covered transitions is *proportional* to the average number of collisions as an estimate for the actual coverage, written $\widehat{TC}_{1/2} \propto n/s$.

Let $N_{REP}(set(a))$ denote the average number of replacement events for the cache set assigned to the memory block where location a resides. An increase in $N_{REPL}(set(a))$ raises the probability of transitions from Class 3. For an estimate of Class 3 transition coverage, we have to count the number of replacement events, which depends on the associativity and on the memory access pattern. The ratio n/k measures how many operations are mapped to the same cache set on average. A best-case access pattern for replacement events is such that every element of a sequence of n/k accesses makes reference to distinct locations mapped to the same set. Replacement takes place at every $\alpha+1$ such accesses. Therefore, an upper bound for the average number of replacements is $N_{REP}^{max} = (n/k)/(\alpha+1)$. A worst-case pattern for replacement events is such that every element of a subsequence of n/s accesses makes reference to the same location, then another subsequence of n/s accesses makes reference to another location, and so on. There are s such subsequences, s/k of them map to the same set (on average), and

replacement takes place at every $\alpha+1$ transitions between them. Therefore, a lower bound for the average number of replacements is $N_{REP}^{min} = (s/k)/(\alpha+1)$. As a result, we can make the hypothesis that the number of covered transitions is, in the best case, proportional to the average number of operations per set, and in the worst case, it is proportional to the average number of locations mapped to a same cache set. Such hypothesis leads to a suitable estimate for the actual Class 3 transition coverage, i.e. $\widehat{TC}_3 \propto n/k$, in the best case, and $\widehat{TC}_3 \propto s/k$, in the worst case.

Thus, $\widehat{TC}(n,s,k) = \widehat{TC}_{1/2}(n,s) + \widehat{TC}_3(n,s,k)$ can be used by the directing engine for making decisions that favor transition coverage. Besides, since we assume that the classes induce a partition of all transitions in the FSM, the algorithm in the directing engine should lower the probability of transitions from one class in an attempt to raise the probability of transitions from another class and vice-versa. The cumulative effect of such exploration of complementary scenarios should contribute to raising the overall coverage.

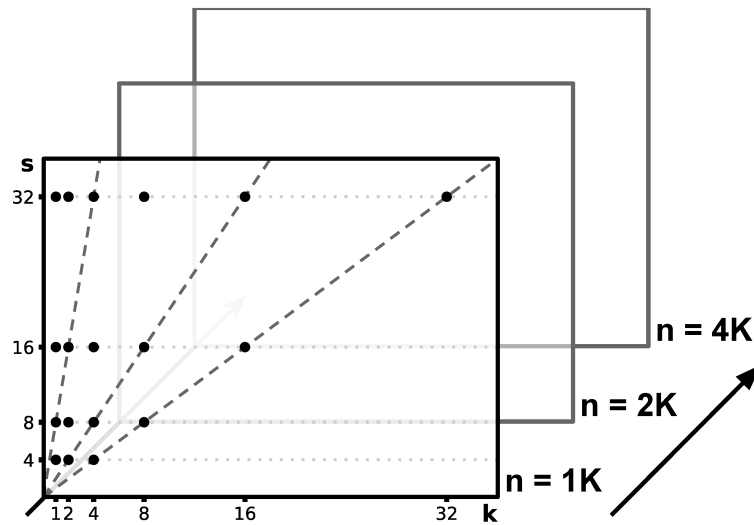
The next section first describes, by means of an example, how the directing engine exploits parameters according to the proposed coverage model, and then it formalizes the underlying algorithm.

4.3 DESCRIPTION OF THE DIRECTING ENGINE

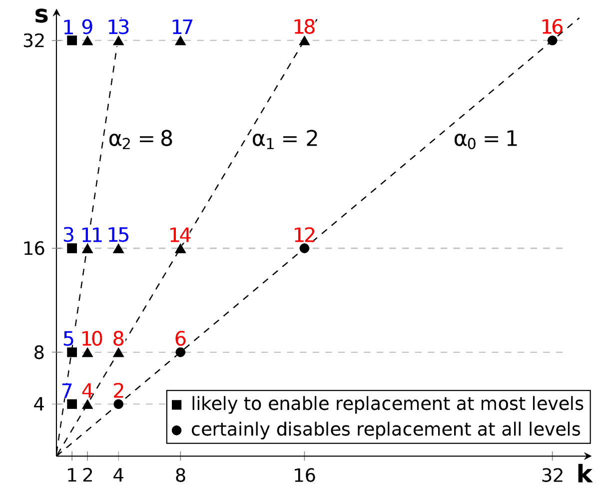
4.3.1 An example of how it works

The values assigned to the parameters (n,s,k) within user-defined ranges induce a tridimensional generation space. Given such generation space, our directing engine visits planes that correspond to increasing test sizes (n). Figure 9a depicts a generation space with planes for 1K, 2K, and 4K operations.

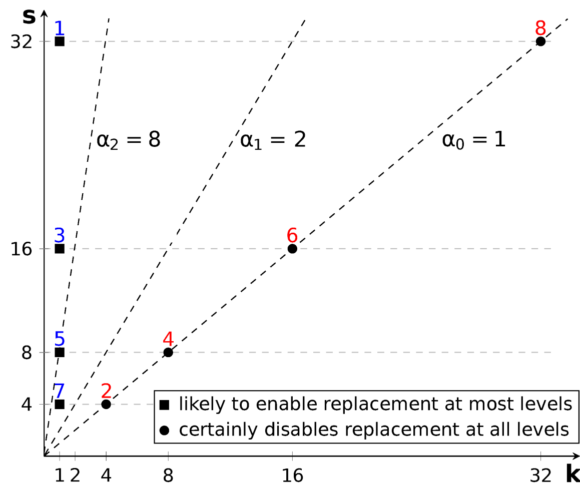
Figure 9b illustrates one such plane for a range of shared locations defined as $S = \{4,8,16,32\}$. Let us suppose a memory hierarchy with three cache levels, each with a distinct associativity: L0 is directed mapped, L1 is 2-way, and L2 is 8-way. A point with a mark (square, triangle or circle) represents a pair (s,k) that leads to a uniform distribution of location competition for cache sets. Unmarked points were excluded from the generation space by Constraint 2. Dashed lines correspond to the distinct associativities at each cache level. Each dashed line groups the points of the generation space that represent the threshold for disabling replacement events for different values of s. For a given number of locations s, a mark in a dashed line labeled as α_j represents the minimum value of k required for disabling replacement events in a α_j-way cache lying at level j. Therefore, the marks to the left of a dashed line denote the values of k that are likely to stimulate replacement-induced transitions at level j (Class 3), while all the marks to the right (or on the line itself) denote values of k that certainly do not induce replacement events in any set of a cache at level j, being therefore likely to stimulate collision-induced transitions instead (Class 1/2). For this reason, when trying to stimulate collision-induced transitions, the directing engine could explore only the pairs marked with circles, because the value $k = s$ is sufficiently large for disabling replacement-induced transitions at *all* levels for a given



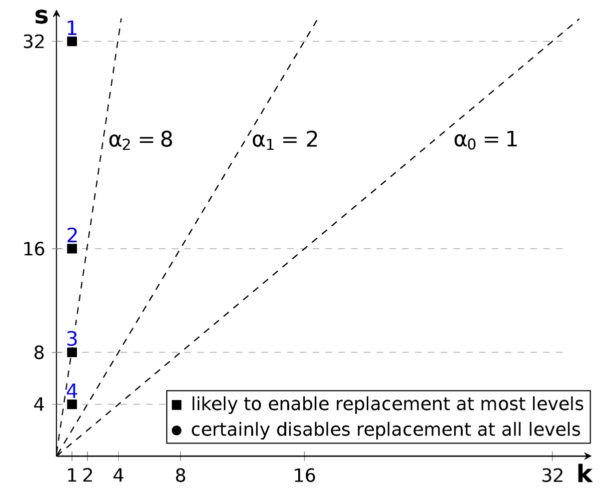
(a) Planes of the generation space



(b) Search space for Variant 1



(c) Search space for Variant 2



(d) Search space for Variant 3

Figure 9 – How the distinct variants of the proposed DTG traverse a plane of the generation space.

s. On the other hand, when trying to stimulate replacement-induced transitions, our engine could explore only the pairs marked with squares, because the value $k = 1$ corresponds to the maximum probability of replacement for a given s. Note, however, that albeit the pairs marked with squares are likely to enable replacement at most levels, this may not necessarily hold for all (for instance, $(4,1)$ and $(8,1)$ may enable replacement at L0 and L1, but not at L2).

Therefore, to control the stimulation of a desired type of transition, the directing engine does not necessarily have to explore all the marked points in Figure 9b, but only search the subspace marked with squares and circles. Figure 9c illustrates such a reduced search space. Note that, any traversal alternating between square and circle is likely to stimulate a sequence of transitions induced by an alternation between replacement and collision events. To select the most convenient traversal, the directing engine relies on the proposed coverage model. Since $\widehat{TC}_3 \propto s/k$ or $\widehat{TC}_3 \propto n/k$, when exploring a plane for a given n, our engine selects the pair in the search space with minimum k and maximum s. Since $\widehat{TC}_{1/2} \propto n/s$, our engine selects the pair in the search space with minimum s. Albeit in such case k could be arbitrary selected according to the coverage model, the engine chooses the maximal k, because it has the advantage of disabling replacement at all levels. Such choices lead to the traversal indicated by the labeling in Figure 9c. Note that, in such a traversal, a move from a square to a circle, corresponds to the alternation between the maximum probability of replacement and the maximum probability of collision for a given *unexplored sub*-space. Thus, such *steep coverage-ascent* traversal was designed to reach the highest coverage as possible in the smallest time (thereby reducing the effort to find errors), while still exploring the tridimensional search space by successively visiting planes induced by increasing values of n (for reaching the highest coverage as possible within a pre-specified range). The labeling in Figure 9b illustrates such a traversal for the original search space. Finally, Figure 9d illustrates a degeneration of it for an even smaller search space, which fosters Class 3 transitions predominantly. We propose the three variants illustrated in Figure 9 for the directing engine.

4.3.2 The proposed algorithm

Let the range of a parameter be the set of all values that can be assigned to it. Let N, S, and K be sets representing the ranges for the parameters n, s, and k, respectively. Let v denote a directing engine variant. N, S, and v are user-defined, whereas K is bounded by the maximum number of locations, i.e. $K = \{k : 1 \leq k \leq \max S\}$. Let $G = N \times S \times K$ be the generation space and let $P = S \times K$ be a plane corresponding to a given n. The directing engine explores planes in order of increasing values of n (for higher test throughput).

Let $CV(n,s,k)$ be a function that maps a setting of parameters to the *cumulative* coverage value obtained by executing all tests generated so far, i.e. it represents the directing engine's input-output interaction with the RTG engine (which actually consumes the parameters) and the coverage analyzer (which actually accumulates the coverage resulting from a suite of tests). Let cv be a coverage value, x be a Boolean value, P be the points of each plane that belong to the

search space, and P^* be the unexplored points of P.

Let $\Pi(s,v)$ denote the adopted sub-range of k in the search space for a given number of locations s and a given variant v, as follows:

$$\Pi(s,v) = \begin{cases} \{k : (1 \leq k \leq s) \wedge (s \mod k = 0)\} & \text{if } v = 1 \\ \{1, s\} & \text{if } v = 2 \\ \{1\} & \text{if } v = 3 \end{cases}$$

Algorithm 1 first defines the search space P for the desired variant v (line 2). Then it explores successive planes for growing test lengths (lines 4-21). It searches each plane in the order depicted in Figure 9 (lines 7-19). Generation is stopped when all planes in the range of n are explored or full coverage is reached.

```
 1: procedure DIRECTING-ENGINE(N, S, v)
 2:     P ← {(s,k) : s ∈ S ∧ k ∈ Π(s,v)}
 3:     x ← 0
 4:     repeat
 5:         n ← min N
 6:         P* ← P
 7:         repeat
 8:             if x = 0 then
 9:                 k* ← min{k: (s,k) ∈ P*}
10:                 s* ← max{s: (s,k) ∈ P* ∧ k = k*}
11:                 if v ≠ 3 then
12:                     x ← 1
13:             else
14:                 s* ← min{s: (s,k) ∈ P* ∧ k ≠ 1}
15:                 k* ← max{k: (s,k) ∈ P* ∧ s = s*}
16:                 x ← 0
17:             P* ← P* \ {(s*,k*)}
18:             cv ← CV(n, s*, k*)
19:         until (P* = ∅) ∨ (cv = 1)
20:         N ← N \ {n}
21:     until (N = ∅) ∨ (cv = 1)
```

Algorithm 1 – The algorithm underlying the directing engine.

Albeit the traversal order is essentially determined by the proposed coverage model, some ranking is used to further induce alternation between replacement and collision as much as possible. Lines 9-10 rank the choice of k before s to stimulate replacement at most levels, whereas lines 14-15 rank the choice of k after s to avoid replacement at most levels, thereby stimulating collisions at most of them.

It should be noted that the apparent simplicity of the directing engine results from proper task encapsulation within the two cooperating DTG engines. First, the complexity of handling Constraints 1 and 2 lies inside the RTG engine. Second, the exploitation of Constraint 2 resulted in preliminary pruning. Third, the applied constraints resulted in an effective coverage model that simplified the traversal of the search space. It is such deliberate encapsulation

that enables the DTG to reach a given coverage in less time than a state-of-the-art generator, as reported in the next section.

4.4 EXPERIMENTAL EVALUATION

4.4.1 Experimental setup

We compared the proposed DTG with the McVerSi (ELVER; NAGARAJAN, 2016) Test Generator (MTG), which is available in the public domain (ELVER, 2016). We extracted only the generator's code, and adapted its interface to our framework's. Except for the directing engine (DTG or MTG), and the random test generator (as McVerSi's code already included a built-in generator), all the other engines employed in the experiments were exactly the same. We preserved all genetic parameters exactly as they were originally set in (ELVER; NAGARAJAN, 2016). We enforced uniform operation bias (equal load and store probabilities) and block-aligned addresses in all experiments.

We relied on the pseudo-code described in (FREITAS; RAMBO; SANTOS, 2013) to implement the axiomatic checker. We relied on gem5's infrastructure (BINKERT et al., 2011) for simulation and design representation (*O3* out-of-order CPU model, *Ruby* memory-system model, and *simple* interconnect-network model). We adopted gem5's 3-level MESI directory protocol with 4KB (directed-mapped) private caches at L0, 64KB (2-way) private caches at L1, and a 2MB (8-way) shared cache, all operating with same block size (64 bytes).

