UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Nícolas Pfeifer

AVALIAÇÃO EXPERIMENTAL DE UM GERADOR DE TESTES DIRIGIDOS PARA A VERIFICAÇÃO DE MEMÓRIA COMPARTILHADA EM MULTICORE CHIPS

Florianópolis

2018

Nícolas Pfeifer

AVALIAÇÃO EXPERIMENTAL DE UM GERADOR DE TESTES DIRIGIDOS PARA A VERIFICAÇÃO DE MEMÓRIA COMPARTILHADA EM MULTICORE CHIPS

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Luiz Cláudio Villar dos Santos

Florianópolis

2018

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

Pfeifer, Nícolas
Avaliação experimental de um gerador de testes dirigidos para a verificação de memória compartilhada em multicore chips / Nícolas Pfeifer ; orientador, Luiz Cláudio Villar dos Santos, 2018.
p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2018.

Inclui referências.

1. Ciências da Computação. 2. Verificação de memória. 3. Geração de testes. 4. Memória compartilhada. 5. CMP. I. Villar dos Santos, Luiz Cláudio. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

RESUMO

No contexto de verificação de memória compartilhada em multicore chips, este trabalho propõe uma avaliação crítica da geração automática de testes dirigidos quando baseada em Programação Genética. A metodologia consiste na comparação do gerador McVerSi (que representa o estado da arte) com geradores de testes aleatórios (que representam a base de geradores dirigidos por cobertura). Dois geradores de testes aleatórios são utilizados: um deles (McVerSi_Rand) pressupõe uma restrição do espaço de endereçamento imposta estaticamente antes de disparar a geração, o outro (IRTG) admite a variação dinâmica de restrições impostas ao espaço de endereçamento. Os três geradores são comparados de acordo com duas métricas: cobertura estrutural dos controladores de cache e esforço requerido na detecção de erros de coerência de memória.

Palavras-chave: Memória Compartilhada, Geração de Testes, Verificação, CMP, EDA

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

O aumento da frequência de relógio e a exploração do paralelismo entre instruções permitiu um crescimento exponencial do desempenho de processadores até se atingir a chamada barreira de potência por volta de 2004, que corresponde à máxima dissipação de potência térmica possível com sistemas habituais de resfriamento (PATTERSON; HENNESSY, 2013). Desde então, fabricantes de microprocessadores construíram processadores com múltiplos núcleos de processamento em um único chip, o que ficou conhecido como *Chip Multiprocessor* (CMP), em uma tentativa de manter o crescimento de desempenho.

CMPs requerem um método de comunicação entre os processadores. Arquiteturas baseadas em troca de mensagem fornecem a cada processador uma memória local que só pode ser acessada por ele e requerem que os processadores se comuniquem através de mensagens. Arquiteturas baseadas em memória compartilhada, entretanto, fazem com que toda a memória seja acessível por todos os processadores, possibilitando a comunicação através de escritas e leituras da memória.

Como a abstração de uma única memória possibilita que múltiplos processadores escrevam e leiam da memória simultaneamente, programadores necessitam de um modelo conceitual da semântica das operações de memória para que possam utilizar a memória compartilhada corretamente (GHARACHORLOO, 1995). Este modelo é chamado de modelo de consistência de memória (MCM). Ele instrui o programador sobre a ordem esperada de execução de instruções das múltiplas *threads* de um programa paralelo. O MCM determina o grau de relaxação da ordem de programa e o grau de atomicidade das escritas (ADVE; GHARACHORLOO, 1996).

Na busca por melhoria de desempenho, os fabricantes passaram a usar modelos de consistência de memória relaxados, onde as operações de memória podem completar fora da ordem especificada no programa e onde as escritas feitas em memória não precisam ser imediatamente visíveis a todos os núcleos ao mesmo tempo. Um MCM pressupõe a existência de um protocolo para permitir a coerência de cache. Espera-se que tais protocolos continuem sendo implementados em hardware para *multicore chips* de propósito geral operando sob memória compartilhada, mesmo para centenas de núcleos (DEVADAS, 2013).

A influência de ambas as tendências, o aumento da quantidade de núcleos em CMP e a implementação de protocolos de coerência em *hardware* corroboram no aumento da complexidade de projetos de no-

vos processadores, tornado-os mais propensos a erros. Por isto, é cada vez mais necessário ferramentas que auxiliem projetistas a detectar erros e garantir o funcionamento correto do projeto, de acordo com as especificações.

1.1 O PAPEL DA VERIFICAÇÃO FUNCIONAL

Verificação funcional é a tentativa de demonstrar que a intenção de um projeto é preservada na sua implementação (PIZIALI, 2008). A verificação pretende aproximar a intenção do projetista, a especificação formulada e a implementação. Ela é, em sua essência, um processo comparativo entre o que o processador deveria ser (intenção e especificação) e o que ele atualmente é (implementação).

No contexto de CMP, o processo de verificação consiste em executar um programa de teste paralelo em um simulador da representação do projeto de um *multicore chip* e avaliar se o comportamento observado corresponde ao esperado. Os programas de testes são gerados automaticamente e a avalização do comportamento é também automaticamente realizada. As ferramentas responsáveis por estas tarefas são chamadas de gerador e *checker*, respectivamente. Sabe-se que o uso de programas paralelos relativamente curtos, que utilizam poucas posições de memória e têm condições de corrida agressivas expõem erros mais rapidamente (MANOVIT; HANGAL, 2006). A primeira sugestão de uso do modelo de memória como base da construção de *checkers* foi feita por Hangal et al. (2004). A utilização do modelo de memória na construção de *checkers* é muito importante pois possibilita o reuso de ferramentas de verificação, não limitando sua utilidade somente ao processador para o qual foi construída.

Ferramentas de teste em processadores podem ser usadas no contexto de pós-fabricação ou pós-silício, onde um protótipo do processador já foi criado e é utilizado para executar o programa de teste. Entretanto, corrigir um erro de projeto nesta etapa, após já se ter construído um protótipo, é demasiadamente caro. Por isto, a verificação pré-silício é utilizada, para se beneficiar do mais baixo custo de reparo de erros em etapas anteriores do projeto. Nesta, a execução do teste se dá por meio de simulação do projeto do *multiprocessor chip*, e assim, é ordens de magnitude mais lenta do que a execução física em protótipo. Por esta razão, não é prático utilizar os mesmos geradores e *checkers* utilizados na etapa pós-silício, como em Hangal et al. (2004), onde o teste é simplesmente pseudo-aleatório sem nenhuma direção. Em cenários

pré-silício, são necessárias ferramentas que dirijam a geração de testes para diminuir o esforço necessário para detecção, este tipo de abordagem pode ser vista em Elver e Nagarajan (2016) e Wagner e Bertacco (2008), por exemplo.

1.2 OBJETIVO

Com o intuito de entender como as diferentes características de geradores de testes no contexto de verificação do subsistema de memória em CMP influenciam a detecção de erros, este trabalho tem como objetivo a avaliação de diferentes métodos de geração de testes, especialmente geração baseada em métodos aleatórios e geração utilizando programação genética.

Para essa avaliação serão usados duas métricas complementares:

- o esforço requerido para a detecção de erros;
- a cobertura estrutural dos controladores de cache que implementam o protocolo de coerência.

1.3 CONTRIBUIÇÃO TÉCNICA

Para atingir o objetivo acima, este trabalho compara três geradores de testes distintos descritos na literatura, denominados *Conventional Random Test Generator* (CRTG) (ELVER; NAGARAJAN, 2016), *Improved Random Test Generator* (IRTG) (ANDRADE, 2017) e *McVerSi Test Generator* (MTG) (ELVER; NAGARAJAN, 2016). Os autores de dois deles (CRTG e MTG) disponibilizaram seu código em domínio público (ELVER, 2016). Os autores do terceiro gerador (IRTG) disponibilizaram seu código localmente.

Esses geradores possuem parâmetros diferentes e assumem restrições distintas, os quais dificultam a definição de condições adequadas a uma justa comparação. Ademais, a comparação direta desses geradores nunca foi reportada na literatura.

Assim, o trabalho descrito nesta monografia teve três contribuições técnicas:

- Proposta de condições experimentais para uma comparação justa.
- Comparação direta dos três geradores num mesmo ambiente experimental.

- Identificação do caráter promissor de uma das técnicas avaliadas para nuclear, em trabalho futuro, um gerador de testes dirigidos.

1.4 ORGANIZAÇÃO DA MONOGRAFIA

Esta monografia está organizada da seguinte maneira: O Capítulo 2 trata da fundamentação teórica requerida para a compreensão do restante do conteúdo. O Capítulo 3 faz uma revisão dos geradores encontrados na literatura e aponta algumas de suas limitações. O Capítulo 4 discute os princípios de funcionamento dos geradores selecionados, suas principais características e propriedades dos testes gerados por estes. O Capítulo 5 descreve alguns aspectos importantes da infraestrutura utilizada para a execução dos experimentos deste trabalho, assim como apresenta os resultados experimentais de esforço e cobertura obtidos a partir da utilização dos geradores descritos no Capítulo 4. Por fim, o Capítulo 6 apresenta as conclusões deste trabalho e aponta possíveis trabalhos futuros relacionados a este tema.

2 FUNDAMENTAÇÃO

Neste capítulo serão feitas algumas definições necessárias para o entendimento do restante do trabalho. Inicialmente serão explicados os conceitos de consistência de memória compartilhada e coerência de cache, dois componentes importantes do modelo de consistência de memória, assim como o protocolo de coerência de cache, mecanismo que tem a função de garantir esta propriedade.

2.1 CONSISTÊNCIA DE MEMÓRIA COMPARTILHADA E COERÊNCIA DE CACHE

O modelo de consistência de memória busca responder à pergunta "Quais propriedades necessitam ser mantidas entre leituras e escritas em diferentes endereços nos diferentes processadores?"(HENNESSY; PATTERSON, 2011). Por exemplo, ele define o resultado que uma leitura deve retornar. O resultado correto de uma leitura é o valor da última escrita em seu endereço. Desta forma, o modelo define uma ordem entre as operações de escrita para o mesmo endereço, sejam elas emitidas pelo mesmo processador ou não. No entanto, se a definição do MCM não contemplar qual a ordem que deve ser observada entre escritas a endereços diferentes, os resultados de programas podem ser diferentes do previsto. O exemplo a seguir deixará mais claro como a relação entre escritas em endereço diferentes é um fator importante do MCM, e como essa definição é relevante para programadores.

O desencontro do modelo de memória implementado pelo CMP e o esperado pelo programador pode causar resultados inesperados. O código mencionado a seguir é um exemplo disso (adaptado de (HILL, 1998)). A Figura 1 contém trechos de código a serem executados por dois processadores, P1 e P2. A variável *data* é compartilhada entre os dois processadores e usada para a propagação de dados, e a variável *flag* é utilizada para sincronizar o seu acesso. Intuitivamente, o valor final esperado de *data_ copy* é *new*, porém, este cenário só irá acontecer se as duas escritas de P1 acontecerem na ordem especificada no programa. Se o modelo de memória implementado no CMP não garantir que todos os processadores enxergam a mesma ordem entre escritas em endereços diferentes, o compilador, ou até mesmo o processador, poderá reordenar estas instruções, podendo gerar *data_ copy* = *old*. Se o programador não estiver ciente disto e esperar que a ordem seja mantida, poderá

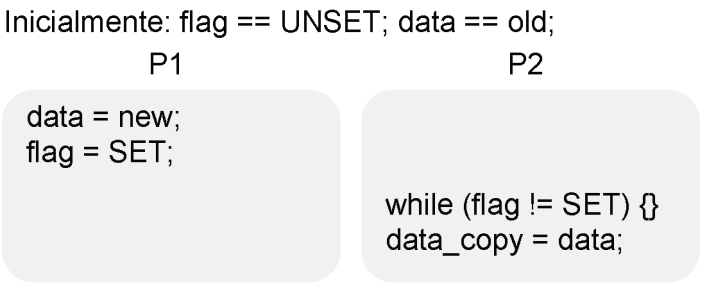


Figura 1 Exemplo de código que pode ter resultado não intuitivo dependendo do MCM implementado.

implementar um programa de forma errada, que nem sempre executará da maneira esperada.

A presença de memória cache em CMPs pode resultar em múltiplas cópias da mesma variável em processadores diferentes. Isto requer um meio de garantir que escritas se propaguem a todas as cópias. A coerência de cache é geralmente definida como duas condições: uma escrita deve ser visível por todos os núcleos e escritas em um mesmo endereço devem ser visíveis na mesma ordem por todos os núcleos (GHARACHORLOO et al., 1990). Um modelo de consistência de memória é especificado através de axiomas que definem em que cenários a ordem de programa deve ser garantidamente preservada em tempo de execução e em que medida a ordem de escritas deve ser observada pelos diferentes núcleos. A literatura reporta a descrição formal desses axiomas para diferentes MCMs (INTERNATIONAL INC., 1992; ADIR; ATTIYA; SHUREK, 2003; OWENS; SARKAR; SEWELL, 2009).

Essencialmente, um *checker* verifica se a ordem observada em tempo de execução satisfaz o comportamento especificado pelos axiomas de um MCM.

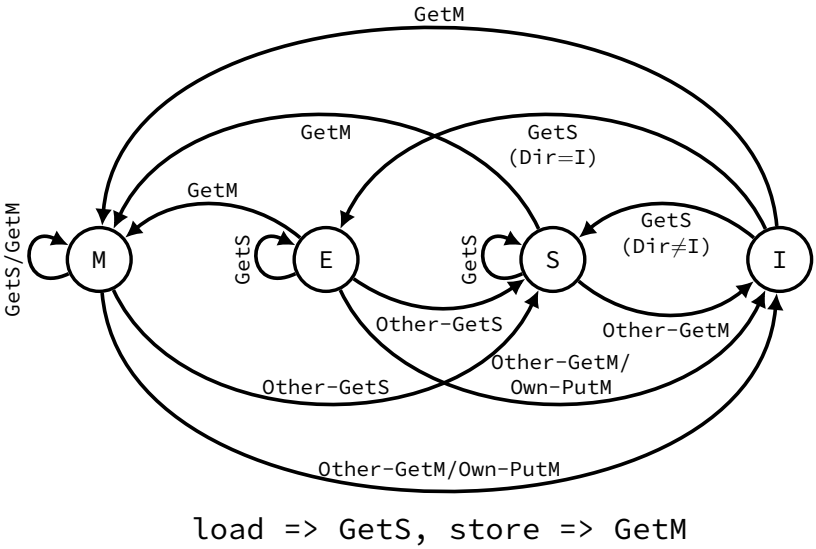
2.2 PROTOCOLOS DE COERÊNCIA

Os protocolos de coerência são um método de manter a coerência de cache. O protocolo MESI (PAPAMARCOS; PATEL, 1984) foi utilizado neste trabalho e é explicado a seguir. Cada bloco de cache de cada processador pode estar em um de quatro estados: *Modified*(M), *Exclusive*(E), *Shared*(S) e *Invalid*(I).

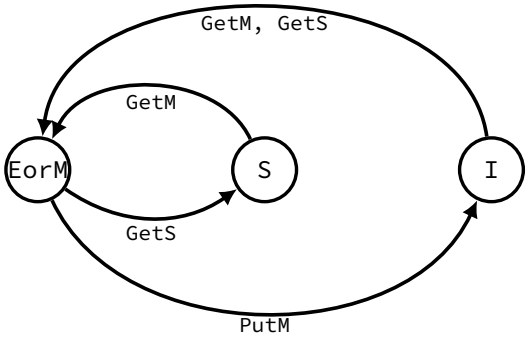
A Figura 2 mostra as máquinas de estado (*Finite-State Machine*

- FSM) simplificadas do protocolo MESI. A Figura 2a mostra a FSM do controlador de cache privativa, enquanto a Figura 2b mostra a FSM do diretório, uma entidade de supervisão global de todos os controladores de cache. Ambas as FSM são simplificadas e contêm apenas os estados estáveis. Os rótulos nas transições indicam o tipo de requisição de coerência emitidas pelo processador em questão. GetS e GetM são emitidas ao executar leituras e escritas, respectivamente. Rótulos que iniciam com *other-* indicam que a operação é executada por outro processador. O rótulo PutM representa a requisição disparada pela evicção de um bloco da cache.

Para facilitar o entendimento do protocolo MESI apresentaremos um exemplo a seguir, simplificado ao considerar somente um bloco da cache. Inicialmente, as FSMs privativas de todos os processadores e a FSM do diretório estão no estado I, isto significa que este bloco da cache não contém dados válidos em nenhum processador. Ao executar uma leitura em um endereço mapeado para o bloco em questão, o processador 1 emite uma requisição GetS ao diretório. O diretório, ao receber a requisição e como estava no estado I, transiciona para o estado EorM e permite o processador 1 transicionar para o estado E. O estado E indica que somente o processador 1 tem uma cópia deste bloco, somente ele pode ler deste bloco, mas não pode escrever. Assim que outro processador, o processador 2, também executa uma leitura, ele emite outro GetS. Tanto o processador 1, quanto o 2, transicionam ao estado S. Se o processador 1 executar uma escrita, emitindo um GetM, o diretório transicionará para EorM novamente, o processador 2 transicionará para o estado I, invalidando o seu bloco, e o processador 1 transicionará ao estado M.



(a) Controlador de cache privativa.



(b) Controlador do diretório.

Figura 2 – FSMs especificando o protocolo MESI.

2.3 FLUXO TÍPICO DE GERAÇÃO DE TESTES

A Figura 3 ilustra os principais componentes de um fluxo de geração de testes típico. O componente denominado *Multicore chip design representation* corresponde ao projeto sob verificação a ser simulado. Há dois componentes acoplados à representação do projeto: *Coverage analyzer* e *Checker*. O primeiro é responsável por monitorar a representação de projeto de modo a emitir uma estimativa de cobertura. Ele analisa as FSMs responsáveis por implementar o protocolo de coerência e registra quais transições foram cobertas durante a execução de um teste. O segundo é responsável por monitorar os eventos de memória (leituras e escritas) de modo a detectar erros de coerência e consistência. Ele emite um diagnóstico sobre o comportamento observado durante a simulação, indicando se este está de acordo ou não com o comportamento esperado (tipicamente especificado por um modelo de consistência de memória). O componente denominado *Test Generator* representa o gerador de testes utilizado para criar o programa de teste que será executado no simulador associado à representação de projeto. Através da análise de cobertura, o engenheiro de verificação pode escolher manualmente novos parâmetros para o gerador de modo a exercitar partes do projeto não cobertas. Tipicamente, um gerador escolhe aleatoriamente operações (*load* e *store*) e endereços efetivos (a partir de um espaço de endereçamento pré-definido). Por isso, o fluxo ilustrado na Figura 3 corresponde à categoria de geradores de testes aleatórios (*Random Test Generators*). Dois dos geradores abordados neste trabalho (CRTG e IRTG) pertencem a esta categoria.

Por outro lado, o fluxo pode ser alterado para conter um laço de realimentação, através da inclusão de um componente Diretor (como, por exemplo, um algoritmo genético), que recebe uma estimativa de cobertura e decide automaticamente a partir dela quais os novos parâmetros a serem utilizados pelo *Test Generator*. Essa realimentação dá origem à categoria de geradores de testes dirigidos (*Directed Test Generators*) (FINE; ZIV, 2003; WAGNER; BERTACCO, 2008; ELVER; NAGARAJAN, 2016). Um dos geradores abordado neste trabalho (MTG) pertence a esta categoria. Nele, um teste é visto como um cromossomo constituído por um número constante de genes (operações) (ELVER; NAGARAJAN, 2016). Assim, os genes podem sofrer mutações e os cromossomos podem ser submetidos a cruzamentos para produzirem novos testes. A partir de uma população de testes gerados aleatoriamente, novos testes são obtidos via cruzamento e mutação. Assim, a geração de testes é dirigida por um algoritmo genético (HOLLAND, 1992).

No próximo capítulo, vários trabalhos correlatos serão analisados à luz do fluxo da Figura 3.

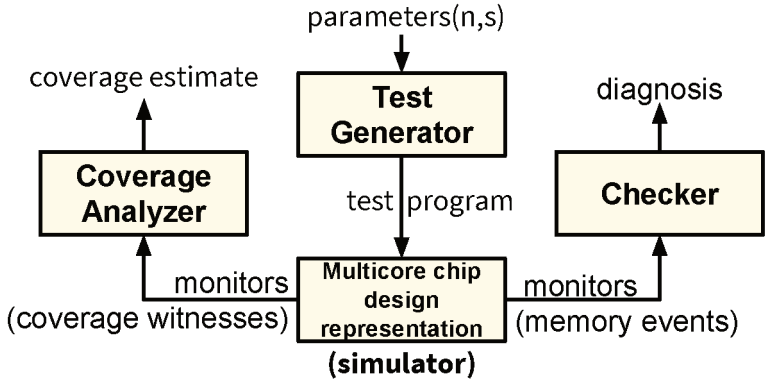


Figura 3 Os principais componentes de um fluxo de geração de testes típico.