Without loss of generality, but for experimental convenience, we adopted within our coverage analyzer a structural metric similar to the one employed in McVerSi (ELVER; NAGARAJAN, 2016). This choice allows us to stress the different coverage roles in our DTG: it is directed by an external *structural* coverage metric, but it drives generation according to an inner *functional* coverage model. The structural coverage was measured as follows. While running a test, we tracked the number of distinct transitions covered in the code of FSMs. No distinction was made between identical controller instances (at a same hierarchical level). Thus, the coverage is the fraction of all transitions that were tracked. The MTG receives the coverage of each test, while the DTG receives the *cumulative* coverage of the sequence of tests.

We adopted an uniform operation bias for all generators (50% of probability for loads and 50% for stores). In all generators, addresses were constrained to be aligned to the block (i.e. base addresses are multiple of a 64-byte stride).

4.4.2 Experimental results

DTG and MTG differ in which parameters are *statically* defined by the user and which are *dynamically* set by their engines. The MTG requires the static definition of a *fixed* test length and a *single* address-space constraint, while the DTG dynamically exploits *variable* test lengths and *multiple* address-space constraints (as a result of varying the parameter *k*). Since

the *dynamic* exploitation of *multiple* address-space constraints is one of the distinguishing features of the proposed generator, it would be pointless to enforce a *single*, *static* address-space constraint when running the DTG for a comparison with the MTG. Instead, we compare our results under the dynamic exploitation of multiple address-space constraints with McVerSi's static exploitation of a single address-space constraint. However, we also report results for the DTG under fixed test length and for the MTG under *distinct* static constraints: the test memory sizes $TM = 1KB$ and $TM = 8KB$ defined in (ELVER; NAGARAJAN, 2016) (where the useful address space is spread over non-contiguous chunks of 512B, each starting at a distance of 1MB).

McVersi is free to select as many locations as available within its useful address space. In the proposed DTG, however, the number of locations must belong to a pre-specified range. To accommodate such difference, we compared the generators under the same upper bound on the number of (useful) memory locations. Different upper bounds correspond to each TM size adopted for McVerSi. Under Constraint 2, the maximum number of block-aligned locations is 16 for TM=1KB and 128 for TM=8KB. As a result, we adopted $S = \{4, 8, 16\}$ and $S = \{4, 8, 16, 32, 64, 128\}$ for the DTG when comparing with the MTG under $TM = 1KB$ and $TM = 8KB$, respectively.

We first measured the cumulative coverage resulting from the execution of a sequence of tests on designs containing *no errors*. Then we inserted different artificial errors to challenge the generators by changing the FSMs (either by modifying the next state or precluding some due output action). Each faulty design under verification contained a single, distinct error. The errors studied in our experiments are described in Table 11. To determine the effort spent in an attempt to find a given error, we measured the runtime until it was found or until the directing engine stopped generation. Each generated test was executed five times under different

Table 11 – Studied design errors

ID	**State**	**Input event**	**Next state**	**Precluded output action**
E.4.1 (L0)	E	Store	M	dirty=1 in store_hit
E.4.2 (L1)	M_ILO	L0_DataAck	EE instead of MM	(preserved)
E.4.3 (L1)	E	WriteBack	MM	writeDataFromL0Request
E.4.4 (L1)	IS_I	Data_all_Acks	I	writeDataFromL2Response
E.4.5 (L1)	E_ILO	L0_DataAck	MM	writeDataFromL0Response
E.4.6 (L1)	IS_I	DataS_fromL1	I	writeDataFromL2Response
E.4.7 (L1)	E_IL0	WriteBack	MM_IL0	writeDataFromL0Request
E.4.8 (L1)	IS SM SM	Inv Data_all_Acks Data	IS instead of IS_I M SM	(preserved) (preserved as in (IM, M)) (preserved as in (IM, SM))
E.4.9 (L1)	S	L0_Invalidate_Own	SS instead of S_IL0	forward_eviction_to_L0

simulation states (not related to the test itself) in such a way that the distinct executions of the same test are all perturbed differently (ELVER; NAGARAJAN, 2016). To obtain the reported values of coverage and effort, we launched each generator ten times by exploiting different seeds, and we took the median values of the resulting distributions. Runtimes were measured in an HP xw8600 workstation (Intel Xeon E5430, 2.66 GHz, 8GB memory).

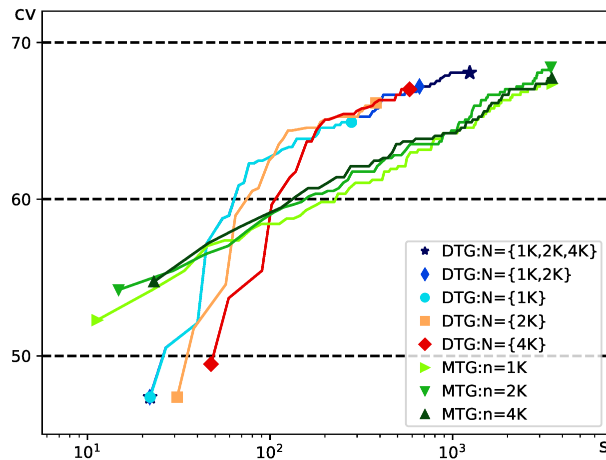
We let each generator run until it stops or a time limit of one hour is elapsed, and we measured the runtime required for reaching increasing cumulative coverage values. Figure 10 compares DTG's Variant 1 with the MTG. It plots the time to reach increasing coverage values when DTG and MTG exploit fixed sizes $(1K, 2K, 4K)$ in different test suites, and the DTG exploits them in a single suite within some combined range (either $\{1K, 2K\}$ or $\{1K, 2K, 4K\}$).

Notice that, for a given test size, the MTG requires less time than the DTG to reach coverage values below a certain threshold. This can be explained by the MTG's exploitation of variable thread length, as opposed to the current implementation of our RTG engine, which constrains all threads to have the same length. However, such MTG feature does not seem to pay off indefinitely. Indeed, above that threshold, the MTG takes much longer than the DTG. This indicates that the proposed enforcement of alternated conflicts (Constraint 1) is more efficient for guiding generation towards higher coverage than the MTG's selective crossover, i.e. the MTG takes longer to learn how to select the most promising conflicting events than our approach takes to enforce them by construction.

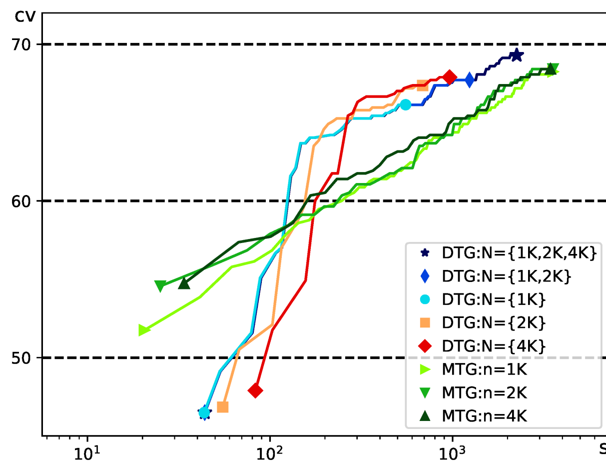
Observe that, typically, the MTG reached a given coverage faster with $n = 4K$, which led to its highest coverage within the one-hour time limit for a 32-core design (67.5%). In contrast, below 65%, the DTG reached a given coverage *slower* with $N = \{4K\}$. This means that, up to that value, the variation of parameters (s, k) is sufficient to cover easy-to-stimulate transitions whatever the test size. However, harder-to-stimulate transitions above 65% coverage are more sensitive to test size. That is why the highest values obtained by the DTG with $N = \{1K\}, \{2K\}$, and $\{4K\}$ are different (66.8%, 67.7%, and 68.9%, respectively, for 32-core designs).

Note that the MTG sometimes reached higher coverage than the DTG for a *fixed* test size, as far as it is let run for a long time. This apparent limitation of the DTG in trading runtime for higher coverage is just a consequence of the artificial restriction of imposing a fixed size to an algorithm designed to exploit a range of sizes. Such restriction makes it stop prematurely, ruling out one of its distinguishing features: the ability to exploit multiple, increasing test sizes (in the same test suite) for reaching higher coverage faster.

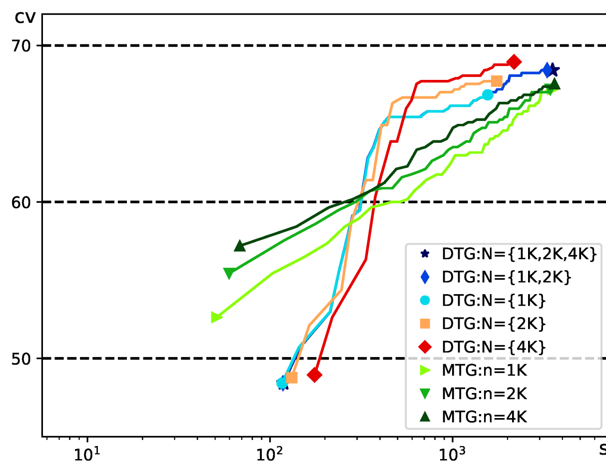
Indeed, when the range {1K, 2K, 4K} was exploited, the DTG attained an even higher coverage (e.g. 69.3% for a 16-core design) than obtained with fixed sizes (for a 32-core design, such effect is also observed, but beyond the one-hour time limit adopted in Figure 10). This can be explained as follows. For a given core count, the shorter the test, the shorter the threads, and the shorter the multiprocessor chains (i.e. the fewer the number of conflicts). As a result, the shorter the test, the lower the probability of conflicts. This means that, in our approach, the variability in test size works as a mechanism for modulating the probability of conflicts, thereby



(a) 8-core design



(b) 16-core design



(c) 32-core design

Figure 10 – Coverage evolution for fixed and variable test sizes (under an upper bound of 128 locations and one-hour time limit).

inducing distinct transitions with different test sizes, which benefits coverage. Such advantage of variable over fixed test size shows that the proposed DTG indeed allows the user to trade runtime for coverage (or vice-versa) by extending (or shrinking) the range of sizes. Instead, an MTG user should be able to either guess the proper test size or, otherwise, launch multiple fixed-size test sequences (at the expense of extra effort).

In short, for 8-core designs, the MTG reached the highest coverage (68.4%), but it was 2.5 times slower to attain the highest DTG coverage (68.1%). For 32-core designs, the DTG reached the highest coverage (68.9%), and it was from 1.9 to 5.7 times faster to attain the highest MTG coverage (67.5%).

Table 12 reports the effort required by each generator for exposing errors in different designs under distinct upper bounds on the number of locations. It shows the best values in bold. An entry filled in black indicates that the error was never found despite the wasted effort. The left-hand partition of Table 12 shows that, albeit the DTG exposes every error in all cases, the MTG is unable to expose E.4.1 and E.4.2 in most designs. This is due to the inappropriate choice of address-space constraint ($TM = 1KB$), which restrains replacement-induced transitions. The right-hand partition shows that, with a more adequate constraint ($TM = 8KB$) for the MTG and, equivalently, with a large enough range of locations for the DTG, every error is exposed in all cases by each generator. However, the contrast between the partitions show how the proposed dynamic exploitation of multiple address-space constraints can prevent the user from inadvertently limiting error exposure.

The right-hand partition of Table 12 shows that $n = 2K$ led the MTG to the best effort in most 32-core designs, as compared with $n = 1K$, and it indicates that Variant 3 could be used for exposing errors in early phases of the verification process, because it was the DTG variant that exposed all errors with the best effort (except for E.4.4). It shows that, when Variant 3 is compared with the MTG for $n = 2K$, both led to similar effort for E.4.1, E.4.2 and E.4.5; otherwise, Variant 3 was up to 2.8 times faster, except for a few E.4.4 and E.4.6 designs, where it was down to 1.8 times slower than the MTG (for $n = 2K$). However, for two E.4.4 designs, Variant 1 was up to 4 times faster. This can be explained as follows. When an error is exposed with the same effort by all DTG variants (e.g. E.4.1, E.4.2, E.4.3, E.4.5), this means that the transitions required to expose it happen to be induced by points of the search space with $k = 1$ (i.e. replacement-induced transitions), and they were covered with few tests. However, when Variant 3 leads to the smallest observed effort among all DTG variants (e.g. E.4.6), this means that, albeit the transitions required to expose that error are also predominantly replacement-induced transitions, a larger number of tests is required to cover them. Finally, when Variant 3 leads to the largest effort among all DTG variants (e.g. E.4.4), collision-induced transitions are essential to foster error exposure.

Table 13 reports the effort for the designs containing the most challenging studied errors (whose detection depends on rarely-covered transitions). To handle the difficulty of exposing them, we extended the range of test sizes for both DTG and MTG. We also reported (between parentheses) how many test suites effectively exposed an error out of the ten suites

Table 12 – Effort required for finding errors under distinct upper bounds on the number of locations.

	16 locations ($TM = 1KB$)															128 locations ($TM = 8KB$)														
	DTG with N = {1K, 2K}									MTG						DTG with N = {1K, 2K}									MTG					
	Variant 1			Variant 2			Variant 3			n = 1K			n = 2K			Variant 1			Variant 2			Variant 3			n = 1K			n = 2K		
Error	8	16	32	8	16	32	8	16	32	8	16	32	8	16	32	8	16	32	8	16	32	8	16	32	8	16	32	8	16	32
E.4.1	**4**	**7**	335	**4**	**7**	178	**4**	**7**	**100**	3600	3600	3600	3600	3600	3600	**5**	**7**	**14**	**5**	**7**	**14**	**5**	**7**	**14**	**5**	18	23	**5**	8	36
E.4.2	**4**	**6**	**12**	**4**	**6**	**12**	**4**	**6**	**12**	3600	3600	3600	3600	3600	3600	4	7	13	4	7	13	4	7	13	**3**	**6**	**12**	4	**6**	13
E.4.3	**4**	**7**	**25**	**4**	**7**	**25**	**4**	**7**	**25**	260	558	1839	264	677	1284	**5**	**8**	**16**	**5**	**8**	**16**	**5**	**8**	**16**	8	11	132	**5**	16	32
E.4.4	**14**	21	22	**14**	21	22	**14**	21	22	25	18	**13**	36	**7**	15	**26**	**44**	112	**26**	**44**	112	59	97	187	113	59	128	106	72	**104**
E.4.5	20	73	101	20	72	101	**13**	**42**	**62**	34	174	610	55	207	1091	5	**7**	**14**	5	**7**	**14**	5	**7**	**14**	**4**	13	23	**4**	8	15
E.4.6	19	31	72	19	31	72	13	**21**	**47**	**12**	138	459	**12**	92	182	82	142	352	82	142	356	57	**102**	**249**	**37**	190	1039	42	250	705

Table 13 – Effort required for finding errors under an upper bound of 128 locations.