3 TRABALHOS CORRELATOS

Neste capítulo serão discutidas algumas técnicas de geração apresentadas na literatura, iniciando por técnicas de geração pseudo-aleatória, seu uso e suas limitações. Em seguida serão abordadas quatro técnicas de geração. As duas primeiras utilizam métricas de cobertura externas às técnicas propostas, pré-definidas no ambiente de verificação. A primeira é uma técnica que utiliza uma métrica externa para selecionar diretivas que guiam a geração de teste, esta seleção pode ser manualmente feita ou através de um mecanismo automatizado, como redes Bayesianas. Outro método que utiliza uma métrica externa faz uso de geração de testes baseada em algoritmo genético e a métrica externa pode ser usada como função de *fitness*, efetivamente guiando a geração de testes. As outras duas técnicas abordadas utilizam métricas de cobertura internas, incluídas nas técnicas propostas. A primeira divide a máquina produto do protocolo de coerência em estruturas regulares de forma que elas possam ser percorridas eficientemente por um caminho euleriano. A segunda utiliza uma rede de agentes cooperantes para gerar as operações necessárias para se obter alta cobertura.

3.1 GERAÇÃO DE TESTES PSEUDO-ALEATÓRIOS

Geradores pseudo-aleatórios utilizam sequências de instruções que geram condições de corrida para expor erros de projeto. Estas são geradas pseudo-aleatoriamente com o intuito de explorar não determinismo, pois isto aumenta a probabilidade de detecção de erros (MANOVIT; HANGAL, 2006). Estes geradores são geralmente utilizados em cenários de verificação pós-fabricação, ou pós-silício (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; ROY et al., 2006; HU et al., 2012), onde um protótipo físico do processador já foi criado e o teste é nele executado diretamente. Ao contrário da verificação pré-silício, onde a execução é feita em simulador, o teste pós-silício é ordens de magnitude mais rápido e, por isso, utiliza programas com milhões de operações.

Alguns parâmetros utilizados para geração são o número de operações de leitura e escrita em memória, a frequência relativa de tipos de instrução e a quantidade de endereços compartilhados (HANGAL et al., 2004). Estes são usados pelo engenheiro de verificação para controlar o teste e podem ser explorados para aumentar a probabilidade de detecção, apesar de sua escolha não ser trivial.

3.2 GERAÇÃO DE TESTES DIRIGIDOS

Realizar a verificação na etapa pré-silício do projeto é muito vantajoso, pois a correção de erros de projeto nesta etapa tem custo muito menor. Entretanto, como a execução em simulação é ordens de magnitude mais lenta que em protótipo, não se pode usar os mesmos métodos de geração pseudo-aleatória na etapa pré-silício. Por esta razão, buscou-se maneiras de aumentar a eficiência dos testes, utilizando testes mais curtos mas com maior poder de detecção. Uma maneira utilizada para melhorar a qualidade da geração de testes foi a utilização de geradores dirigidos. Geradores dirigidos não se utilizam somente de não determinismo para expor erros de projeto, mas exploram um modelo de cobertura para guiar a geração. Alguns geradores de testes dirigidos exploram modelos de cobertura pré-definidos no ambiente de verificação (ADIR et al., 2004; ELVER; NAGARAJAN, 2016), outros exploram seus próprios modelos de cobertura (WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012). Utilizando o modelo criado, pode-se gerar as instruções necessárias para se exercitar uma parte específica do sistema, a qual não havia sido coberta anteriormente.

3.2.1 Métrica de cobertura externa

Genesys-Pro (ADIR et al., 2004) é um gerador de testes modular que traduz o problema de geração de testes em um problema de satisfação de restrições (*Constraint Satisfaction Problem* - CSP), e utiliza um solucionador genérico de CSP personalizado para geração de testes pseudo-aleatórios a fim de aumentar a qualidade dos testes gerados.

Um aspecto importante do Genesys-Pro é a sua modularidade. Ele é divido em três módulos distintos: o motor de geração, o modelo arquitetural e os *templates* de teste. O motor de geração possui conhecimento genérico sobre arquiteturas de projeto e técnicas de geração de testes. O modelo arquitetural possui informações específicas do processador-alvo da verificação, como uma descrição declarativa da arquitetura do processador-alvo e conhecimento de teste relevante para este processador. O conhecimento de teste pode ser utilizado para criar testes de alta qualidade. Por último, os *templates* de teste descrevem cenários específicos que serão utilizados para atingir os objetivos do plano de verificação. Utilizando esta abordagem, o método é capaz de separar o componente de geração, que é independente de qualquer definição do processador-alvo, dos detalhes específicos do projeto em

verificação, possibilitando a reutilização das partes que são independentes do projeto sendo verificado.

O motor de geração produz testes a partir dos cenários de verificação especificados nos *templates* limitado pelas restrições. As restrições são utilizadas para influenciar a geração aleatória em direção a áreas de interesse. Restrições podem ser obtidas de dois modos: no conhecimento de domínio específico do processador, e no conhecimento genérico de teste, que contém restrições gerais que podem ser aplicadas a qualquer processador. Alguns exemplos de restrições são: alinhamento (alinhamento dos endereços escolhidos), cache (mapeamento em cache dos endereços escolhidos) e dependência de recursos (registradores utilizados por instruções). A partir de um *template* é possível gerar testes com diferentes focos ao se aplicar diferentes restrições.

Os *templates* de teste possuem diretivas que são usadas pelos usuários para guiar o processo de geração a fim de exercitar partes do sistema ainda não cobertas. O processo de escolha de diretivas pode ser manualmente feito através de análise de relatórios de cobertura, ou de forma automatizada. Fine e Ziv (2003) reportam o uso de redes Bayesianas para dirigir a geração de novos testes a partir dos resultados de testes já executados.

Outro método, McVerSi (ELVER; NAGARAJAN, 2016), utiliza uma abordagem baseada em algoritmo genético para a geração de testes. Cada teste é associado a um cromossomo e cada operação, a um gene. Um cromossomo é representado por um grafo direcionado acíclico das operações escolhidas para cada núcleo. Cada gene é representado por uma instrução da *instruction set architecture* (ISA) do processador-alvo.

O cruzamento (*crossover*) é realizado através de uma função de cruzamento seletivo, que prioriza operações de memória que contribuem mais para o não determinismo, em busca de aumentar a probabilidade de detecção de erros. O não determinismo de uma operação é calculado a partir da quantidade de operações que ocorrem antes de dado evento e estão em conflito[1] com ele, isto favorece operações que têm maior probabilidade de causar condição de corrida. A fim de guiar o sistema a exercitar o máximo da lógica do protocolo possível, a cobertura estrutural (de código) do protocolo de coerência é utilizada como função-objetivo (*fitness*) do algoritmo genético, porém, ela é computada de forma a desconsiderar transições muito frequentes, valorizando transições pouco exercitadas. Portanto, McVerSi gera testes que ten-

[1]Duas operações são conflitantes se referenciam a mesma posição de memória e ao menos uma delas é uma escrita.

dem a exercitar mais partes do protocolo de coerência e, ao mesmo tempo, produz testes que contêm mais operações não determinísticas, que auxiliam na detecção de erros e percorrimento de *corner cases* da máquina de estados do protocolo.

Esta técnica, porém, requer que o usuário escolha alguns parâmetros de geração como quantidade de instruções por teste (n) e a quantidade de memória utilizada em um teste (TM). O autor avaliou o McVerSi utilizando valores de $n = 1k$ instruções por teste e $TM = \{1KB, 8KB\}$ e reportou os resultados para estes valores, entretanto, a obrigatoriedade de definição estática de alguns parâmetros pode levar um usuário a, inadvertidamente, limitar o potencial de detecção de erros da técnica ao selecionar valores inapropriados.

3.2.2 Métrica própria de cobertura

Qin e Mishra (2012) apresentam um método de geração dirigida de testes que permite obter cobertura total de estados e transições de uma grande variedade de protocolos de coerência. A motivação desta estratégia veio do método de busca em largura (*breadth-first search* - BFS) do espaço de estados da máquina produto, a combinação das FSMs do protocolo de coerência de cada núcleo. É possível utilizar BFS para atingir cobertura completa da máquina produto, porém, esta solução apresenta dois problemas: as transições próximas do estado inicial são visitadas múltiplas vezes, desperdiçando esforço sem reflexo na cobertura; e é necessário guardar quais estados já foram visitados, fazendo com que a quantidade de memória necessária para a execução cresça exponencialmente com o número de estados.

Para lidar com os problemas presentes no método BFS, Qin e Mishra (2012) buscaram dividir o espaço de estados em estruturas regulares, de forma que possam ser percorridas eficientemente. Estruturas como hipercubos e cliques podem ser percorridas visitando cada transição somente uma vez. Esta solução possibilitou a criação de um algoritmo dinâmico de geração de testes que, ao percorrer os estados através de um caminho euleriano, necessita de espaço de memória proporcional ao número de processadores. O algoritmo proposto gera as instruções necessárias para percorrer a transição desejada através da execução de instruções de leitura, escrita, ou evicção (através de leitura de um endereço de memória diferente mas que é mapeado para o mesmo bloco da cache), podendo obter 100% de cobertura de estados e transições. Para se manter a ordem de execução de instruções en-

tre diferentes núcleos, característica necessária para o funcionamento do método, foi usado o par de instruções da arquitetura ALPHA *load-linked* e *store-conditional*. Esta técnica gerou testes 50% a 60% mais curtos que testes gerados diretamente por BFS para obter cobertura completa.

Apesar de Qin e Mishra (2012) reportar a aplicação de sua solução para diversos protocolos de coerência, como MSI, MESI, MOSI e MOESI, ela não poderia ser diretamente aplicada a outros protocolos. O uso desta técnica em um novo protocolo de coerência necessitaria de uma nova análise do protocolo para dividir o espaço de busca de uma maneira que possa ser percorrida por um caminho euleriano. Isto torna a técnica de difícil aplicação a novos protocolos, pela necessidade de adaptação.

Outra técnica, McJammer (WAGNER; BERTACCO, 2008), de acordo com os autores, é uma ferramenta de verificação adaptativa para projetos de CMP que utiliza *feedback* em circuito fechado de maneira a ajustar dinamicamente a simulação para testar efetivamente *corner cases* do comportamento do projeto. McJammer utiliza múltiplos agentes cooperativos, cada um associado a um núcleo do CMP-alvo, para obter alta cobertura em pouco tempo. Cada agente contém um modelo simplificado do protocolo de coerência global. Duas métricas são utilizadas para guiar a geração: cobertura das máquinas simplificadas de cada agente; e uma medida denominada pressão. Pressão é calculada como o tempo médio entre eventos de memória conflitantes e ela é utilizada para medir o "estresse" no sistema de memória. Se o valor de pressão estiver baixo demais, o gerador pode diminuir o tempo entre ações iniciadas pelos agentes para que mais conflitos ocorram. Em cada rodada, agentes podem buscar cumprir os seus objetivos, aumentar a cobertura no seu modelo local, ou cooperar com outros agentes para atingir os objetivos destes agentes. A probabilidade de um agente auxiliar outros agentes a cumprir seus objetivos é inversamente proporcional à quantidade de objetivos próprios cumpridos pelo agente, assim, agentes que já obtiveram muitos de seus objetivos completos tendem a auxiliar outros agentes a cumprir seus objetivos.