	DTG with N = {1K, 2K,4K}						MTG								
	Variant 1			Variant 3			n = 1K			n = 2K			n = 4K		
Error	8	16	32	8	16	32	8	16	32	8	16	32	8	16	32
E.4.7	**81** (10)	403 (10)	555 (10)	**81** (10)	**136** (10)	**369** (10)	443 (10)	1483 (9)	3600 (2)	399 (10)	842 (10)	3600 (3)	482 (10)	671 (10)	1280 (6)
E.4.8	**89** (10)	**47** (10)	**121** (10)	224 (6)	189 (10)	587 (10)	2541 (10)	506 (8)	484 (10)	2758 (9)	1441 (10)	294 (10)	2096 (6)	1492 (10)	257 (10)
E.4.9	**253** (10)	**740** (10)	**1841** (9)	306 (0)	505 (0)	1102 (0)	3397 (5)	3600 (1)	3600 (1)	3600 (3)	3600 (1)	3600 (1)	3600 (1)	3600 (1)	3600 (0)

generated with different seeds. Notice that the MTG often led to the maximum number of effective test suites for $n = 4K$ and that Variant 1 was always the most effective DTG variant. Besides, the test suites generated by DTG's Variant 1 and by the MTG with $n = 4K$ were similarly effective in exposing errors (except for E.4.9), but the DTG was always faster (from 1.6 to 31 times). The DTG was more effective and efficient in exposing E.4.9 in 8, 16, and 32-core designs.

Thus, the experiments indicate that, for a given range of test sizes, when the DTG adopts Variant 1 and the MTG operates at the extreme of that range to face hard-to-find errors in late phases of the verification process, the proposed DTG tends to reach similar or superior effectiveness in error exposure as the MTG, often requires significantly less effort, and reaches similar coverage much faster.

4.5 CONCLUSIONS

Racy tests tend to expose shared-memory bugs faster (MANOVIT; HANGAL, 2006; SHACHAM et al., 2008). Deterministic tests are less likely to expose errors, because too few of their execution witnesses are invalid (ELVER; NAGARAJAN, 2016). Since the proposed generator reached similar coverage much faster than the most recently reported generator of racy tests (ELVER; NAGARAJAN, 2016), this chapter reveals a new facet of test generation beyond the usual exploitation of non-determinism: when a generator enforces multiprocessor dependence chains for alternating between intra-processor and inter-processor conflicts, it typically requires shorter and fewer racy tests for reaching the same coverage, especially if the alternation between replacements and collisions is properly controlled.

5 HYBRID DIRECTED TEST GENERATION

This chapter describes a second contribution on directed test generation. Since it was previously reported in a journal article (ANDRADE et al., 2020), this chapter reproduces most of that text. It leads to a generalization of the model-based Directing Engine proposed in Chapter 4 (for the framework depicted in Figure 1), and it also relies on the RTG Engine described in Chapter 3. The generalization involves both data-driven and model-based techniques, and gives rise to a *Hybrid Test Generator* (HTG), whose main features are:

1. The formulation of Directed Test Generation (DTG) as a double-objective optimization problem deliberatety defined in the Random Test Generation (RTG) space.

2. A novel data-driven technique that explores neighborhoods of the RTG space (instead of enumerating the protocol state space as in Qin & Mishra (2012), Wagner & Bertacco (2008), Lyu et al. (2019)), and avoids excluding optimal solutions from the search space.

3. A generalization of the model-based technique proposed in Chapter 4 such that constraints can be exploited for fast coverage evolution without hampering neighborhood exploration.

The first feature allows a pragmatic way of building upon the legacy of constrained RTG, but under a novel *hybrid approach* enabled by the other features. Instead of a plain composition of well-known techniques, the approach was designed for proper decoupling of competences: *data-driven exploration* and *model-based exploitation*, as illustrated in Figure 11.

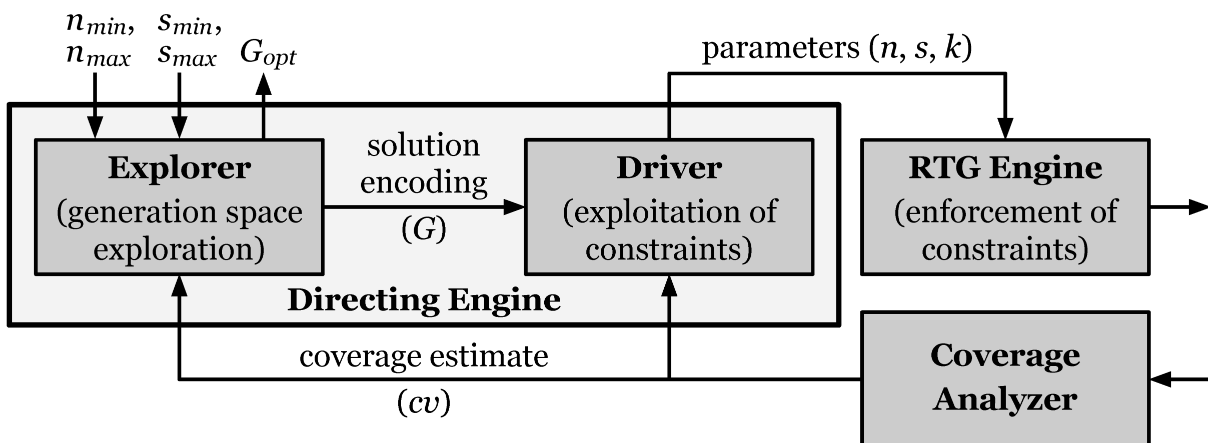


Figure 11 – The anatomy of the proposed Directing Engine.

The Directing Engine consists of a generation space *Explorer* and a test generation *Driver*. The Explorer assumes bounds on test sizes (n_{min}, n_{max}) and on the amount of shared locations (s_{min}, s_{max}). It defines an encoding (G) consisting of multiple settings for the parameters (n, s, k) for inducing a test suite. The Driver serializes them to induce an order of random tests leading to fast coverage evolution. The Explorer defines new encodings as far as coverage is

not satisfactory, and returns the one leading to the best solution (G_{opt}). The synergy between its sub-engines is demonstrated by experimental results.[1]

The remainder of this chapter is organized as follows. Section 5.1 shows the similarities and differences of the proposed approach as compared to related works. Section 5.2 formulates the target problem as an optimization problem. Sections 5.3 illustrates how the proposed approach works. Section 5.4 describes the proposed generalization. Section 5.5 reports its experimental evaluation. Finally, Section 5.6 draws the chapter's conclusions.

5.1 RELATED WORK

Table 14 splits DTG approaches into two main classes. *Model-based generators* rely on some abstraction (e.g. FSM or graph) that either explicitly encodes what has to be covered (e.g. the states or transitions of a coherence protocol (QIN; MISHRA, 2012)) or implicitly captures how coverage can be increased (ANDRADE et al., 2018). *Data-driven generators* rely on feedback loops to adaptively change generation parameters according to coverage evolution.

Table 14 – DTG approaches for functional verification of multicore chips.

Approach	**Scope**	**Directing Engine**
Qin & Mishra (2012)	coherence protocol	Euler tour
Lyu et al. (2019)	coherence protocol	Euler tour, quotient space
Andrade et al. (2018)	shared memory	coverage model
Fine et al. (2006)	full system	Bayesian network
Wagner & Bertacco (2008)	coherence protocol	multiple intelligent agents
Elver & Nagarajan (2016)	shared memory	genetic programming

One approach focuses on coherence protocol verification (QIN; MISHRA, 2012). It builds sequences of instructions that induce an Euler tour on a graph representing the state space. Since it relies on the product of FSMs at each core domain, such approach was not scalable. It deserved a recent extension (LYU et al., 2019), based on a symmetry reduction technique, which defines equivalent classes and restricts the state space to class representatives, allowing a trade-off between coverage and verification budget. The approach assumes that transitions between stable and transient states are correct. Such assumption is not suitable for memory consistency verification, because it requires artificial order constraints[2] for proper controllability, thereby inhibiting data races, which are well-known mechanisms for exposing shared-memory design errors (SHACHAM et al., 2008; HANGAL et al., 2004). Besides, the approach leads to abstract transitions that may aggregate multiple paths over transient states, but only one of them will be covered by the Euler tour, limiting error discovery. Since the

[1] The coverage analyzer and the RTG engine in Figure 11 are the same as already described in Chapter 1.
[2] It employs thread barriers to create a global serialization of all instructions.

approach explicitly encodes *what* has to be covered and fixes the way of traversal, it cannot exploit coverage evolution dynamically.

Bayesian networks were exploited for DTG (FINE et al., 2006). Statistical inference is used to build a Bayesian network for providing the most probable generator settings that would achieve a certain coverage goal. This technique requires a training phase to establish the basis for future decision making. However, the need for training may become a drawback, unless its contribution to the overall effort can be kept negligible.

Mcjammer (WAGNER; BERTACCO, 2008) decomposes the product FSM into dichotomic FSMs that capture the protocol behavior from the perspective of each core domain. Multiple intelligent agents, each working at a distinct core domain, cooperate to improve the overall transition coverage.

Genetic Programming offers another approach for building new tests from old ones. For instance, McVerSi (ELVER; NAGARAJAN, 2016) tailors the fitness function for improving memory consistency verification. 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.

Our prior model-based work (described in Chapter 4) focused on ordering random tests to favor coverage evolution, but it did not allow *dynamic* coverage control, and it relied on *greedy* heuristics that induced fixed neighborhoods, which hampered faster evolution when exploring the RTG space. This motivated the more general formulation proposed in the next section and the hybrid approach proposed in Section 5.4, where a data-driven engine focuses on coverage-controlled neighborhood exploration and a generalized model-driven engine focuses on ordering the tests induced by a given neighborhood.

5.2 DTG FORMULATION AS AN OPTIMIZATION PROBLEM

Let N and S be the sets of allowed values for the parameters n and s (respectively) that are within user-defined bounds and are induced by the range of a function[3] $f(i) = 2^i$, and let the values allowed for parameter k be bounded for each allowed value of s and be constrained to be multiples[4] of k, as follows:

$$N = \{n : n_{min} \leq n \leq n_{max} \wedge n = 2^i \, for\, some\ i \in \mathbb{N}\},$$
$$S = \{s : s_{min} \leq s \leq s_{max} \wedge s = 2^i \, for\, some\ i \in \mathbb{N}\},$$
$$K = \{k : (1 \leq k \leq s) \wedge (s \in S) \wedge (s \mod k = 0)\}.$$

Each *test* to be synthesized by the RTG engine is specified by a setting of the parameters corresponding to a point in a three-dimensional *generation space* $\mathbb{G}{=}N{\times}S{\times}K$. A collection of such points induces the generation of a *test suite*.

[3] This could be replaced by another function without loss of generality, as far as the *perturbation* function defined in Section 5.4.1 is accordingly adjusted.

[4] This constraint results in uniform competition of locations for cache sets, which benefits coverage control, as will be explained in Section 5.4.2

Problem 1. *Given $\mathbb{G}=N\times S\times K$, find an ordered subspace $G_{opt}\subseteq\mathbb{G}$ that maximizes coverage in minimum time.*

Note that Problem 1 ranks coverage first among the two objectives. The most important difference between Problem 1 and conventional coverage-driven RTG instances is the finding of an *ordered* subset of constraints, because that is the key to shaping coverage evolution, as shown in the next section.

5.3 THE DIRECTING ENGINE AT WORK: AN EXAMPLE

We encode every candidate solution as a subset $G\subseteq\mathbb{G}$. Our Directing Engine iteratively explores solutions induced by a neighborhood function. It combines the current solution and a neighbor into a new solution whose coverage is not inferior.

To illustrate how it works, let us arbitrarily select an encoding (G) representing a solution, as shown in Figure 12a. The x-axis displays all pairs in $S\times K$ induced by a range of locations between 4 and 32. The y-axis shows all values in N (in logarithmic scale) for a range of operations between 1Ki and 8Ki. The Explorer provides the encoding G to the Driver, which serializes its points (as depicted by the numbering above the marks) in an order that speeds up coverage evolution. Such serialization is induced by alternately favoring replacement and collision events. A *replacement* results from accesses to distinct locations that happen to induce the eviction of a block from cache. A *collision* (GHARACHORLOO, 1995) results from accesses to the same location. As detailed in Section 5.4.2, an odd (even) number represents a test where replacements (collisions) are favored.

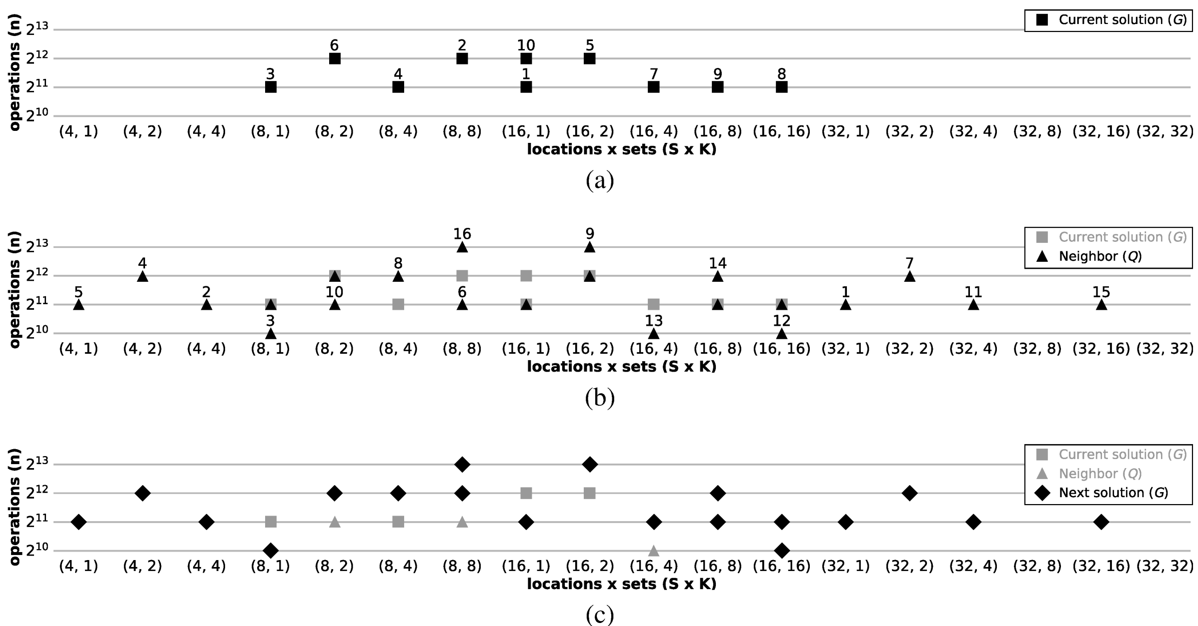


Figure 12 – Explorer and Driver at work: how the exploration of neighbors influences the exploitation of constraints within the Directing Engine.

From the current solution (G), the Explorer generates an encoding for a neighbor (Q). Figure 12b illustrates the encoding of one *possible* neighbor of G. A few of the neighbor's points may coincide with the current solution's. Such points are not sent to the Driver, as shown by the unnumbered marks. This avoids re-executing the same tests.

Then the Explorer combines the current solution and its neighbor into a new solution that certainly does not lead to inferior coverage. A possible outcome of such combination is depicted in Figure 12c. From this new solution, the entire process is repeated until a termination criterion is satisfied.

5.4 THE DIRECTING ENGINE: ALGORITHMS

Instead of trying to find the optimal subspace G_{opt}, we rely on local search for an approximate solution. We let $V \subset \mathbb{G}$ denote the collection of points that have been *visited*, and we let $cv(X)$ denote the cumulative coverage of all tests induced by an arbitrary collection $X \subset \mathbb{G}$.

From user-defined bounds, Algorithm 2 induces the generation space (line 2), as described in Section 5.2. If no initial solution is provided, it randomly defines one (lines 5-7) from which local search is launched (line 8).

1: **procedure** DIRECTING ENGINE$(G_0, n_{\min}, n_{\max}, s_{\min}, s_{\max})$
2: $\quad\mathbb{G} \leftarrow$ GENERATION SPACE$(n_{\min}, n_{\max}, s_{\min}, s_{\max})$
3: $\quad V \leftarrow \emptyset$
4: $\quad$**if** $G_0 = \emptyset$ **then**
5: $\quad\quad G_0 \leftarrow \{(n,s,k) : (n,s,k) \in \mathbb{G} \wedge n = n_{min}\}$
6: $\quad\quad s_{chosen} \leftarrow$ RANDOM$(\{s : (n,s,k) \in G_0\})$
7: $\quad\quad G_0 \leftarrow \{(n,s,k) : (n,s,k) \in G_0 \wedge s \leq s_{chosen}\}$
8: $\quad G \leftarrow$ EXPLORER(G_0)
9: $\quad$**return** G

Algorithm 2 – The algorithm underlying the generalized Directing Engine.

5.4.1 The Data-Driven Explorer

Algorithm 3 iteratively generates neighboring solutions (line 4), and keeps the best solution so far (line 8) until a termination criterion is satisfied (line 9). It invokes the Driver for the initial solution (line 2) and for every generated neighbor (line 7). To save time, it invokes the Driver only to points of the neighbor that have never been visited before. Between such major steps, it treats the cases where the neighbor is either singular or redundant (lines 5-6), i.e. either all candidate points happened to be out of bounds or have already been visited.

Algorithm 4 combines the points of the current solution and its neighbor (i.e. $G \cup Q$) and reduces them to a subset (G') that certainly has the same cumulative coverage.

From the current solution (G), Algorithm 5 generates another solution (Q) according to a neighborhood function. It first defines an empty dictionary (D) to register which candidate

```
1:  procedure EXPLORER(G)
2:      DRIVER(G)
3:      repeat
4:          Q ← GENERATE NEIGHBOR(G)
5:          if Q \ V = ∅ then
6:              Q ← GENERATE NEIGHBOR(V)
7:          DRIVER(Q \ V)
8:          G ← REDUCE NEIGHBOR(G, Q)
9:      until cv(V) = 1 ∨ TIMEOUT() ∨ (V = 𝔾)
10:     return G
```

Algorithm 3 – The algorithm underlying the Explorer.

```
1:  procedure REDUCE NEIGHBOR(G, Q)
2:      G' ← ∅
3:      for all (n, s, k) ∈ G ∪ Q do
4:          if cv(G' ∪ {(n, s, k)}) > cv(G') then
5:              G' ← G' ∪ {(n, s, k)}
6:      return G'
```

Algorithm 4 – The algorithm of the auxiliary routine Reduce Neighbor.

points will be used (or not) for encoding the neighbor (line 2). Then, for each point belonging to the current solution, it selects candidate points for the neighbor (lines 4-5). Finally, the neighbor is encoded with the selected points (lines 6-8).

```
1:  procedure GENERATE-NEIGHBOR(G)
2:      let D : 𝔾 → {TRUE, FALSE} be an empty dictionary
3:      Q ← ∅
4:      for each (n, s, k) ∈ G do
5:          D ← SELECT-NEIGHBORING-POINTS((n, s, k), D)
6:      for each (n, s, k) ∈ D do
7:          if D[(n, s, k)] = TRUE then
8:              Q ← Q ∪ {(n, s, k)}
9:      return Q
```

Algorithm 5 – The algorithm of the auxiliary routine Generate Neighbor.

Our approach uses a function to map a point of the current solution to candidate points for the neighbor, as follows:

Definition 5. *The perturbation function* $\pi : \mathbb{G} \to \mathscr{P}(\mathbb{G})$ *maps each point* (n,s,k) *to a subset of the generation space* $\{(2n,s,k),(n,2s,k),(n,s,2k),(n/2,s,k),(n,s/2,k),(n,s,k/2)\}$.

Note that π can induce whatever point of the generation space, thereby not excluding optimal solutions a priori. However, multiple criteria could be used for the *selection* of neighboring points. Without loss of generality, but for compactness, this article shows a single way of tailoring the neighborhood. Given a point (n,s,k) of the current solution, Algorithm 6 randomly selects neighboring points that are within bounds.

1: **procedure** SELECT-NEIGHBORING-POINTS($(n,s,k),D$)
2: **for each** $p \in \pi((n,s,k))$ **do**
3: **if** $p \in \mathbb{G} \land p \notin D$ **then**
4: $D[p] \leftarrow$ RANDOM($\{$TRUE, FALSE$\}$)
5: **return** D

Algorithm 6 – The algorithm of the auxiliary routine Select Neighboring Points.

5.4.2 The Model-Based Driver

The Driver dynamically exploits constraints when commanding the RTG engine[5]. The choice of constraints relies on a model that (pessimistically) assumes that transitions are induced either by replacements or by collisions (as discussed in Chapter 4). The Driver applies a constraint to induce a test favoring replacements and then a different constraint to induce the next test favoring collisions. The alternation between these two types of bias makes transitions *less likely to be revisited*, which favors coverage evolution. To induce such alternation, the Driver lowers the probability of transitions from one type in an attempt to raise the probability of transitions of the other type. The control on replacement events is the key to enabling such alternation, as follows.

For higher controllability, we enforce uniform competition. Remind that k defines the number of distinct cache sets for which the s shared locations compete. For each value of s, we constrain the values that can be assigned to k such that exactly s/k locations compete for each cache set. Such *uniform* distribution maximizes the probability of controlling replacements in all sets for a given setting of s.

Let α denote the associativity of a cache. For inducing a replacement event in a given cache set, a sequence of at least $\alpha+1$ references to *distinct* locations competing for that set is required. Therefore, a necessary condition for enabling replacement is $s/k \geq \alpha+1$. Conversely, a sufficient condition for disabling replacement in all sets is $s/k < \alpha+1 \Leftrightarrow s/k \leq \alpha$. Thus, there is a threshold s/α for the value of k above which replacement is certainly disabled, but below which it may be enabled depending on the sequence of references that turns out to be generated randomly. Note that such threshold is different for hierarchical levels with typically distinct degrees of associativity. This indicates that multiple values of k should be exploited for controlling transitions in the FSMs at all levels.

Therefore, the Driver can stimulate the intended alternation by selecting values of k that enable or disable replacement events. Such selection relies on the following model. Let $N_R(set(a))$ denote the average number of replacement events for the cache set assigned to the memory block where location a resides. An increase in $N_R(set(a))$ raises the probability of transitions induced by replacements. We have to count the number of replacement events, which depends on the associativity and on the memory access pattern. The ratio n/k measures how

[5] The RTG engine (in its turn) enforces such constraints with the *biasing* technique (described in Chapter 3) to induce successive tests.

many operations are mapped to the same cache set on average. A best-case access pattern for replacement events is such that every element of a sequence of n/k accesses makes reference to distinct locations mapped to the same set. Replacement takes place at every $\alpha+1$ such accesses. Thus, an upper bound for the average number of replacements is $N_R^{max} = (n/k)/(\alpha+1)$. A worst-case pattern for replacement events is such that every element of a subsequence of n/s accesses makes reference to the same location, then another subsequence of n/s accesses makes reference to another location, and so on. There are s such subsequences, s/k of them map to the same set (on average), and replacement takes place at every $\alpha+1$ transitions between them. Thus, a lower bound for the average number of replacements is $N_R^{min} = (s/k)/(\alpha+1)$.

Thus, we can estimate the number of replacement-induced transitions as *proportional* to the average number of operations per set, i.e. n/k (best case), or locations per set, i.e. s/k (worst case). The Driver tries to either maximize or minimize that number for stimulating either replacement-induced or collision-induced transitions. Note that, although the model is transition-oriented, it just serves as a *proxy* for (indirectly) increasing coverage under whichever metric is adopted by the design environment. Algorithm 7 relies on such mechanism for setting a distinct constraint on each random test so as to favor either replacements or collisions.

1: **procedure** DRIVER(G)
2: $\quad x \leftarrow 0$
3: $\quad$**repeat**
4: $\quad\quad P^* \leftarrow \{(s,k) : \exists (n,s,k) \in G\}$
5: $\quad\quad$**repeat**
6: $\quad\quad\quad$**if** $x = 0$ **then**
7: $\quad\quad\quad\quad k^* \leftarrow \min\{k : (s,k) \in P^*\}$
8: $\quad\quad\quad\quad s^* \leftarrow \max\{s : (s,k) \in P^* \wedge k = k^*\}$
9: $\quad\quad\quad\quad x \leftarrow 1$
10: $\quad\quad\quad$**else**
11: $\quad\quad\quad\quad s^* \leftarrow \min\{s : (s,k) \in P^*\}$
12: $\quad\quad\quad\quad k^* \leftarrow \max\{k : (s,k) \in P^* \wedge s = s^*\}$
13: $\quad\quad\quad\quad x \leftarrow 0$
14: $\quad\quad\quad n \leftarrow \min\{n : (n,s,k) \in G \wedge s = s^* \wedge k = k^*\}$
15: $\quad\quad\quad \mathrm{RTG}(n,s^*,k^*)$
16: $\quad\quad\quad G \leftarrow G \setminus \{(n,s^*,k^*)\}$
17: $\quad\quad\quad P^* \leftarrow P^* \setminus \{(s^*,k^*)\}$
18: $\quad\quad\quad V \leftarrow V \cup \{(n,s^*,k^*)\}$
19: $\quad\quad$**until** $(P^* = \emptyset) \vee (cv(V) = 1)$
20: $\quad$**until** $(G = \emptyset) \vee (cv(V) = 1)$

Algorithm 7 – The algorithm underlying the Driver.

We let x be a Boolean value, and we let $\mathrm{RTG}(n,s,k)$ denote the invocation of the RTG engine for creating a test.

The outer loop (lines 3-20) iteratively visits the points corresponding to a given candidate solution G until all are visited or full coverage is reached. In each iteration, P^* keeps the generation subspace $S \times K$ induced by the points of the candidate solution that have not been visited yet.

The inner loop (lines 5-19) visits those points in an order that tries to maximize coverage in minimal time. In each iteration, lines 6-13 induce an alternation between a test favoring replacements and a test favoring collisions, according to the above model. Besides, some ranking is used to better control the effect of alternation. Lines 7-8 rank the choice of k before s to stimulate replacement at as most hierarchical levels as possible, whereas lines 11-12 rank the choice of k after s to avoid replacement at as most levels as possible, thereby stimulating collisions at most of them. Once a pair (s^*, k^*) is chosen, the inner loop selects the point (n, s^*, k^*) corresponding to the minimum test size (line 14). This selection is another attempt to increase coverage in minimal time. The inner loop induces the generation of a test in each iteration (line 15), and removes the selected point from those to be visited (line 16).

As a result, successive iterations of the inner loop induce tests where the intended alternation is applied to points (n, s^*, k^*) with *possibly distinct* test sizes, as opposed to our early technique (described in Chapter 4), whose heuristics restricted exploration to neighborhoods corresponding to *fixed* test sizes. Thus, Algorithm 7 is a generalization that improves the synergy between Driver and Explorer.

Finally, let us assess the impact of the proposed approach on time complexity. Test generation is dominated by the time complexity of the RTG engine, which is polynomial (ANDRADE; GRAF; SANTOS, 2020). The complexity of the verification algorithms underlying the adopted MCM checker is also polynomial (FREITAS; RAMBO; SANTOS, 2013).

5.5 EXPERIMENTAL EVALUATION

5.5.1 Experimental setup

We evaluated the *hybrid* test generator (HTG) built with the proposed approach as compared to pure *model-based* and *data-driven* generators. The former is the Coverage-driven Test Generator (CTG) built with our early technique (ANDRADE et al., 2018). The latter is the state-of-the-art McVerSi (ELVER; NAGARAJAN, 2016) test generator (MTG), which is available in the public domain (ELVER, 2016). We preserved all genetic parameters exactly as they were originally set in (ELVER; NAGARAJAN, 2016).

We relied on a checker similar to (FREITAS; RAMBO; SANTOS, 2013) and on gem5's infrastructure (BINKERT et al., 2011) for simulation (*O3*, *Ruby*, and *simple* as CPU, memory, and network models). For the designs, we adopted either a 2-level (L1, L2) MOESI or a 3-level (L0, L1, L2)[6] MESI directory protocol with 4KB (directed-mapped) private caches at L0, 64KB (2-way) private caches at L1, and a 2MB (8-way) shared L2 cache, all with same block size (64 bytes).

For observing to which extent the generators are independent of coverage metric, they were all evaluated under two different metrics: one tracking *structural* coverage, another track-

[6] This is the original labeling used in the gem5 implementation, which conceptually corresponds to the standard levels (L1, L2, L3).

ing *functional* coverage. While running a test, we tracked the number of distinct transitions covered in the code of each cache controller's FSM. For the structural metric, no distinction was made between identical controller instances in distinct core domains (as in (ELVER; NAGARAJAN, 2016)). However, a functional metric should distinguish memory events in multiple core domains as a result of every coherence transaction. Since the global product of all local FSMs leads to a state space that grows exponentially with core count, an useful metric should avoid full enumeration by restricting the scope while still capturing the impact of the most relevant memory events. That is why we adopted a functional metric that distinguishes transitions between identical controller instances in distinct core domains.