O modelo simplificado do sistema, do qual o agente busca maximizar a cobertura, é denominado máquina de estados dicotômica (*Dichotomic Finite-State Machine* - DFSM). A DFSM une o estado do agente com o do ambiente, simplificando a visão do agente. A DFSM de um dado agente corresponde à FSM produto do agente em questão com a de outro agente. Como cada agente tem uma DFSM única, o sistema todo abrange todas as combinações de estados de agentes dois

a dois. Cada transição desta máquina possui um vetor de cobertura (de tamanho igual à quantidade de núcleos no projeto); o valor salvo na posição i deste vetor indica a quantidade de vezes que dada transição foi percorrida com a ajuda do agente número i. Como para cada transição da DFSM, é necessário um vetor de tamanho p (quantidade de processadores no projeto), ela cresce de maneira linear com o número de processadores no projeto. Como a combinação dois a dois é uma simplificação da máquina produto total, este método requer que cada agente exercite uma transição múltiplas vezes até ela ser considerada coberta. McJammer foi capaz de atingir coberturas mais altas e com menos esforço do que um gerador pseudo-aleatório.

Assim como a técnica anterior, McJammer requer alterações para ser utilizado com outros protocolos de coerência. Um redesenho da DFSM é necessário se esta ténica for ser utilizada com outros protocolos de coerência, isto dificulta o seu reuso.

4 COMPARAÇÃO QUALITATIVA DOS GERADORES AVALIADOS

Para cada um dos geradores sob avaliação, este capítulo descreve seus parâmetros de controle, seu princípio de funcionamento e as propriedades gerais dos testes gerados por eles.

4.1 OS GERADORES E SEUS PARÂMETROS DE CONTROLE COMUNS

Os três geradores avaliados são: *McVerSi Test Generator* (MTG), *Conventional Random Test Generator* (CRTG) e *Improved Random Test Generator* (IRTG).

Os geradores MTG e CRTG são baseados no trabalho de Elver e Nagarajan (2016) e seus códigos foram adaptados do repositório fornecido pelo autor (ELVER, 2016). O MTG utiliza o método denominado McVerSi-All. O CRTG utiliza o método denominado McVerSi-RAND. O gerador IRTG é baseado no trabalho de Andrade (2017).

Alguns parâmetros comuns aos geradores são:

- n : quantidade de instruções total do teste gerado;
- p : quantidade de núcleos da arquitetura-alvo;
- s : quantidade de endereços compartilhados distintos[1];
- mix : proporção entre instruções de leitura e de escrita desejada;
- *seed* : semente aleatória.

4.2 PRINCÍPIO DE FUNCIONAMENTO DE CADA GERADOR E PROPRIEDADES DOS TESTES GERADOS

A Tabela 1 resume as principais diferenças e semelhanças entre os geradores avaliados.

Primeiramente, vamos discutir os diferentes princípios de funcionamento para dois principais aspectos diferentes de geração: a seleção

[1]Na verdade, o parâmetro de geração dos geradores MTG e CRTG relacionado à quantidade de endereços compartilhados distintos é o *test memory* (TM), mas ambos possuem semânticas semelhantes.

de operações de memória e a seleção de endereços efetivos para as variáveis compartilhadas.

Inicialmente, os testes do MTG são gerados aleatoriamente, porém, após terem sido executados, eles são combinados utilizando uma função de cruzamento, a função utilizada é chamada de cruzamento seletivo. O objetivo desta função é produzir filhos a partir de dois pais de forma a criar indivíduos mais aptos que os pais. Ela faz isto procurando por características interessantes nos pais que podem gerar testes melhores. A característica buscada pelo cruzamento seletivo é o não determinismo. Quanto mais um evento contribui para o não determinismo, maior é a probabilidade dele ser propagado para os filhos do teste. Esta função tende a aumentar o não determinismo dos testes a medida que o conjunto de testes é executado. Desta maneira, a seleção de operações de memória do MTG é feita através da função de cruzamento seletivo.

Em contraste, o CRTG utiliza somente geração aleatória pura. O próximo teste a ser executado não utiliza nenhuma informação obtida na execução dos testes passados para o melhorar, ele é simplesmente gerado aleatoriamente. Assim, a seleção de operações de memória do CRTG é feita através de escolha aleatória.

O gerador IRTG, entretanto, possui um outro mecanismo para selecionar as operações. A escolha das instruções se dá ainda de forma aleatória, porém esta é restringida. A geração de instruções é feita através da criação de cadeias de dependência entre processadores. De modo geral, uma cadeia de dependência é uma sequência de instruções que pode gerar dependência de dados através da memória, seja isso no escopo de um processador ou de múltiplos. A ideia por trás deste método é de que somente sequências de instruções que são necessárias para manter a semântica do modelo de memória podem expor erros. Andrade, Graf e Santos (2016) se basearam no conceito de utilizar cadeias de dependência para definir um modelo de memória (GHARACHORLOO, 1995), buscando diminuir a quantidade de instruções que não contribuem para a detecção de erros. Assim, a seleção de operações de memória do gerador IRTG se dá de forma aleatória, mas com restrição de encadeamento entre *threads*.

A respeito da seleção de endereços, o MTG a faz de maneira aleatória, porém, a seleção é restrita pelo espaço de endereçamento. O espaço de endereçamento é definido pelo usuário no início da execução de um conjunto de testes e o seu valor permanece inalterado até o fim da execução. O espaço de endereçamento contém a memória utilizada em um teste. O MTG divide o espaço de endereçamento em partições

contínuas de 512B. A distância entre o endereço inicial de uma partição e o da próxima é de 1MB. Elver e Nagarajan (2016) utilizam duas configurações de espaço de endereçamento: 2MB e 16MB, estes valores representam testes que utilizam 1KB e 8KB de memória, respectivamente.

A seleção de endereços no CRTG é feita da mesma maneira que no MTG: aleatória sob restrição do espaço de endereçamento.

Apesar da seleção de endereços no IRTG também ser aleatória, ela é restrita por um mecanismo diferentes dos geradores anteriores. A competição dos endereços gerados pelos blocos de cache é restrita. A restrição se dá através da escolha dos seus valores de forma a obedecerem dois novos parâmetros de geração: κ e χ. κ determina a quantidade total de blocos de cache que os endereços gerados competirão. Assim, a escolha dos endereços se dá de forma que todos os endereços gerados sejam mapeados para somente κ blocos de cache. χ determina o máximo número de endereços que serão mapeados para cada bloco de cache. A técnica garante que pelo menos uma linha de cache terá χ endereços mapeados a ela. Este mecanismo é descrito em Andrade, Graf e Santos (2016). O valor de χ é fixo em $\frac{s}{\kappa}$ para se obter um mapeamento uniforme.

Agora vamos abordar as propriedades dos programas de teste gerados a partir dos geradores avaliados. Cinco propriedades serão avaliadas: quantidade de operações (n), quantidade de variáveis compartilhadas (s), mix de operações, quantidade de *threads* (p) e quantidade de operações em cada uma das threads.

A quantidade de operações dos testes gerados por ambos MTG e CRTG é um parâmetro e seu valor é escolhido antes de iniciar a execução do conjunto de testes e não se altera durante a execução de um conjunto de testes.

A quantidade de operações nos testes gerados pelo IRTG também não se altera, contudo, ela pode ser modificada para cada teste individualmente. Apesar de este valor se manter fixo nos testes executados neste trabalho, seu valor poderia ser escolhido dinamicamente por um agente diretor a fim de buscar aumentar a eficiência dos testes gerados.

A quantidade de variáveis compartilhadas utilizadas nos testes gerados pelo CRTG e nos testes aleatórios (antes de iniciar o cruzamento) do MTG é variável, pois elas são aleatoriamente escolhidas. Ela pode variar de 1 (caso todas as escolhas aleatórias selecionem a mesma variável) ao máximo $\frac{TM}{stride}$ (caso cada uma delas selecione um endereço distinto). O *stride* é um indicador da distância entre dois endereço com-

partilhados. A distância entre dois endereços compartilhados deve ser *stride* ou um múltiplo inteiro positivo de *stride*. O MTG diverge desse comportamento assim que cruzamentos começam a ocorrer. Apesar da quantidade de variáveis compartilhadas ainda ser variável, podendo ser única para cada teste, o cruzamento seletivo assume um grande papel na escolha do valor de variáveis compartilhadas de testes gerados pelo MTG. O valor determinado para cada teste, porém, dependerá do não determinismo proveniente das interações entre os eventos de memória nos testes.

A quantidade de variáveis compartilhadas para o gerador IRTG é determinada pelo parâmetro s. Esse parâmetro é fixo para cada teste, mas diferentes valores podem ser explorados em um conjunto de testes.

Todos os três geradores avaliados mantêm o mix de operações fixo. Entretanto, somente o MTG e o CRTG necessitam que o valor seja o mesmo em todos os testes do conjunto. O IRTG pode modificar o mix para cada teste, mas a implementação utilizada neste trabalho somente utilizou um valor fixo por todos os testes.

Os três geradores avaliados mantêm fixa a quantidade de *threads* e consideram fixa a alocação de uma *thread* por núcleo do processador-alvo, a quantidade é escolhida através do parâmetro p.

Para os geradores MTG e CRTG, a quantidade de instruções que cada *thread* executa em um teste, o tamanho da *thread*, é aleatoriamente gerado durante o teste. Cada *thread* tem a mesma probabilidade de receber uma instrução, sendo assim, a média do tamanho das *threads* tende a $\frac{n}{p}$, mas possibilita tamanhos maiores e menores que este valor. Esta flexibilidade nos tamanhos é benéfico ao gerador, pois possibilita que sejam exploradas *threads* de diversos tamanhos em um mesmo teste, não limitando ao gerador somente tamanhos de $\frac{n}{p}$. Esta característica é especialmente desejada quando podemos ter tamanhos de *thread* maiores que a média. Tamanhos maiores podem aumentar a probabilidade de detecção, pois tais testes contém *threads* de tamanho somente obtido com valores maiores de n, aumento este que aumenta a probabilidade de detecção de erros.