The generators differ in which parameters are *statically* defined by the user and which are *dynamically* set by their engines. The MTG requires the static definition of a *fixed* test length and a *single* address-space constraint, while the HTG and CTG dynamically exploit *variable* test lengths and *multiple* address-space constraints (as a result of varying the parameter k). We report results for the MTG under the (best) static constraint defined in (ELVER; NAGARAJAN, 2016): the test memory size TM=8KB. McVersi is free to select as many locations as available within its useful address space. For HTG and CTG, however, the number of locations must belong to a pre-specified range. To accommodate such difference, we compared the generators under the same upper bound on the number of (useful) memory locations. Under the adopted constraint that enforces uniform distribution of locations over cache sets, the maximum number of block-aligned locations is 128 TM=8KB. As a result, we adopted $S = \{4, 8, 16, 32, 64, 128\}$ for HTG and CTG.

Table 15 – Studied errors for MESI 3-level designs

ID	State	Input event	Next state	Precluded output action
E.5.1 (L1)	IS_I	Data_all_Acks	I	writeDataFromL2Response
E.5.2 (L1)	IS	Inv	IS instead of IS_I	(preserved)
	SM	Data_all_Acks	M	(preserved as in (IM, M))
	SM	Data	SM	(preserved as in (IM, SM))
E.5.3 (L1)	IS_I	DataS_fromL1	I	writeDataFromL2Response
E.5.4 (L1)	E_IL0	WriteBack	MM_IL0	writeDataFromL0Request
E.5.5 (L1)	S	L0_Invalidate_Own	SS instead of S_IL0	forward_eviction_to_L0

We first measured the cumulative coverage resulting from the execution of a sequence of tests on designs containing *no errors*. Then we inserted different artificial errors by changing the FSMs (either by modifying the next state or precluding some due output action)[7]. Each faulty design contained a single, distinct error. The errors studied in our experiments are described in Tables 15 and 16.[8] We measured the runtime until the error was found or until the

[7] Although our checker is also able to find consistency errors, we focused on coherence errors for experimental convenience, without loss of generality.

[8] Among the MESI 3-level design errors used in the previous chapter, we selected – for the experiments reported

Table 16 – Studied errors for MOESI 2-level designs

ID	State	Input event	Next state	Precluded output action
e.5.1 (L1)	SI	Writeback_Ack_Data	I	Data block in sendData
e.5.2 (L2)	ILXW	L1_WBDIRTYDATA	M	writeDataToCache
e.5.3 (L1)	OM	Fwd_GETS	OM	Data block in sendData
e.5.4 (L2)	ILOXW	L1_WBCLEANDATA	M	writeDataToCache
e.5.5 (L1)	SM	Fwd_GETS	SM	Data block in sendData

Directing Engine stopped generation. Each generated test was executed five times under different simulation states (not related to the test itself) in such a way that the distinct executions of the same test are all perturbed differently (ELVER; NAGARAJAN, 2016). To get the reported values, we launched each generator ten times with different seeds, and we took the median values of the resulting distributions. Runtimes were measured in an Intel Xeon E5430 workstation (2.66 GHz, 8GB memory).

5.5.2 Structural coverage evolution

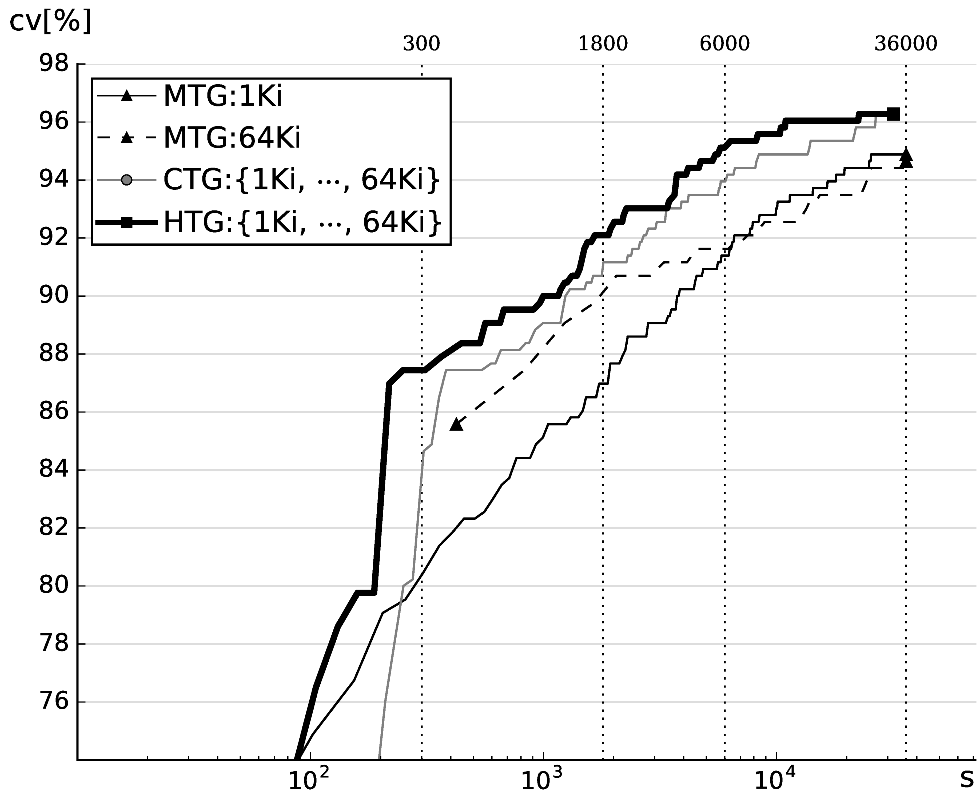
Since the initial verification phase does not display interesting aspects of the generators under comparison (due to transitions that are easy to stimulate randomly), we focus on the intermediate phase (after a few tests have been executed) and the final phase (after many tests have been executed), where sophisticated techniques are needed for increasing coverage.

Figure 13a shows the evolution for a MESI 3-level design. Let us first focus on the MTG's behavior. In the intermediate phase, large tests pay off. As compared to n=1Ki, tests with n=64Ki lead to higher coverage (until they break even around 6000s). In the final phase, short tests pay off instead, because increasing coverage values (above a certain threshold) becomes more dependent on the MTG's genetic algorithm, which is sensitive to test throughput. As a result, for covering 94.65% of the transitions, the MTG required 7 hours when n=1Ki, but around 10 hours when n=64Ki.

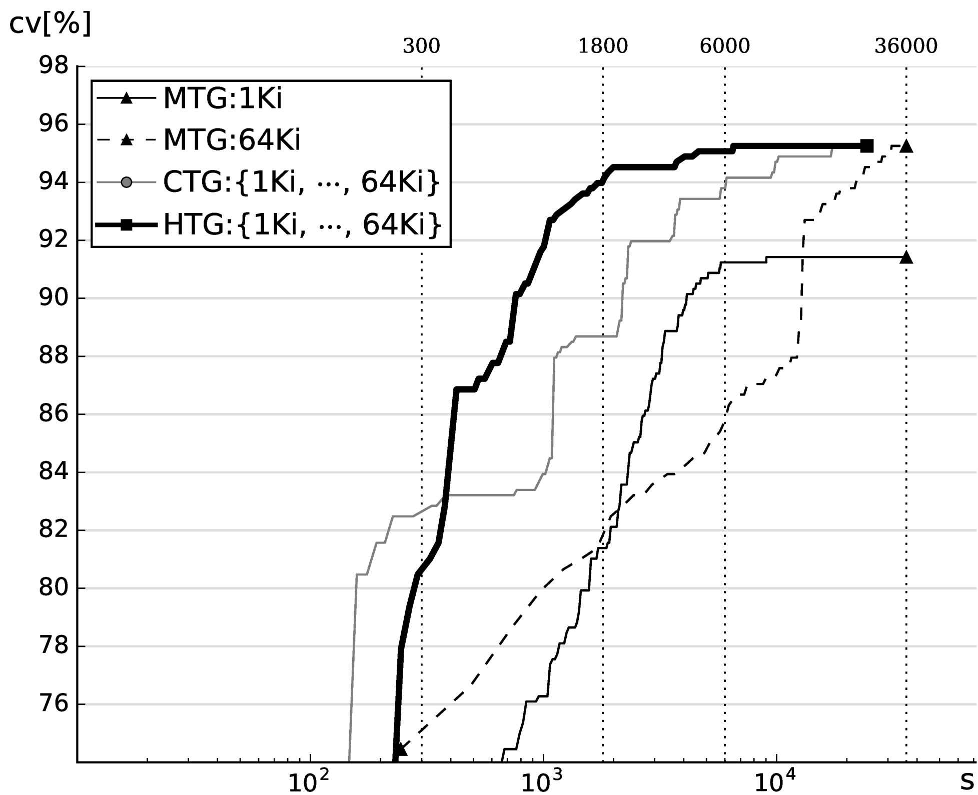
Let us now compare the CTG's behavior with the MTG's. Since the CTG is not data-driven, it stops generation as soon as the generation space is exhausted (in this case, after 9 hours). As opposed to the MTG, the CTG automatically adjusts the test size during verification. The CTG became superior to the MTG after running for 5 minutes (for n=1Ki) and 30 minutes (for n=64Ki). After that, the CTG always led to higher coverage. The MTG reached its maximal coverage (94.88%) after 7 hours, whereas the CTG reached that same coverage after only 2.5 hours, and attained its own maximal coverage (96.28%) after 7.5 hours.

Finally, let us compare the HTG with the others. HTG and MTG reached their highest

in this chapter – the ones that were the hardest to find with all generators. This focus on the most challenging errors is an attempt to capture design scenarios that actually deserve sophisticated generators.



(a)



(b)

Figure 13 – Structural coverage evolution for 32-core designs as a function of the required time (limited to 10 hours): (a) MESI 3-level and (b) MOESI 2-level.

coverage values, i.e. 96.28% and 94.88% in around 6 and 7 hours, respectively. Although both allow dynamic coverage control, the HTG reached the MTG's maximal coverage 4.6 times faster. This indicates that the HTG's exploration of neighborhoods and exploitation of constraints (when properly coupled) can lead to better evolution than genetic-based learning. Albeit both HTG and CTG dynamically exploit constraints, the former exhibited superior coverage evolution from the intermediate up to the final phase. Thus, the superiority of the HTG over the CTG does not come from their similar model-based policy (serialization of constraints), but it is due to the HTG's data-driven policy (exploration of neighborhoods). The CTG favors test throughput: it *fully* explores a plane of the generation space corresponding to the minimum test size, before fully exploring the next plane corresponding to a larger test size. The results indicate that the HTG's policy is better for two reasons: (1) the *partial* exploration of a plane of generation space allows the early run of a few larger test programs, which favors coverage growth, and (2) the perturbation function (devoid of built-in heuristics) is more effective for exploring multiple test sizes in the same suite.

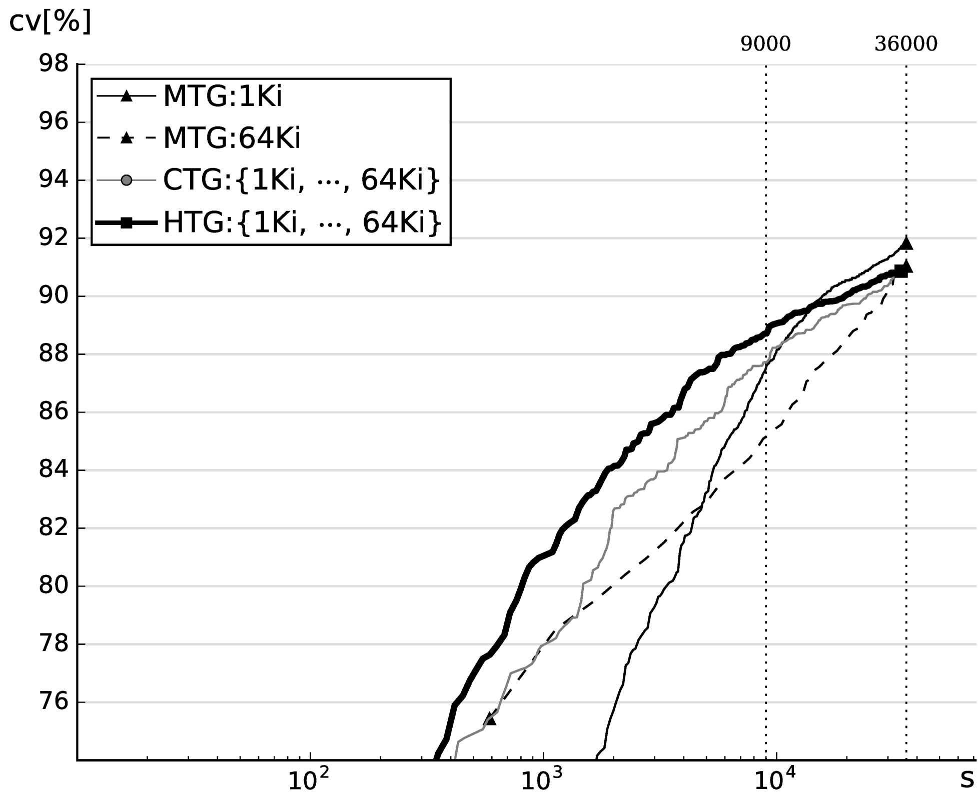
Figure 13b shows that the more complex MOESI 2-level protocol enhances the differences between the generators. The MTG's behavior was more sensitive to test size: it is able to achieve the same final coverage as the HTG only for n=64Ki. The CTG was always superior to the MTG (after the 3 initial minutes). The HTG reached the maximal coverage 4.8 times faster than the MTG, although both exploit dynamic coverage control. Besides, the HTG stopped generation after 7 hours (when the whole generation space was explored), while the MTG kept generating tests up to the 10-hour time limit. After 7 min, the HTG became superior to the CTG until they reach the same coverage (95.26%), which clearly shows the impact of the novel data-driven Explorer.

The contrast between Figure 13a and Figure 13b indicate that designs relying on more complex protocols (such as MOESI) are likelier to benefit from the novel data-driven Explorer and from the generalized model-based Driver, because their synergy leads to higher controllability.

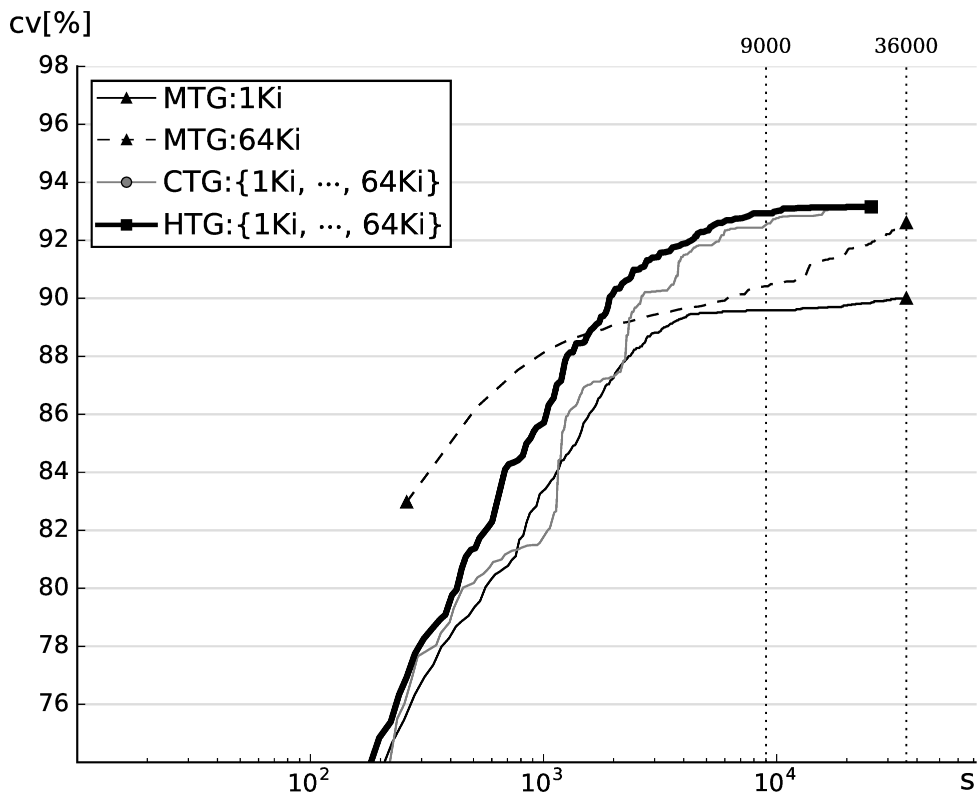
5.5.3 Functional coverage evolution

Figure 14a shows the functional coverage evolution for a MESI 3-level design. Similarly to what happened for the structural metric, short tests pay off for the MTG at the final verification phase. After 2.5 hours, HTG, CTG, and MTG covered, respectively, 88.72%, 87.73%, and 87.53% of the transitions. After 10 hours, both HTG and CTG ended up reaching the same coverage of 90.96%, while the MTG reached 91.04% for n=64Ki and 91.84% for n=1Ki. This is the only scenario where higher final coverage was observed for the MTG as compared to the HTG. The most likely difference to explain this behavior seems the MTG's exploitation of finer granularity for the amount of shared locations.

Figure 14b shows the functional coverage evolution for a MOESI 2-level design. Similarly to what was observed for the structural metric, the MTG reached higher coverage (92.62%)



(a)



(b)

Figure 14 – Functional coverage evolution for 32-core designs as a function of the required time (limited to 10 hours): (a) MESI 3-level and (b) MOESI 2-level.

for the largest test size than for the shortest one (90.01%). However, even after running for 10 hours, the MTG covered less transitions (92.62%) than the HTG was able to cover (93.15%) in only 5.5 hours. This contrasts with the *structural* coverage evolution observed for MOESI 2-level, where the MTG reached the same final coverage as the HTG. This indicates that the HTG can reach higher coverage than the MTG, because the higher complexity of the MOESI protocol is better captured by the more expressive metric. Similarly to what was observed for the structural metric, the HTG displayed better evolution than the CTG, but the same final coverage (93.15%).

By contrasting all four coverage evolution scenarios, we conclude that our hybrid approach was superior in most cases, especially for the more complex MOESI protocol. This indicates that a well-designed hybrid approach is likely to cope with the growing complexity of protocols used in high-end designs, which is better tracked by more expressive coverage metrics.

5.5.4 Error discovery rate and detection time

Tables 17 and 18 show the median time (in seconds) required by each technique for exposing every error. They show the best values in bold and report (between parentheses) how many (out of ten) test suites exposed an error.

Table 17 – Time for finding errors in MESI 3-level 32-core designs =in seconds

		MTG		CTG	HTG
Error	Metric	1Ki	64Ki	{1Ki, ..., 64Ki}	{1Ki, ..., 64Ki}
E.5.1	structural	138 (10)	57 (10)	109 (10)	**29** (10)
	functional	89 (10)	72 (10)	114 (10)	**20** (10)
E.5.2	structural	622 (10)	166 (10)	114 (10)	**69** (10)
	functional	827 (10)	160 (10)	126 (10)	**75** (10)
E.5.3	structural	524 (10)	493 (10)	352 (10)	**43** (10)
	functional	919 (10)	798 (10)	362 (10)	**61** (10)
E.5.4	structural	7370 (8)	2752 (10)	**261** (10)	665 (10)
	functional	5491 (9)	2649 (10)	**690** (10)	838 (10)
E.5.5	structural	11958 (10)	17532 (8)	2026 (10)	**821** (10)
	functional	11110 (10)	23980 (8)	1769 (10)	**851** (10)

For MESI 3-level designs, all CTG and HTG suites exposed every error (while some MTG suites failed to expose E.5.4 and E.5.5). The HTG required the least time in all cases but E.5.4. In this case, the HTG required a slightly worse effort (10–14 minutes) than the CTG (4–11 minutes), but significantly better than the MTG (1.5–2 hours to uncover E.5.4). The HTG was from 1.65 to 8 times faster than the CTG (except for E.5.4, for which the CTG was 2.5 times faster). The HTG was from 1.65 to 28 times faster than the MTG.

Table 18 – Time for finding errors in MOESI 2-level 32-core designs.

Error	Metric	MTG 1Ki		MTG 64Ki		CTG {1Ki, ..., 64Ki}		HTG {1Ki, ..., 64Ki}	
e.5.1	structural	8	(10)	48	(10)	8	(10)	**7**	(10)
	functional	8	(10)	47	(10)	8	(10)	**7**	(10)
e.5.2	structural	8	(10)	47	(10)	8	(10)	**7**	(10)
	functional	8	(10)	48	(10)	8	(10)	**7**	(10)
e.5.3	structural	49	(10)	48	(10)	78	(10)	**35**	(10)
	functional	42	(10)	51	(10)	75	(10)	**25**	(10)
e.5.4	structural	2692	(5)	2643	(10)	1041	(10)	**641**	(10)
	functional	4191	(5)	2552	(10)	**1043**	(10)	1073	(10)
e.5.5	structural	3863	(3)	2861	(10)	982	(10)	**586**	(10)
	functional	4011	(2)	1965	(10)	1050	(10)	**1044**	(10)

For MOESI 2-level designs, the HTG required the least time in nine out of ten cases, where it was from 1.14 to 3 times faster than CTG, and it was from 1.14 to 6.6 times faster than the MTG. The exception was e.5.4 under the functional metric, for which the CTG was slightly faster than the HTG (in this case, the CTG's greedy shorter-test-size-first heuristic happened to pay off). The HTG and the CTG were able to expose *every* error in less than 18 minutes, while the MTG took as much as one hour and 10 minutes. Besides, the MTG (for n=1Ki) was unable of exposing errors in e.5.4 and e.5.5 for 50% and 70–80% of the test suites.

Overall, the HTG was able to consistently expose every error in *all* trials, independently of metric and protocol, while the MTG's error discovery rate was sensitive to metric and test size. Thus, the HTG seems more robust to cope with different verification scenarios.

5.6 CONCLUSIONS

Our experimental evaluation relied on 54483 test runs, each one executed five times in distinct simulation conditions, i.e. 272415 test executions in total. The results show that hybrid coverage-driven RTG can be superior to purely data-driven or model-based approaches, not as a result of plain composition, but due to proper *decoupling* of competences: data-driven (neighborhood) exploration and model-based (constraint) exploitation. The results also indicate that a well-designed hybrid approach is likely to pay off as designs tend to rely on more complex protocols, being effective in discovering design errors (quite independently of the coverage metric adopted in a given design environment). By allowing superior coverage evolution and high error discovery rate, a hybrid approach can raise the confidence in face of tight verification deadlines.

6 A COMPARISON OF APPROACHES TO DIRECTED TEST GENERATION

This chapter describes an experimental evaluation of the Directing Engines proposed in this thesis (used by the hybrid HTG and by the model-based CTG) as compared to the one used by a recently reported directed test generator based on Reinforced Learning (RLG), which is a pure data-driven approach (PFEIFER et al., 2020). The author of this thesis influenced the design of the RLG with respect to the choice of Random Test Generation (RTG) engine, and he helped coupling it to the new Directing Engine, which was developed by a fellow co-worker[1]. Like the CTG (proposed in Chapter 4) and the HTG (proposed in Chapter 5), the RLG uses the same RTG engine proposed in Chapter 3. An important consequence of such choice is that it enables a comparison of different approaches for exactly the same generation space and under *the same level of constraint exploitation* (HTG, CTG, and RLG all rely on *chaining* and *biasing*, and all use the same granularity for the generation parameters, as opposed to the MTG, the third-party generator employed in previous chapters as a reference for evaluation).

Section 6.1 summarizes the key ideas underlying the Directing Engine of the RLG. Section 6.2 compares RLG, CTG, and HTG as representatives of data-driven, model-based, and hybrid approaches. Section 6.3 draws the conclusions on that comparison.

6.1 A BRIEFING ON THE KEY IDEAS BEHIND THE RLG

The RLG (PFEIFER et al., 2020) addresses DTG as a decision process. It models the definition of a sequence of tests as a decision process, and it tries to find a decision-making policy that maximizes cumulative rewards resulting from successive actions taken by an agent.

The formulation of the design process is as follows. The *environment* includes an RTG engine, the simulator, and the Coverage Analyzer. The Directing Engine is formulated as an *agent* that takes *actions* in such environment. The Coverage Analyzer interprets the environment into a *state* representation and a *reward* value is assigned to each action taken in a given state. The equation defining reward values is a function of coverage and time.

Since the agent interacts with the environment through the RTG engine's interface, actions are formulated in terms of parameters of the RTG engine. The RLG version used for comparison in the next section employs the parameters n, s, and k (defined in Chapter 3), and it takes six distinct actions based on them. The implementation employs an adaptation of the Rainbow agent (HESSEL et al., 2018), where the original deep neural network was replaced by a recurrent neural network.

[1] The Directing Engine used by the RLG will be described in detail – together with experimental evidence on *learning*-specific issues – as part of the MSc. dissertation by Nícolas Pfeifer, who is the first author of the paper where the RLG was reported (PFEIFER et al., 2020). The author of this thesis is a co-author of that paper.

6.2 EXPERIMENTAL EVALUATION

In the previous chapter, we have evaluated the proposed *hybrid* test generator (HTG) with respect to pure *model-based* and *data-driven* generators (CTG and MTG, respectively). This section extends such evaluation to also compare the HTG to the RLG, a data-driven generator where Machine Learning is exploited to *direct* random test generation for synthesizing programs. This complement is relevant, because the MTG – in contrast – is a data-driven generator where Machine Learning is exploited to *directly generate* test programs (without the use of an RTG engine). Since the RLG relies on the same constrained random test generator as the HTG and the CTG, this section reports results under the same experimental setup described in Section 5.5 (including the errors specified in Tables 15 and 16). Regarding the setup for the RLG itself, we preserved the three-parameter actions described in Pfeifer et al. (2020).

6.2.1 Structural coverage evolution

Figure 15a shows the coverage evolution spanning the intermediate and the final phases of test generation (but omitting the initial phase) for a MESI 3-level design. Let us first focus on the comparison of generators whose policy of constraint exploitation is the same (i.e. HTG, CTG, and RLG).

In the intermediate phase, the HTG exhibited the best coverage evolution, while the CTG was superior to the RLG (until they broke even after 14000 seconds). The worse evolution of the RLG in the intermediate phase means that the initial phase was not long enough for it to learn how to properly constrain random test generation. Indeed, proper learning takes place exactly during the intermediate phase. That is why, only at the final phase, the RLG displays similar evolution as the CTG. This indicates that data-driven techniques should be expected to be less effective at early phases of the verification process, as opposed to a model-based technique. Although a data-driven technique could be expected to show its advantage during the final phase, the results show otherwise. The RLG has actually shown a worse evolution than the CTG. After running for 2.5 hours, the CTG reached 94.88% coverage while the RLG reached 93.14% (and the HTG reached 95.58%). The HTG displayed the best evolution. While the RLG reached its highest coverage value (95.81%) after running for around 6 hours, the HTG obtained the same value in 3 hours (i.e. the HTG was 2 times faster). At first glance, the superiority of the HTG *could* be seen as resulting from the simple combination of approaches: the built-in model allows for better evolution in the intermediate phase, and the data-driven engine in the final phase. However, the fact that the RLG – neither was able to match the HTG's maximal coverage nor the CTG's – provides extra evidence of synergy between complementary techniques inside the Directing Engine of the HTG.

Now let us include the MTG into the analysis. During the intermediate phase, the MTG had a better coverage evolution than the RLG (for both test sizes, 1Ki and 64Ki), but the latter became superior in the final phase. Such behavior could be explained by distinct constraint

policies and different machine learning techniques, but it could also be put down to a major difference: the *direct* generation of test programs (in the MTG) and the *directed* random test generation (in the RLG). Regardless the difficulty in comparing MTG and RLG, it is actually the HTG that displayed the best overall coverage evolution. For instance, the HTG was 2.2 times faster than the RLG to reach the coverage value of 95.81%.

Figure 15b shows that the more complex MOESI 2-level protocol enhances the differences between the generators. Similarly to what happened under the other protocol, the HTG displayed the best overall coverage evolution. Therefore, the HTG's advantage seems independent of coherence protocol.

Moreover, note that, again, both HTG and CTG displayed a better coverage evolution than the RLG. Although these three generators reached the same final coverage value (95.26%), the HTG took 2 hours, the CTG needed 5 hours, and RLG spent 6 hours (i.e. the HTG was 3 times faster than the RLG).