A implementação do IRTG utilizada neste trabalho gera sempre testes com um tamanho fixo de *thread* $\frac{n}{p}$. Desta forma, as instruções são igualmente distribuídas entre os núcleos em verificação.

Tabela 1 – Diferenças e semelhanças de cada gerador avaliado.

	Aspecto	**CRTG**	**IRTG**	**MTG**
Princípio de funcionamento	Seleção de operações de memória	Aleatória	Aleatória sob restrições	Cruzamento seletivo
	Seleção de endereços efetivos	Aleatória sob restrição estática	Aleatória sob restrição dinâmica	Aleatória sob restrição estática
Propriedades dos testes gerados	Qt. operações (n)	Fixa para conjunto de testes	Fixa para cada teste	Fixa para conjunto de testes
	Qt. variáveis compartilhadas (s)	Variável para cada teste	Fixa para cada teste	Variável para cada teste
	Mix de operações	Fixo para conjunto de testes	Fixo para cada teste	Fixo para conjunto de testes
	Qt. *threads*	Fixa (p)	Fixa (p)	Fixa (p)
	Qt. operações em cada *thread*	Variável	Fixa ($\frac{n}{p}$)	Variável

5 AVALIAÇÃO EXPERIMENTAL

Nesta seção será descrita a infraestrutura utilizada para a realização de experimentos. O simulador e o *checker* serão descritos, assim como os erros inseridos na representação de projeto, necessários para a medição de esforço.

5.1 INFRAESTRUTURA EXPERIMENTAL

Os experimentos utilizaram o simulador Gem5 (BINKERT et al., 2011) instrumentado com o modelo de processador com execução fora de ordem, com o modelo de memória denominado *Ruby* e com o modelo de rede de interconexão denominado *simple*. Foi utilizada a implementação do protocolo MESI baseado em diretório, com 3 níveis com caches L0(4KB com mapeamento direto) privativa, L1(64KB com associatividade 2-way) privativa e L2(2MB com associatividade 8-way) compartilhada. O tamanho do bloco é 64 bytes em todos os níveis.

Erros foram inseridos em representações de projeto corretas para avaliar a habilidade de cada gerador em expor anomalias que o *checker* seja capaz de encontrar. Para implementar um erro, alterações foram feitas na máquina de estados dos controladores de cache, através de modificações do próximo estado de uma transição ou de supressão de uma ação de saída associada a ela. Cada gerador foi usado para sintetizar programas paralelos não sincronizados que servem de teste para verifica a representação de projeto via simulação. Um *checker* é acoplado ao simulador, ele tem a função de observar alguns pontos da representação de projeto e determinar se o modelo de consistência de memória escolhido está sendo seguido. Se uma infração das regras do MCM é detectada, a simulação é interrompida e o tempo até a detecção, anotado.

A avaliação experimental requer a inserção de erros no projeto do processador. Os erros selecionados neste trabalho estão descritos na Tabela 2. Os nomes de estados, transições e sinais adotam a terminologia utilizada no Gem5 para reprodutibilidade.

A implementação do *checker* utilizada foi baseada no pseudocódigo descrito em (FREITAS; RAMBO; SANTOS, 2013). Este é um *checker on-the-fly*, isto é, ele não necessita que a execução do teste tenha completado para avaliar se o MCM foi seguido. Isto possibilita a interrupção do teste assim que uma infração for detectada, não executando

Tabela 2 – Erros de projeto selecionados.

ID	Estado	Evento de entrada	Próximo estado	Ação de saída impedida
E1 (L1)	E_ILO	L0_DataAck	MM	writeDataFromL0Response
E2 (L1)	E	WriteBack	MM	writeDataFromL0Request
E3 (L1)	IS_I	DataS_fromL1	I	writeDataFromL2Response
E4 (L1)	IS_I	Data_all_Acks	I	writeDataFromL2Response
E5 (L1)	E_IL0	WriteBack	MM_IL0	writeDataFromL0Request

instruções desnecessárias. Este *checker*, ao utilizar três pontos de monitoramento por núcleo, garante a não ocorrência de falsos-positivos e falsos-negativos, tornando-o próprio para a avaliação necessária neste trabalho.

5.2 CONDIÇÕES EXPERIMENTAIS

Neste seção serão descritos os valores de parâmetros utilizados nos geradores e a razão de escolha destes valores.

Todos os geradores foram testados utilizando tamanho de teste $n \in \{1024, 2048, 4096\}$. O valor 1024 é utilizado por Elver e Nagarajan (2016), os valores de 2048 e 4096 foram escolhidos com o intuito de investigar se o aumento de n melhora a detecção de erros, ou se o aumento no tempo de execução do teste acaba diminuindo a eficiência dos testes.

A quantidade máxima de variáveis compartilhadas distintas foi $s = 128$ para os geradores MTG e CRTG, pois esse é o valor utilizado pelos autores. Este valor foi calculado a partir do tamanho de memória de teste de 8KB dividido pelo *stride* de 64B. Este valor de *stride* foi escolhido para alinhar o endereço compartilhado com o tamanho do bloco de cache. Como o algoritmo genético, ao fazer o cruzamento, pode variar o seu valor de s ao combinar as cadeias de instruções com maior grau de não determinismo, o conjunto de valores de s para o gerador IRTG varia do valor mínimo 4 até o máximo de 128, para uma comparação mais adequada com o MTG.

Os valores do parâmetro κ no gerador IRTG dependem do s escolhido para o teste. Os valores válidos de κ seguem a seguinte fórmula $\kappa : (1 \leq \kappa \leq s) \wedge (s \mod \kappa = 0)$. Por exemplo, para $s = 128$, os valores de κ são $\{1, 2, 4, 8, 16, 32, 64, 128\}$, pois estes são os valores que dividem 128. A escolha do valor de κ foi feita para aumentar o controle sobre eventos de evicção de blocos em cache. Para uma cache *n-way*,

quando $\chi = \frac{s}{\kappa} > n$ há potencial para evicção de um bloco; quando $\chi = \frac{s}{\kappa} \leq n$ a evicção é impossível. Assim, o controle do valor de χ permite estimular transições distintas, o que beneficia a cobertura.

O espaço de geração, isto é, o conjunto de (s, κ) válidos utilizados pelo IRTG, foi percorrido de forma aleatória.

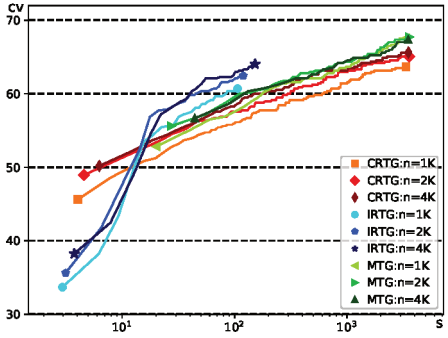
5.3 RESULTADOS EXPERIMENTAIS: COBERTURA

A Figura 4 mostra os gráficos de cobertura estrutural no tempo, isto é, da cobertura do código de implementação da máquina de estados do protocolo de coerência de cache. A quantidade de variáveis compartilhadas s máxima foi de 128 e o tempo máximo de simulação utilizado foi uma hora. A Figura 4a contém os valores para projetos com 8 núcleos, a Figura 4b, para 16 núcleos e a Figura 4c, para 32 núcleos.

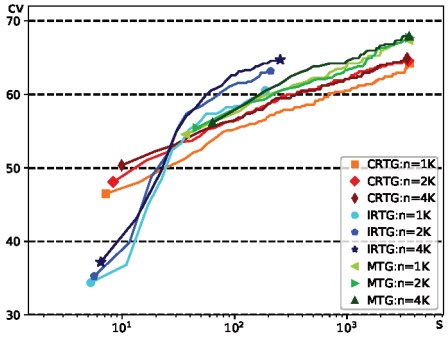
Note-se que, qualquer que seja a quantidade de núcleos, o gerador IRTG inicia com uma cobertura menor que ambos CRTG e MTG, porém, o primeiro logo os ultrapassa, ficando acima destes até o fim de sua execução. As curvas do gerador IRTG sempre terminam acima das curvas dos outros dois geradores, isto indica que a cobertura alcançada pelo IRTG é maior que a cobertura alcançada pelos outros geradores no mesmo tempo. Entretanto, ao final da execução, a cobertura total obtida por ambos CRTG e MTG é maior em alguns cenários. Para 8 núcleos, a cobertura do IRTG foi menor que a dos outros dois geradores. Para 16 núcleos, a cobertura do IRTG foi similar à cobertura do CRTG, mas inferior à do MTG. Para 32 núcleos, a cobertura máxima obtida pelo IRTG foi acima da do CRTG e similar à do MTG. Com o aumento do número de núcleos, a cobertura máxima obtida pelo IRTG aumentou, enquanto a cobertura máxima obtida pelos outros geradores não foi grandemente alterada. Este fenômeno está ligado ao aumento da probabilidade de interação entre as múltiplas *threads* de um teste quando mais núcleos são utilizados. Esta interação acontece através de operações conflitantes. O IRTG tem mais facilidade de tirar proveito desta propriedade por causa do seu uso de cadeias de dependência.

Apesar de os geradores obterem uma cobertura máxima semelhante, em torno de 67%, o gerador IRTG atinge este valor 7 vezes mais rápido. Isto indica que testes dirigidos, mesmo que de forma rudimentar, ao estimular conflitos em variáveis compartilhadas e controlar a evicção de blocos em cache, tendem a atingir mais rapidamente a mesma cobertura que múltiplos testes aleatórios ou que o uso de algo-

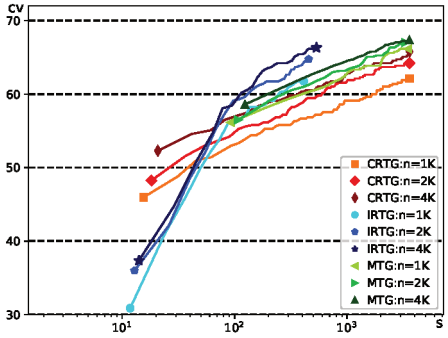
ritmo genético que busca aumentar o não determinismo dos testes.



(a) 8 núcleos.



(b) 16 núcleos.



(c) 32 núcleos.

Figura 4 Evolução da cobertura estrutural no tempo.

5.4 RESULTADOS EXPERIMENTAIS: ESFORÇO

A Tabela 3 mostra o esforço (medido em segundos) para encontrar cada erro de projeto com testes sintetizados pelos geradores selecionados. Células com o fundo preto indicam que o erro não foi detectado. Cada célula apresenta o tempo mediano de detecção em 10 execuções do gerador com os parâmetros adotados (n dado na tabela, s limitado a 128) com diferentes sementes aleatórias.