Yet another similarity to the behavior observed for the MESI 3-level design is that the RLG started losing to the MTG, but displayed a better coverage evolution during the final phase. Therefore, such final advantage of the RLG over the MTG seems independent of coherence protocol. Note that the MTG has shown a better evolution than the RLG for the shortest test size (1Ki) until a saturation point. Such saturation can not be explained by the different constraint policies nor by the distinct learning techniques underlying those generators, because it does not happen to the largest test size (64Ki). Indeed, such saturation raises a known limitation of MTG, which requires a *fixed* test size for the whole test suite. This seems to indicate that the superiority of the RLG comes from its ability to shift from one test size to another when diminishing rewards are observed for a given test size.

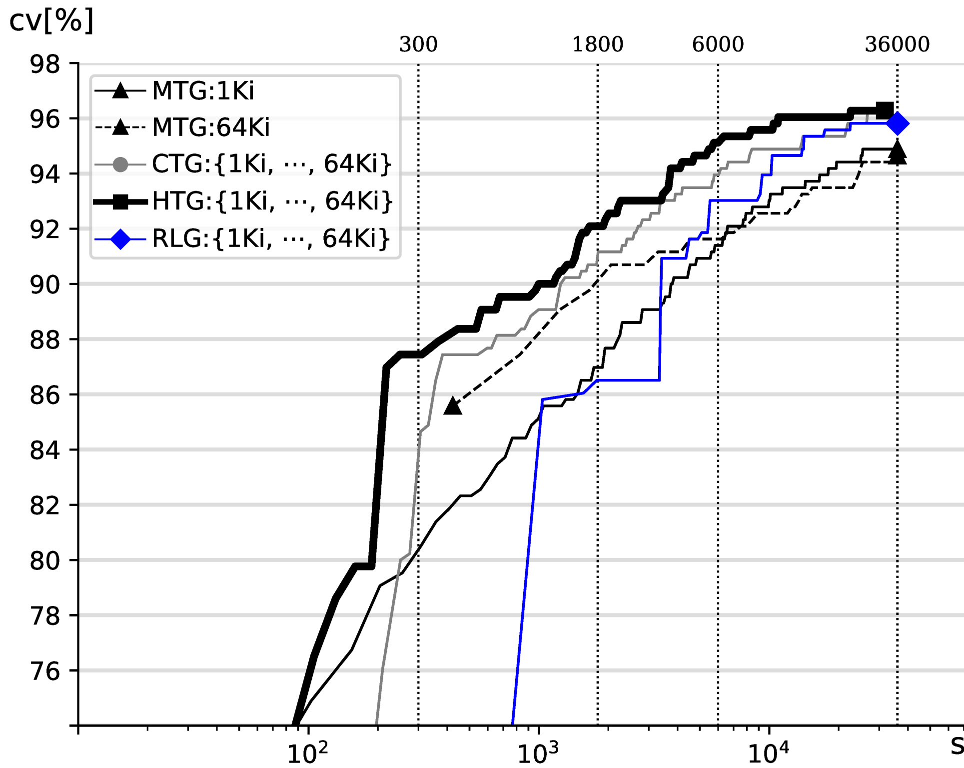
It should be noted that, although the RLG reached similar final evolution as the CTG under MESI 3-level, it failed to do the same under MOESI 2-level. This may indicate a limitation of RLG when targeting more complex protocols.

In conclusion, not only the HTG was superior to all others independently of protocol, but it exhibited a sharper evolution for the more complex of the two protocols. This makes it promising to targeting the complexities of future protocols.

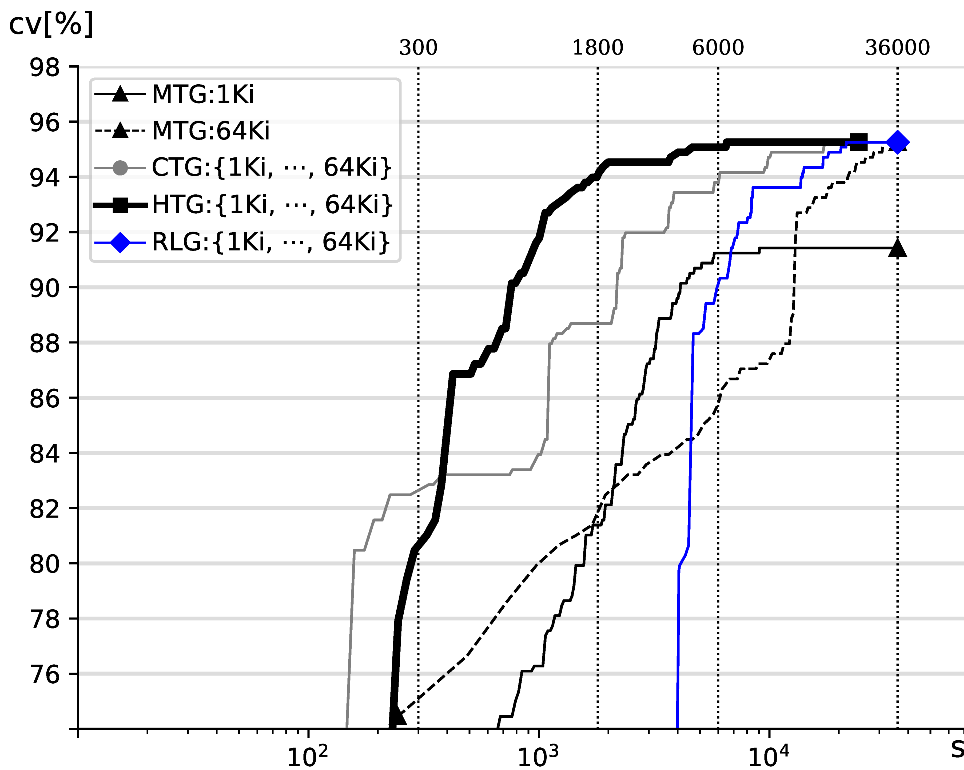
6.2.2 Functional coverage evolution

Figures 16a and 16b show the functional coverage evolution for MESI 3-level and MOESI 2-level designs, respectively. Similarly to what we have observed for the structural metric, the HTG displayed a better coverage evolution than the CTG and the RLG for both designs. Once more, the more complex protocol enhanced the differences between them.

For the MESI 3-level design, CTG, RLG, and HTG covered, respectively, 87.73%, 88.03%, 88.72% at the 2.5 hours mark (9000s). While the RLG required 10 hours to reach its highest coverage value (90.33%), the HTG required only 6.5 hours to reach the same coverage (i.e the HTG was 1.5 times faster).

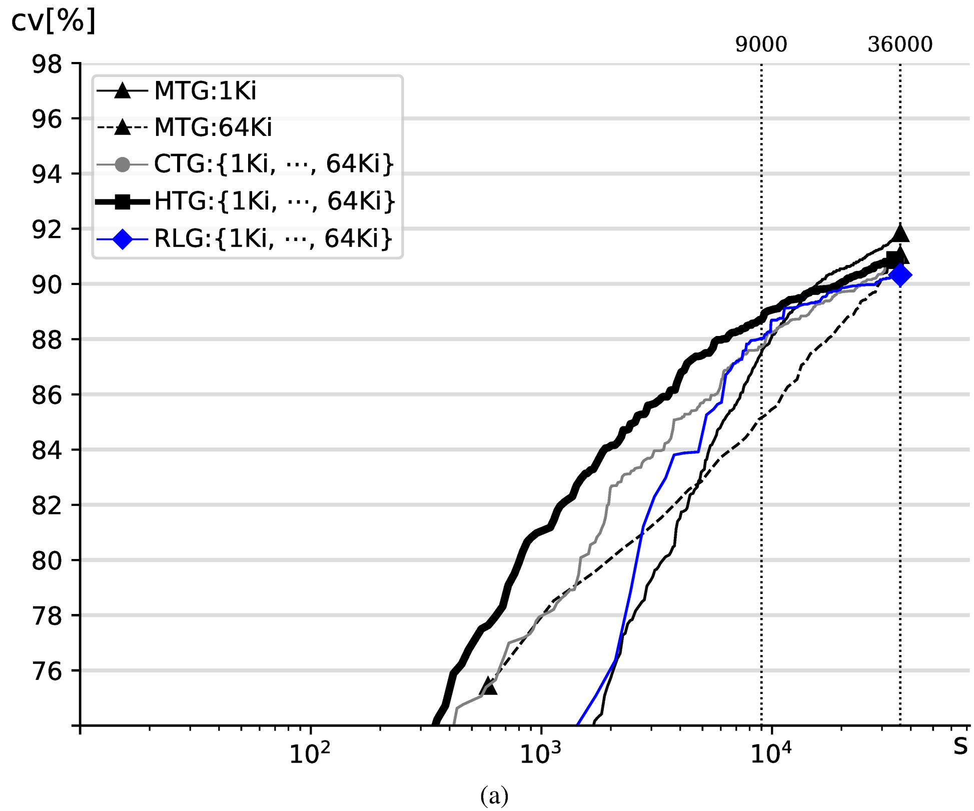


(a)

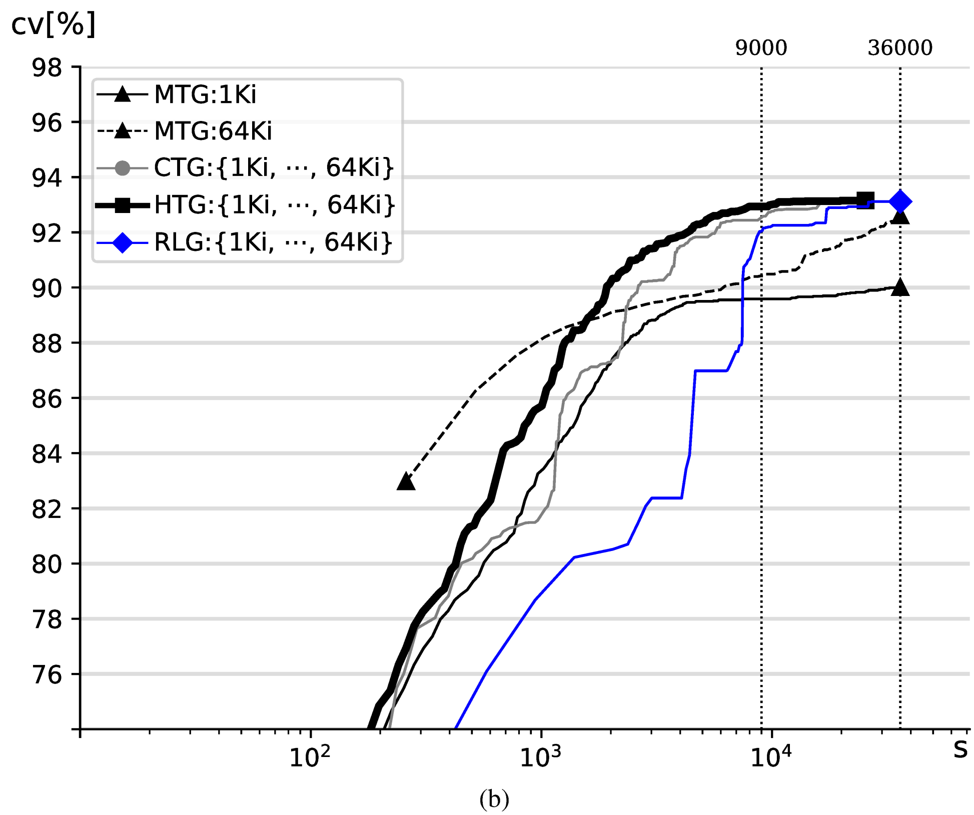


(b)

Figure 15 – Structural coverage evolution for 32-core designs as a function of the required time (limited to 10 hours): (a) MESI 3-level and (b) MOESI 2-level.



(a)



(b)

Figure 16 – Functional coverage evolution for 32-core designs as a function of the required time (limited to 10 hours): (a) MESI 3-level and (b) MOESI 2-level.

For the MOESI 2-level design, CTG, RLG, and HTG covered, respectively, 92.04%, 92.53%, and 92.93% after 2.5 hours. While the RLG required 7.5 hours to reach its highest coverage value (93.11%), the HTG required only 3.4 hours to reach the same coverage (i.e. the HTG was 2.2 times faster).

Therefore, for the same generation space and under the same policy of constraint exploitation, the experimental evidence indicates that the hybrid approach is superior to pure data-driven or model-based approaches, independently of protocol and coverage metric. Moreover, such advantage seems to be higher for more complex coherence protocols.

Recall that (as already reported in Chapter 5) the MTG reached the highest final coverage as compared to CTG and HTG for MESI 3-level under functional coverage. Even with its inclusion into the comparison, the RLG was also unable to reach the MTG's final coverage. Since CTG, HTG, and RLG all have the same generation space and exploit the same constraints, this complementary result reinforces the explanation given in the previous chapter. Such advantage of the MTG comes from the finer granularity it allows for one of the generation parameters: the amount of shared locations (s). However, note that, after 2.5 hours (9000s), the MTG reached the smallest coverage value among all four generators (i.e. 87.53%). The disadvantage of the MTG in terms of coverage evolution probably comes from the fixed test size (n) limitation, as already mentioned in the previous section).

6.2.3 Error discovery rate and detection time

Tables 17 and 18 show the median time (in seconds) required by each technique to expose every error. They show the best values in bold and report (between parentheses) how many (out of ten) test suites exposed an error.

For MESI 3-level designs, all CTG, HTG, and RLG test suites exposed every error. On the one hand, the RLG required the least time to expose E.5.1 and E.5.2 (in both cases), and E.5.3 (in one case). Note, however, that the times required by the HTG for those errors were competitive with the RLG's (both found E.5.1, E.5.2, and E.5.3 in less than one minute). This shows that for discovering design errors that are easy-to-find (the ones that are first discovered in the verification process), the hybrid approach may not always be advantageous. On the other hand, the HTG required significantly less time than the RLG to expose the two most challenging among the studied errors. The HTG was (from 2.4 to 4.3 times) faster than the RLG to expose E.5.4 and E.5.5, independently of the coverage metric. Since the verification process is over only after ensuring that *no* design error can be found after proper coverage is reached, the bottleneck for verification lies in the time required to expose the most challenging errors. Therefore, the advantage of the HTG over the RLG for E.5.4 and E.5.5 seems more significant in practice than the RLG's advantage for the three other errors.

For MOESI 2-level designs, the HTG required the least time in 7 out of 10 cases, for which it was from 2.4 to 7.2 times faster than the RLG. In 2 out of 10 cases, the RLG required the least time. The RLG was from 1.9 to 2.3 times faster than the HTG to expose e.5.3,

Table 19 – Time for finding errors in MESI 3-level 32-core designs.

Error	Metric	MTG 1Ki	MTG 64Ki	CTG {1Ki, ..., 64Ki}	HTG {1Ki, ..., 64Ki}	RLG {1Ki, ..., 64Ki}
E.5.1	structural	138 (10)	57 (10)	109 (10)	29 (10)	**12** (10)
	functional	89 (10)	72 (10)	114 (10)	20 (10)	**12** (10)
E.5.2	structural	622 (10)	166 (10)	114 (10)	69 (10)	**36** (10)
	functional	827 (10)	160 (10)	126 (10)	75 (10)	**36** (10)
E.5.3	structural	524 (10)	493 (10)	352 (10)	**43** (10)	59 (10)
	functional	919 (10)	798 (10)	362 (10)	61 (10)	**59** (10)
E.5.4	structural	7370 (8)	2752 (10)	**261** (10)	665 (10)	2846 (10)
	functional	5491 (9)	2649 (10)	**690** (10)	838 (10)	2198 (10)
E.5.5	structural	11958 (10)	17532 (8)	2026 (10)	**821** (10)	2335 (10)
	functional	11110 (10)	23980 (8)	1769 (10)	**851** (10)	2039 (10)

Table 20 – Time for finding errors in MOESI 2-level 32-core designs.

Error	Metric	MTG 1Ki	MTG 64Ki	CTG {1Ki, ..., 64Ki}	HTG {1Ki, ..., 64Ki}	RLG {1Ki, ..., 64Ki}
e.5.1	structural	8 (10)	48 (10)	8 (10)	**7** (10)	17 (10)
	functional	8 (10)	47 (10)	8 (10)	**7** (10)	17 (10)
e.5.2	structural	8 (10)	47 (10)	8 (10)	**7** (10)	18 (10)
	functional	8 (10)	48 (10)	8 (10)	**7** (10)	18 (10)
e.5.3	structural	49 (10)	48 (10)	78 (10)	35 (10)	**13** (10)
	functional	42 (10)	51 (10)	75 (10)	25 (10)	**13** (10)
e.5.4	structural	2692 (5)	2643 (10)	1041 (10)	**641** (10)	4509 (10)
	functional	4191 (5)	2552 (10)	**1043** (10)	1073 (10)	6127 (10)
e.5.5	structural	3863 (3)	2861 (10)	982 (10)	**586** (10)	4262 (10)
	functional	4011 (2)	1965 (10)	1050 (10)	**1044** (10)	3739 (10)

respectively). As already pointed out in the previous chapter, the CTG was the best generator in a single case (e.5.4 under functional coverage). In this case, the HTG was the second best generator after the CTG (and the HTG was 5.7 times faster than the RLG). Similarly to what was observed for MESI 3-level designs, the HTG was the fastest for exposing the most challenging errors (the HTG was from 3.6 to 7.2 times faster than the RLG for e.5.4 and e.5.5).

In short, the HTG was able to consistently expose every error in all trials, independently of metric and protocol. While the RLG required more than 30 minutes to expose the most challenging errors, the HTG exposed all errors in less than 20 minutes. Thus, even when compared to the RLG, the HTG seems more robust to cope with different verification scenarios and more efficient to expose errors.

6.3 CONCLUSIONS

This chapter reported an experimental evaluation relying on 54923 test runs, each one executed five times in distinct simulation conditions (i.e. 274615 test executions in total). It compared three *coverage-directed* constrained random test generators under different approaches: model-based, data-driven, and hybrid. Besides, it contrasted their results with those obtained by a directed test generator that does not rely on constrained random test generation. The results showed that the ability of a generator to provide proper coverage and error discovery in less time does not come simply from the choice of the technique adopted to direct generation (e.g. Reinforcement Learning or Genetic Programming), but it is strongly dependent on constraint exploitation and generation space exploration. Besides, they also showed that a data-driven technique that exploits coverage feedback (e.g. RLG) tends to be superior to a data-driven technique that does not (e.g. MTG).

The experiments provided evidence that coverage-directed *constrained* random test generation leads to superior coverage evolution with time when an hybrid approach enables proper synergy between model-based and data-driven engines. Despite the superiority of the hybrid approach, one scenario was found for which a directed test generator (not relying on an RTG engine) was able to reach higher coverage. However, this does not seem to be due to a limitation of the hybrid approach itself, but a limitation on the choice of granularity for one of the parameters defining the generation space (which could be redefined, for instance, by changing the neighborhood function). On the other hand, the results have shown that a limitation due to a fixed test size (as in the MTG) is a poor choice for proper coverage evolution whatever the approach adopted for directed-test generation.

7 CONCLUSIONS AND PERSPECTIVES

Design verification and prototype testing have long been exploiting constraints on random test generation. In the context of shared-memory validation, conventional constraints capture basic properties of parallel programs (such as the number of memory operations, the number of threads, and the number of shared variables) for synthesizing non-deterministic tests, which are likelier to expose anomalous behavior.

This thesis addressed (random and directed) constrained test generation by exploiting non-conventional constraints capturing properties of caches and coherence protocols. It has shown that their adequate exploitation tends to improve the quality of non-deterministic tests in terms of error discovery, coverage, and effort. Since such properties were selected to be general enough so as to avoid ties to specific protocols or hierarchies, the proposed techniques that exploit them have the potential to be useful within a large range of designs and verification environments.

The experimental results have shown that *chaining* and *biasing* tend to improve the quality of tests synthesized by *random* test generation (Chapter 3), by pure *model-based* directed test generation (Chapter 4), by pure *data-driven* directed test generation (Chapter 6), and by *hybrid* directed test generation (Chapters 5). The keys to such improvement lies in the use of canonical dependence chains to foster *conflict* events involving the same address (i.e. races) and the use of proper address assignment to foster *eviction* events involving distinct addresses (i.e. replacements).

The thesis has proposed two novel directed test generation techniques:

- A model-based generator (CTG) that is driven by a coverage model where conflict and eviction events are used to estimate transition coverage, which is used as proxy to the actual coverage metric, thereby granting large independence from the metric adopted in different verification environments.

- An hybrid generator (HTG) that combines the exploration of neighborhoods (data-driven) and the exploitation of constraints (model-based) in such a way that optimal solutions are kept in the *search* space.

This thesis has shown that, when *chaining* and *biasing* are used to control data races and cache replacements, the proposed hybrid approach (HTG) is superior (in terms of coverage evolution) as compared with the proposed model-based approach (CTG) and with the two most recently reported (pure) data-driven generators: one based on Genetic Programming (MTG), another on Reinforcement Learning (RLG).

The exploitation of constraints capturing other general shared-memory phenomena can be devised as future work. For instance, the pragmatic decision of restraining biasing to eviction events for directed test generation let unexploited the biasing for fostering false sharing events

(although the generic biasing technique described in Chapter 3 originally supports it). This is likely to improve the quality of learning-based generators.

Moreover, it has been shown that, when learning-based techniques are used to build directed test generators (such as Genetic Programming in the MTG and Reinforcement Learning in the RLG), they are unlikely to display superior coverage evolution if: (1) they do not exploit constraints capturing general shared-memory properties *dynamically* (as it happens for MTG) and (2) they do not rely on some model to guide coverage evolution while they are *still learning* (as it happens for both RLG and MTG).

Since next generation Electronic Design Automation (EDA) tools are expected to rely on learning techniques, their use for directed test generation is likely. The findings reported in this thesis indicate that, for reaching high coverage in less time in future generation approaches, it is not sufficient to rely on learning-based data-driven approaches. Although the innovation may well come from advanced learning, it should not overlook the legacy from constrained random test generation nor the efficiency of proper coverage models.

The contributions of this thesis were reported in two journal articles (ANDRADE; GRAF; SANTOS, 2020; ANDRADE et al., 2020) and two conference papers (ANDRADE et al., 2018; PFEIFER 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, 3 2004. ISSN 0740-7475.

ADIR, A.; SHUREK, G. Generating concurrent test-programs with collisions for multi-processor verification. In: **7th IEEE International High-Level Design Validation and Test Workshop**. [S.l.]: IEEE, 2002. p. 77–82.

ADVE, S. V.; GHARACHORLOO, K. Shared Memory Consistency Models: a Tutorial. **Computer**, IEEE, v. 29, n. 12, p. 66–76, 12 1996. ISSN 0018-9162.

ALGLAVE, J. et al. GPU Concurrency: Weak Behaviours and Programming Assumptions. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 50, n. 4, p. 577–591, 3 2015. ISSN 0362-1340.

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

ANDRADE, G. A. G. **Exploiting canonical dependence chains and address biasing constraints to improve random test generation for shared-memory verification**. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, Florianópolis, 2017.

ANDRADE, G. A. G. et al. Steep Coverage-ascent Directed Test Generation for Shared-memory Verification of Multicore Chips. In: **IEEE/ACM International Conference on Computer-Aided Design**. New York, NY, USA: Association for Computing Machinery, 2018. 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, v. 39, n. 12, p. 5295–5303, 12 2020. ISSN 0278-0070.

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**. [S.l.]: IEEE, 2016. p. 552–559.

ANDRADE, G. A. G.; GRAF, M.; SANTOS, L. C. V. dos. Chaining and Biasing: Test Generation Techniques for Shared-Memory Verification. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 39, n. 3, p. 728–741, 3 2020. ISSN 1937-4151.

ARM. **ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile**. [S.l.], 2018. Disponível em: https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

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, 8 2011. ISSN 0163-5964.

CHEN, Y. et al. Fast Complete Memory Consistency Verification. In: **15th International Symposium on High Performance Computer Architecture**. [S.l.]: IEEE, 2009. p. 381–392.

DEVADAS, S. Toward a Coherent Multicore Memory Model. **Computer**, IEEE, n. 10, p. 30–31, 2013.

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: **International Symposium on High Performance Computer Architecture**. [S.l.]: IEEE, 2016. p. 618–630.

ESMAEILZADEH, H. et al. Dark Silicon and the End of Multicore Scaling. In: IEEE. **38th Annual International Symposium on Computer Architecture**. [S.l.], 2011. p. 365–376.

ESMAEILZADEH, H. et al. Dark Silicon and the End of Multicore Scaling. **IEEE Micro**, IEEE, v. 32, n. 3, p. 122–134, 2012.

FINE, S. et al. Harnessing Machine Learning to Improve the Success Rate of Stimuli Generation. **IEEE Transactions on Computers**, IEEE Computer Society, Washington, DC, USA, v. 55, n. 11, p. 1344–1355, 10 2006. ISSN 1557-9956.

FINE, S.; ZIV, A. Coverage Directed Test Generation for Functional Verification Using Bayesian Networks. In: **40th Annual Design Automation Conference**. New York, NY, USA: ACM, 2003. p. 286–291. ISBN 1-58113-688-9.

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 Test in Europe Conference Exhibition**. [S.l.]: IEEE, 2013. p. 631–636. ISBN 978-1-4503-2153-2. ISSN 1530-1591.

GHARACHORLOO, K. **Memory consistency models for shared-memory multiprocessors**. Tese (Doutorado) — Stanford University, 1995.

GRAF, M. et al. Spec&Check: An Approach to the Building of Shared-Memory Runtime Checkers for Multicore Chip Design Verification. In: IEEE/ACM. **International Conference on Computer-Aided Design (ICCAD)**. [S.l.], 2019. v. 39, n. 3, p. 728–741. ISSN 0278-0070.

HANGAL, S. et al. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. **SIGARCH Computer Architecture News**, Association for Computing Machinery, New York, NY, USA, v. 32, n. 2, p. 114–123, 3 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., 2019. ISBN 978-0-12-811905-1.

HESSEL, M. et al. Rainbow: Combining Improvements in Deep Reinforcement Learning. **Proceedings of the AAAI Conference on Artificial Intelligence**, AAAI, v. 32, n. 1, 4 2018.

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

IBM. **POWER9 Processor User's Manual**. [S.l.], 2019. Disponível em: https://ibm.ent.box.com/s/tmklq90ze7aj8f4n32er1mu3sy9u8k3k.

LUSTIG, D. et al. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In: **International Conference on Architectural Support for Programming Languages and Operating Systems**. [S.l.]: Association for Computing Machinery, 2017. p. 661–675. ISBN 978-1-4503-4465-4.

LYU, Y. et al. Directed Test Generation for Validation of Cache Coherence Protocols. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 38, n. 1, p. 163–176, 1 2019. ISSN 0278-0070.

MANOVIT, C.; HANGAL, S. Completely verifying memory consistency of test program executions. In: **International Symposium on High-Performance Computer Architecture**. [S.l.]: IEEE, 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**, Association for Computing Machinery, v. 55, n. 7, p. 78–89, 6 2012.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**. 2th. ed. [S.l.]: Morgan Kaufmann Publishers Inc., 2020. ISBN 9780128203316.

PFEIFER, N. et al. A Reinforcement Learning Approach to Directed Test Generation for Shared Memory Verification. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.]: IEEE, 2020. p. 538–543. ISSN 1558-1101.

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

RAMBO, E.; HENSCHEL, O.; SANTOS, L. C. V. dos. Automatic generation of memory consistency tests for chip multiprocessing. In: **International Conference on Electronics, Circuits and Systems**. [S.l.: s.n.], 2011. p. 542–545.

RAMBO, E. A.; HENSCHEL, O. P.; SANTOS, L. C. V. dos. On ESL verification of memory consistency for system-on-chip multiprocessing. In: **Conference on Design, Automation, and Test in Europe**. [S.l.]: EDA Consortium, 2012. p. 9–14. ISSN 1530-1591.

ROY, A. et al. Fast and Generalized Polynomial Time Memory Consistency Verification. In: BALL, T.; JONES, R. B. (Ed.). **18th International Conference on Computer Aided Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. v. 4144, p. 503–516. ISBN 978-3-540-37411-4.

SHACHAM, O. et al. Verification of Chip Multiprocessor Memory Systems Using a Relaxed Scoreboard. In: **41st IEEE/ACM International Symposium on Microarchitecture**. Los Alamitos, CA, USA: IEEE Computer Society, 2008. p. 294–305. ISBN 978-1-4244-2836-6.

TRIPPEL, C. et al. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In: **22nd International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: Association for Computing Machinery, 2017. p. 119–133. ISBN 978-1-4503-4465-4.

WAGNER, I.; BERTACCO, V. MCjammer: Adaptive Verification for Multi-core Designs. In: **Conference on Design, Automation, and Test in Europe**. New York, NY, USA: Association for Computing Machinery, 2008. p. 670–675. ISSN 1530-1591.

WATERMAN, A.; ASANOVI, K. **The RISC-V Instruction Set Manual Volume I: Unprivileged ISA**. [S.l.], 2019. Disponível em: https://riscv.org/specifications/.

ZHANG, M. et al. PVCoherence: Designing Flat Coherence Protocols for Scalable Verification. **IEEE Micro**, v. 35, n. 3, p. 84–91, 5 2015. ISSN 0272-1732.