Para os erros E1 e E2, CRTG, IRTG e MTG necessitam de esforço semelhante para a detecção, apesar de o gerador IRTG ser menos impactado pela aumento do número de núcleos do que os outros geradores. No contexto do erro E3, CRTG e IRTG necessitam de esforço semelhante para a detecção de erro quando $n = 1K$. O gerador MTG necessita um esforço maior do que estes dois geradores para detectar o erro E3. O IRTG necessita de esforço significativamente menor do que o CRTG quando n é aumentado para $2K$ e $4K$. Estes dados indicam que a construção de cadeias de dependência e a restrição de competição entre endereços auxiliam na cobertura de transições que são de maior dificuldade de se estimular somente com testes aleatórios, ou até mesmo com o uso de algoritmos genéticos com foco em gerar testes com maior não determinismo. Este argumento se intensifica ao olhar para o erro E5, um erro de mais difícil detecção que os outros. Apesar de o IRTG não ter detectado o erro com $n = 1K$ e $p = 32$, a detecção se deu de 2 a 19 vezes mais rápida em todos os outros casos.

O aumento de n, em geral, diminuiu ou manteve o tempo de detecção para todos os geradores, quando comparado com cenários com valores de p iguais. Porém, CRTG com $n = 4k$ obteve tempos de detecção consideravelmente piores do que com $n = 2k$ para E3 (para $p = 32$) e E5. Isto acontece porque quanto mais longo é o teste, menor é a quantidade de testes que é possível executar no tempo máximo de uma hora, pois testes de maior tamanho demoram mais tempo para executar. O aumento do tempo de detecção quando o valor de n é aumentado indica que existe um limite quando aumentar o n diminui a probabilidade de detecção, em vez de aumentar.

Tabela 3 – Esforço requerido para detectar erros.

	CRTG								
	n = 1K			n = 2K			n = 4K		
Erro	8	16	32	8	16	32	8	16	32
E1	7	13	25	4	6	33	5	7	13
E2	7	26	159	4	14	50	4	22	54
E3	29	57	307	39	191	281	26	93	436
E4	65	71	200	64	67	30	33	36	23
E5	2105	1385	3600	407	970	1188	549	1314	3041
	IRTG								
	n = 1K			n = 2K			n = 4K		
Erro	8	16	32	8	16	32	8	16	32
E1	11	13	31	5	10	26	10	14	28
E2	11	25	38	8	18	32	7	14	29
E3	13	71	333	7	24	128	6	24	128
E4	87	35	88	25	17	39	29	16	28
E5	152	233	462	64	194	508	65	69	463
	MTG								
	n = 1K			n = 2K			n = 4K		
Erro	8	16	32	8	16	32	8	16	32
E1	5	14	25	6	11	19	8	12	23
E2	9	23	99	6	13	111	8	13	49
E3	50	144	1173	62	173	944	20	226	1050
E4	252	183	40	39	107	72	67	36	24
E5	619	1902	3600	472	1236	3600	274	819	2526

6 CONCLUSÃO E PERSPECTIVA

Este capítulo apresenta as conclusões deste trabalho em relação à área de pesquisa. Também é apresentado uma indicação de trabalho futuro que pode ser realizado a fim de expandir mais o conhecimento nesta área.

6.1 PRINCIPAIS CONCLUSÕES

Neste trabalho analisamos quantitativamente os geradores CRTG, um gerador aleatório típico, IRTG, um gerador aleatório que usa cadeias de dependência e restrição da competição de endereços e MTG, um gerador baseado em algoritmo genético, de acordo com seus princípios de funcionamento e propriedades dos testes gerados por eles.

Os geradores selecionados foram analisados experimentalmente nos âmbitos de cobertura estrutural do protocolo de coerência e esforço de detecção de erros.

A cobertura máxima atingida pelos geradores foi semelhante, por volta de 67%, com o gerador CRTG obtendo um máximo levemente menor. Porém, o gerador IRTG alcança esta cobertura 7 vezes mais rápido. No âmbito de detecção de erros, CRTG, IRTG e MTG detectam alguns erros em tempo similar, porém, o gerador IRTG detecta outros erros de 2 a 19 vezes mais rápido, especialmente para os erros mais difíceis.

A partir dos experimentos executados, encontramos fortes indícios de que o gerador IRTG é um bom candidato para ser utilizado como motor de um gerador dirigido de testes. Assim, os seus parâmetros, como n, s e κ podem ser selecionados de modo a exercitar partes não cobertas do sistema, de modo a obter maiores valores de cobertura.

6.2 TRABALHOS FUTUROS

Como trabalho futuro, sugerimos o uso de métodos estatísticos, como a regressão logística, para buscar uma relação entre os valores dos parâmetros de entrada e a probabilidade de detecção de erros específicos. Entendendo melhor essa relação, poderiam ser criados conjuntos de erros com características semelhantes, e eleger erros representantes de cada conjunto. Assim, novas análises poderiam utilizar represen-

tantes de conjuntos e aumentar a representatividade, pois estariam, na verdade, analisando o comportamento de detecção de toda uma classe de erros.

Outra sugestão de trabalho futuro seria a escolha da ordem de execução do espaço de geração do IRTG. Neste trabalho, os valores foram visitados em ordem aleatória, porém eles poderiam ser visitados em uma ordem específica para buscar melhorar a cobertura. Aliado a isso, a variação de outros parâmetros, como n e mix, poderia ser utilizada para aumentar a cobertura, criando um gerador que explora mais parâmetros.

REFERÊNCIAS

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

ADIR, A.; ATTIYA, H.; SHUREK, G. Information-flow models for shared memory with an application to the powerpc architecture. *IEEE Transactions on Parallel and Distributed Systems*, v. 14, n. 5, p. 502–515, May 2003. ISSN 1045-9219.

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

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, SC, Brazil, 2 2017.

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

BINKERT, N. et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 39, n. 2, p. 1–7, ago. 2011. ISSN 0163-5964. <http://doi.acm.org/10.1145/2024716.2024718>.

DEVADAS, S. Toward a coherent multicore memory model. *Computer*, v. 46, n. 10, p. 30–31, October 2013. ISSN 0018-9162.

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

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

FINE, S.; ZIV, A. Coverage directed test generation for functional verification using bayesian networks. In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*. [S.l.: s.n.], 2003. p. 286–291.

FREITAS, L. S.; RAMBO, E. A.; SANTOS, L. C. V. dos. On-the-fly Verification of Memory Consistency with Concurrent Relaxed Scoreboards. In: *Design, Automation, and Test in Europe (DATE)*. [S.l.: s.n.], 2013. p. 631–636. ISBN 978-1-4503-2153-2.

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

GHARACHORLOO, K. et al. Memory Consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 18, n. 2SI, p. 15–26, May 1990. ISSN 0163-5964.

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

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 5th. ed. [S.l.]: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728.

HILL, M. D. Multiprocessors should support simple memory-consistency models. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 31, n. 8, p. 28–34, ago. 1998. ISSN 0018-9162. <http://dx.doi.org/10.1109/2.707614>.

HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992. ISBN 0262082136.

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

INTERNATIONAL INC., C. S. *The SPARC Architecture Manual: Version 8*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN 0-13-825001-4.

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

OWENS, S.; SARKAR, S.; SEWELL, P. A better x86 memory model: X86-tso. In: *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer-Verlag, 2009. (TPHOLs '09), p. 391–407. ISBN 978-3-642-03358-2. <http://dx.doi.org/10.1007/978-3-642-03359-9_27>.

PAPAMARCOS, M. S.; PATEL, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 12, n. 3, p. 348–354, jan. 1984. ISSN 0163-5964. <http://doi.acm.org/10.1145/773453.808204>.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 0124077269, 9780124077263.

PIZIALI, A. *Coverage-Driven Verification*. Boston, MA: Springer US, 2008. 109–137 p. ISBN 978-1-4020-8026-5. <https://doi.org/10.1007/978-1-4020-8026-5_7>.

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

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 (CAV)*. [S.l.]: Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4144). p. 503–516. ISBN 978-3-540-37411-4.

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

ANEXO A – Artigo

Avaliação experimental de um gerador de testes dirigidos para a verificação de memória compartilhada em multicore chips

Nícolas Pfeifer

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

nicolas.pfeifer@grad.ufsc.br

Resumo. *No contexto de verificação de memória compartilhada em multicore chips, este trabalho propõe uma avaliação crítica da geração automática de testes dirigidos quando baseada em Programação Genética. A metodologia consiste na comparação do gerador McVerSi (que representa o estado da arte) com geradores de testes aleatórios (que representam a base de geradores dirigidos por cobertura). Dois geradores de testes aleatórios são utilizados: um deles (McVerSi_Rand) pressupõe uma restrição do espaço de endereçamento imposta estaticamente antes de disparar a geração, o outro (IRTG) admite a variação dinâmica de restrições impostas ao espaço de endereçamento. Os três geradores são comparados de acordo com duas métricas: cobertura estrutural dos controladores de cache e esforço requerido na detecção de erros de coerência de memória. Encontramos que os três geradores analisados atingem uma cobertura máxima semelhante, mas o gerador IRTG atinge esta cobertura 7 vezes mais rápido. Dentro dos erros analisados, os geradores detectam os erros mais fáceis com tempo semelhante, mas o gerador IRTG detecta os erros mais difíceis de 2 a 19 vezes mais rápido.*

1. Introdução

O aumento da frequência de relógio e a exploração do paralelismo entre instruções permitiu um crescimento exponencial do desempenho de processadores até se atingir a chamada barreira de potência por volta de 2004, que corresponde à máxima dissipação de potência térmica possível com sistemas habituais de resfriamento [Patterson and Hennessy 2013]. Desde então, fabricantes de microprocessadores construíram processadores com múltiplos núcleos de processamento em um único chip, o que ficou conhecido como *Chip Multiprocessor* (CMP), em uma tentativa de manter o crescimento de desempenho. A comunicação entre os núcleos de um CMP é feita através de um mecanismo denominado memória compartilhada, onde toda a memória é acessível por todos os processadores, possibilitando a comunicação através de escritas e leituras da memória.

Como a abstração de uma única memória possibilita que múltiplos processadores escrevam e leiam da memória simultaneamente, programadores necessitam de um modelo conceitual da semântica das operações de memória para que possam utilizar a memória compartilhada corretamente [Gharachorloo 1995]. Este modelo é chamado de modelo de consistência de memória (MCM). Ele instrui o programador sobre a ordem esperada de execução de instruções das múltiplas *threads* de um programa paralelo. O MCM determina o grau de relaxação da ordem de programa e o grau de atomicidade das escritas

[Adve and Gharachorloo 1996]. Um MCM pressupõe a existência de um protocolo para permitir a coerência de cache. Espera-se que tais protocolos continuem sendo implementados em hardware para *multicore chips* de propósito geral operando sob memória compartilhada, mesmo para centenas de núcleos [Devadas 2013].

A influência de ambas as tendências, o aumento da quantidade de núcleos em CMP e a implementação de protocolos de coerência em *hardware* corroboram no aumento da complexidade de projetos de novos processadores, tornado-os mais propensos a erros. Por isto, é cada vez mais necessário ferramentas que auxiliem projetistas a detectar erros e garantir o funcionamento correto do projeto, de acordo com as especificações. O processo de verificação é uma destas ferramentas.

No contexto de CMP, o processo de verificação consiste em executar um programa de teste paralelo em um simulador da representação do projeto de um *multicore chip* e avaliar se o comportamento observado corresponde ao esperado. Os programas de testes são gerados automaticamente e a avalização do comportamento é também automaticamente realizada. As ferramentas responsáveis por estas tarefas são chamadas de gerador e *checker*, respectivamente. Sabe-se que o uso de programas paralelos relativamente curtos, que utilizam poucas posições de memória e têm condições de corrida agressivas expõem erros mais rapidamente [Manovit and Hangal 2006]. No contexto de verificação pré-silício, e deste trabalho, a execução do teste se dá por meio de simulação do projeto do *multiprocessor chip*. Em cenários pré-silício, são necessárias ferramentas que dirijam a geração de testes para diminuir o esforço necessário para detecção, pois o tempo de execução de testes é alto, este tipo de abordagem pode ser vista em [Elver and Nagarajan 2016] e [Wagner and Bertacco 2008], por exemplo.

2. Trabalhos correlatos

Geradores pseudo-aleatórios utilizam sequências de instruções que geram condições de corrida para expor erros de projeto. Estas são geradas pseudo-aleatoriamente com o intuito de explorar não determinismo, pois isto aumenta a probabilidade de detecção de erros [Manovit and Hangal 2006]. Estes geradores são geralmente utilizados em cenários de verificação pós-fabricação, ou pós-silício [Hangal et al. 2004, Manovit and Hangal 2006, Roy et al. 2006, Hu et al. 2012], onde um protótipo físico do processador já foi criado e o teste é nele executado diretamente. Ao contrário da verificação pré-silício, onde a execução é feita em simulador, o teste pós-silício é ordens de magnitude mais rápido e, por isso, utiliza programas com milhões de operações.

Realizar a verificação na etapa pré-silício do projeto é muito vantajoso, pois a correção de erros de projeto nesta etapa tem custo muito menor. Entretanto, como a execução em simulação é ordens de magnitude mais lenta que em protótipo, não se pode usar os mesmos métodos de geração pseudo-aleatória na etapa pré-silício. Por esta razão, buscou-se maneiras de aumentar a eficiência dos testes, utilizando testes mais curtos mas com maior poder de detecção. Uma maneira utilizada para melhorar a qualidade da geração de testes foi a utilização de geradores dirigidos. Geradores dirigidos não se utilizam somente de não determinismo para expor erros de projeto, mas exploram um modelo de cobertura para guiar a geração. Alguns geradores de testes dirigidos exploram modelos de cobertura pré-definidos no ambiente de verificação [Adir et al. 2004, Elver and Nagarajan 2016], outros exploram seus próprios modelos de

cobertura [Wagner and Bertacco 2008, Qin and Mishra 2012]. Utilizando o modelo criado, pode-se gerar as instruções necessárias para se exercitar uma parte específica do sistema, a qual não havia sido coberta anteriormente.

Genesys-Pro [Adir et al. 2004] e McVerSi [Elver and Nagarajan 2016] são dois geradores de testes que utilizam um modelo de cobertura externo ao seu método de geração. Genesys-Pro é um gerador de testes modular que traduz o problema de geração de testes em um problema de satisfação de restrições (*Constraint Satisfaction Problem* - CSP), e utiliza um solucionador genérico de CSP personalizado para geração de testes pseudo-aleatórios a fim de aumentar a qualidade dos testes gerados. Os testes são gerados a partir de *templates* de teste, estes devem ser escritos manualmente, necessitando assim de muito trabalho. Outro método, McVerSi, utiliza uma abordagem baseada em algoritmo genético para a geração de testes. Cada teste é associado a um cromossomo e cada operação, a um gene. Um cromossomo é representado por um grafo direcionado acíclico das operações escolhidas para cada núcleo. Cada gene é representado por uma instrução da *instruction set architecture* (ISA) do processador-alvo. O cruzamento (*crossover*) é realizado através de uma função de cruzamento seletivo, que prioriza operações de memória que contribuem mais para o não determinismo, em busca de aumentar a probabilidade de detecção de erros. O não determinismo de uma operação é calculado a partir da quantidade de operações em conflito com esta. Esta técnica, porém, requer que o usuário escolha alguns parâmetros de geração manualmente. A obrigatoriedade de definição estática de alguns parâmetros pode levar um usuário a, inadvertidamente, limitar o potencial de detecção de erros da técnica ao selecionar valores inapropriados.

McJammer [Wagner and Bertacco 2008] e a técnica reportada em [Qin and Mishra 2012] possuem modelos de cobertura próprios. McJammer é uma ferramenta de verificação adaptativa para projetos de CMP que utiliza *feedback* em circuito fechado de maneira a ajustar dinamicamente a simulação para testar efetivamente *corner cases* do comportamento do projeto. McJammer utiliza múltiplos agentes cooperativos, cada um associado a um núcleo do CMP-alvo, para gerar a sequência de instruções executada por cada núcleo. Cada agente contém um modelo simplificado do protocolo de coerência global, a máquina de estados dicotômica. [Qin and Mishra 2012] buscaram dividir o espaço de estados do protocolo de coerência em estruturas regulares, de forma que possam ser percorridas eficientemente. Estruturas como hipercubos e cliques podem ser percorridas visitando cada transição somente uma vez. Esta solução possibilitou a criação de um algoritmo dinâmico de geração de testes que, ao percorrer os estados através de um caminho euleriano, necessita de espaço de memória proporcional ao número de processadores. Ambos McJammer e [Qin and Mishra 2012], por possuírem modelos de cobertura próprios, sofrem do problema da necessidade de adaptação. Se um usuário quiser utilizar estas técnicas com outros protocolos de coerência, necessitará realizar o trabalho de adaptação, dificultando o use destas soluções em outros contextos.

3. Comparação qualitativa dos geradores selecionados

A Tabela 1 resume as principais diferenças e semelhanças entre os geradores avaliados. Os geradores são comparados de acordo com os seguintes critérios: princípios de funcionamento, seleção de endereços efetivos, e propriedades dos testes gerados. As propriedades dos testes avaliados foram as seguintes: quantidade de operações (n), quantidade de variáveis compartilhadas (s), mix de operações, quantidade de *threads* (p) e quantidade

de operações em cada *thread*.

Os três geradores comparados neste trabalhos são: McVerSi Test Generator (MTG), Common Random Test Generator (CRTG) e Improved Random Test Generator (IRTG). Os geradores MTG e CRTG são baseados no trabalho de [Elver and Nagarajan 2016] e seus códigos foram adaptados do repositório fornecido pelo autor [Elver 2016]. O MTG utiliza o método denominado McVerSi-All. O CRTG utiliza o método denominado McVerSi-RAND. O gerador IRTG é baseado no trabalho de [Andrade 2017].

Tabela 1. Diferenças e semelhanças de cada gerador avaliado.

	Aspecto	**CRTG**	**IRTG**	**MTG**
Princípio de funcionamento	Seleção de operações de memória	Aleatória	Aleatória sob restrições	Cruzamento seletivo
	Seleção de endereços efetivos	Aleatória sob restrição estática	Aleatória sob restrição dinâmica	Aleatória sob restrição estática
Propriedades dos testes gerados	Qt. operações (n)	Fixa para conjunto de testes	Fixa para cada teste	Fixa para conjunto de testes
	Qt. variáveis compartilhadas (s)	Variável para cada teste	Fixa para cada teste	Variável para cada teste
	Mix de operações	Fixo para conjunto de testes	Fixo para cada teste	Fixo para conjunto de testes
	Qt. *threads*	Fixa (p)	Fixa (p)	Fixa (p)
	Qt. operações em cada *thread*	Variável	Fixa ($\frac{n}{p}$)	Variável

4. Infraestrutura experimental

Os experimentos utilizaram o simulador Gem5 [Binkert et al. 2011] instrumentado com o modelo de processador com execução fora de ordem, com o modelo de memória denominado *Ruby* e com o modelo de rede de interconexão denominado *simple*. Foi utilizada a implementação do protocolo MESI baseado em diretório, com 3 níveis com caches L0(4KB com mapeamento direto) privativa, L1(64KB com associatividade 2-way) privativa e L2(2MB com associatividade 8-way) compartilhada. O tamanho do bloco é 64 bytes em todos os níveis.

Erros foram inseridos em representações de projeto corretas para avaliar a habilidade de cada gerador em expor anomalias que o *checker* seja capaz de encontrar. Para implementar um erro, alterações foram feitas na máquina de estados dos controladores de cache, através de modificações do próximo estado de uma transição ou de supressão de uma ação de saída associada a ela. Cada gerador foi usado para sintetizar programas paralelos não sincronizados que servem de teste para verifica a representação de projeto via simulação. Um *checker* é acoplado ao simulador, ele tem a função de observar alguns pontos da representação de projeto e determinar se o modelo de consistência de memória escolhido está sendo seguido. Se uma infração das regras do MCM é detectada, a simulação é interrompida e o tempo até a detecção, anotado. O *checker* utilizado neste trabalho foi baseado no código-fonte descrito em [Freitas et al. 2013].

A avaliação experimental requer a inserção de erros no projeto do processador. Os erros selecionados neste trabalho estão descritos na Tabela 2. Os nomes de estados, transições e sinais adotam a terminologia utilizada no Gem5 para reprodutibilidade.

5. Condições experimentais

Neste seção serão descritos os valores de parâmetros utilizados nos geradores e a razão de escolha destes valores. Todos os geradores foram testados utilizando tamanho de teste $n \in \{1024, 2048, 4096\}$. O valor 1024 é utilizado por [Elver and Nagarajan 2016], os valores de 2048 e 4096 foram escolhidos com o intuito de investigar se o aumento de

Tabela 2. Erros de projeto selecionados.

ID	Estado	Evento de entrada	Próximo estado	Ação de saída impedida
E1 (L1)	E_IL0	L0_DataAck	MM	writeDataFromL0Response
E2 (L1)	E	WriteBack	MM	writeDataFromL0Request
E3 (L1)	IS_I	DataS_fromL1	I	writeDataFromL2Response
E4 (L1)	IS_I	Data_all_Acks	I	writeDataFromL2Response
E5 (L1)	E_IL0	WriteBack	MM_IL0	writeDataFromL0Request

n melhora a detecção de erros, ou se o aumento no tempo de execução do teste acaba diminuindo a eficiência dos testes.

A quantidade máxima de variáveis compartilhadas distintas foi $s = 128$ para os geradores MTG e CRTG, pois esse é o valor utilizado pelos autores. Este valor foi calculado a partir do tamanho de memória de teste de 8KB dividido pelo *stride* de 64B. Este valor de *stride* foi escolhido para alinhar o endereço compartilhado com o tamanho do bloco de cache. Como o algoritmo genético, ao fazer o cruzamento, pode variar o seu valor de s ao combinar as cadeias de instruções com maior grau de não determinismo, o conjunto de valores de s para o gerador IRTG varia do valor mínimo 4 até o máximo de 128, para uma comparação mais adequada com o MTG.

Os valores do parâmetro κ no gerador IRTG dependem do s escolhido para o teste. Os valores válidos de κ seguem a seguinte fórmula $\kappa : (1 \leq \kappa \leq s) \wedge (s \mod \kappa = 0)$. Por exemplo, para $s = 128$, os valores de κ são $\{1, 2, 4, 8, 16, 32, 64, 128\}$, pois estes são os valores que dividem 128. A escolha do valor de κ foi feita para aumentar o controle sobre eventos de evicção de blocos em cache. Para uma cache *n-way*, quando $\chi = \frac{s}{\kappa} > n$ há potencial para evicção de um bloco; quando $\chi = \frac{s}{\kappa} \leq n$ a evicção é impossível. Assim, o controle do valor de χ permite estimular transições distintas, o que beneficia a cobertura. O espaço de geração, isto é, o conjunto de (s, κ) válidos utilizados pelo IRTG, foi percorrido de forma aleatória.

6. Resultados experimentais: cobertura

A Figura 4 mostra os gráficos de cobertura estrutural no tempo, isto é, da cobertura do código de implementação da máquina de estados do protocolo de coerência de cache. A quantidade de variáveis compartilhadas s máxima foi de 128 e o tempo máximo de simulação utilizado foi uma hora. A Figura 1 contém os valores para projetos com 8 núcleos, a Figura 2, para 16 núcleos e a Figura 3, para 32 núcleos.

Note-se que, qualquer que seja a quantidade de núcleos, o gerador IRTG inicia com uma cobertura menor que ambos CRTG e MTG, porém, o primeiro logo os ultrapassa, ficando acima destes até o fim de sua execução. As curvas do gerador IRTG sempre terminam acima das curvas dos outros dois geradores, isto indica que a cobertura alcançada pelo IRTG é maior que a cobertura alcançada pelos outros geradores no mesmo tempo. Entretanto, ao final da execução, a cobertura total obtida por ambos CRTG e MTG é maior em alguns cenários. Para 8 núcleos, a cobertura do IRTG foi menor que a dos outros dois geradores. Para 16 núcleos, a cobertura do IRTG foi similar à cobertura do CRTG, mas inferior à do MTG. Para 32 núcleos, a cobertura máxima obtida pelo IRTG

foi acima da do CRTG e similar à do MTG. Com o aumento do número de núcleos, a cobertura máxima obtida pelo IRTG aumentou, enquanto a cobertura máxima obtida pelos outros geradores não foi grandemente alterada. Este fenômeno está ligado ao aumento da probabilidade de interação entre as múltiplas *threads* de um teste quando mais núcleos são utilizados. Esta interação acontece através de operações conflitantes. O IRTG tem mais facilidade de tirar proveito desta propriedade por causa do seu uso de cadeias de dependência.

Apesar de os geradores obterem uma cobertura máxima semelhante, em torno de 67%, o gerador IRTG atinge este valor 7 vezes mais rápido. Isto indica que testes dirigidos, mesmo que de forma rudimentar, ao estimular conflitos em variáveis compartilhadas e controlar a evicção de blocos em cache, tendem a atingir mais rapidamente a mesma cobertura que múltiplos testes aleatórios ou que o uso de algoritmo genético que busca aumentar o não determinismo dos testes.

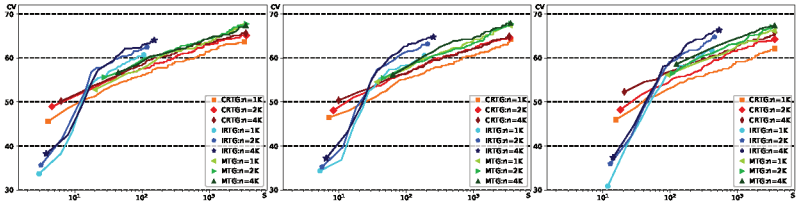


Figura 1. 8 núcleos. Figura 2. 16 núcleos. Figura 3. 32 núcleos.

Figura 4. Evolução da cobertura estrutural no tempo.

7. Resultados experimentais: esforço

A Tabela 3 mostra o esforço (medido em segundos) para encontrar cada erro de projeto com testes sintetizados pelos geradores selecionados. Células com o fundo preto indicam que o erro não foi detectado. Cada célula apresenta o tempo mediano de detecção em 10 execuções do gerador com os parâmetros adotados (n dado na tabela, s limitado a 128) com diferentes sementes aleatórias.

Para os erros E1 e E2, CRTG, IRTG e MTG necessitam de esforço semelhante para a detecção, apesar de o gerador IRTG ser menos impactado pela aumento do número de núcleos do que os outros geradores. No contexto do erro E3, CRTG e IRTG necessitam de esforço semelhante para a detecção de erro quando $n = 1K$. O gerador MTG necessita um esforço maior do que estes dois geradores para detectar o erro E3. O IRTG necessita de esforço significativamente menor do que o CRTG quando n é aumentado para $2K$ e $4K$. Estes dados indicam que a construção de cadeias de dependência e a restrição de competição entre endereços auxiliam na cobertura de transições que são de maior dificuldade de se estimular somente com testes aleatórios, ou até mesmo com o uso de algoritmos genéticos com foco em gerar testes com maior não determinismo. Este argumento se intensifica ao olhar para o erro E5, um erro de mais difícil detecção que os outros. Apesar de o IRTG não ter detectado o erro com $n = 1K$ e $p = 32$, a detecção se deu de 2 a 19 vezes mais rápida em todos os outros casos.

O aumento de n, em geral, diminuiu ou manteve o tempo de detecção para todos os geradores, quando comparado com cenários com valores de p iguais. Porém, CRTG com $n = 4k$ obteve tempos de detecção consideravelmente piores do que com $n = 2k$ para E3 (para $p = 32$) e E5. Isto acontece porque quanto mais longo é o teste, menor é a quantidade de testes que é possível executar no tempo máximo de uma hora, pois testes de maior tamanho demoram mais tempo para executar. O aumento do tempo de detecção quando o valor de n é aumentado indica que existe um limite quando aumentar o n diminui a probabilidade de detecção, em vez de aumentar.

Tabela 3. Esforço requerido para detectar erros.

	CRTG								
	n = 1K			n = 2K			n = 4K		
Erro	8	16	32	8	16	32	8	16	32
E1	7	13	25	4	6	33	5	7	13
E2	7	26	159	4	14	50	4	22	54
E3	29	57	307	39	191	281	26	93	436
E4	65	71	200	64	67	30	33	36	23
E5	2105	1385	3600	407	970	1188	549	1314	3041
	IRTG								
	n = 1K			n = 2K			n = 4K		
Erro	8	16	32	8	16	32	8	16	32
E1	11	13	31	5	10	26	10	14	28
E2	11	25	38	8	18	32	7	14	29
E3	13	71	333	7	24	128	6	24	128
E4	87	35	88	25	17	39	29	16	28
E5	152	233	462	64	194	508	65	69	463
	MTG								
	n = 1K			n = 2K			n = 4K		
Erro	8	16	32	8	16	32	8	16	32
E1	5	14	25	6	11	19	8	12	23
E2	9	23	99	6	13	111	8	13	49
E3	50	144	1173	62	173	944	20	226	1050
E4	252	183	40	39	107	72	67	36	24
E5	619	1902	3600	472	1236	3600	274	819	2526

8. Conclusões e perspectivas

Neste trabalho analisamos quantitativamente os geradores CRTG, um gerador aleatório típico, IRTG, um gerador aleatório que usa cadeias de dependência e restrição da

competição de endereços e MTG, um gerador baseado em algoritmo genético, de acordo com seus princípios de funcionamento e propriedades dos testes gerados por eles.

Os geradores selecionados foram analisados experimentalmente nos âmbitos de cobertura estrutural do protocolo de coerência e esforço de detecção de erros.

A cobertura máxima atingida pelos geradores foi semelhante, por volta de 67%, com o gerador CRTG obtendo um máximo levemente menor. Porém, o gerador IRTG alcança esta cobertura 7 vezes mais rápido. No âmbito de detecção de erros, CRTG, IRTG e MTG detectam alguns erros em tempo similar, porém, o gerador IRTG detecta outros erros de 2 a 19 vezes mais rápido, especialmente para os erros mais difíceis.

A partir dos experimentos executados, encontramos fortes indícios de que o gerador IRTG é um bom candidato para ser utilizado como motor de um gerador dirigido de testes. Assim, os seus parâmetros, como n, s e κ podem ser selecionados de modo a exercitar partes não cobertas do sistema, de modo a obter maiores valores de cobertura.

Como trabalho futuro, sugerimos o uso de métodos estatísticos, como a regressão logística, para buscar uma relação entre os valores dos parâmetros de entrada e a probabilidade de detecção de erros específicos. Entendendo melhor essa relação, poderiam ser criados conjuntos de erros com características semelhantes, e eleger erros representantes de cada conjunto. Assim, novas análises poderiam utilizar representantes de conjuntos e aumentar a representatividade, pois estariam, na verdade, analisando o comportamento de detecção de toda uma classe de erros.

Outra sugestão de trabalho futuro seria a escolha da ordem de execução do espaço de geração do IRTG. Neste trabalho, os valores foram visitados em ordem aleatória, porém eles poderiam ser visitados em uma ordem específica para buscar melhorar a cobertura. Aliado a isso, a variação de outros parâmetros, como n e mix, poderia ser utilizada para aumentar a cobertura, criando um gerador que explora mais parâmetros.

Referências

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

Adve, S. V. and Gharachorloo, K. (1996). Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76.

Andrade, G. A. G. (2017). Exploiting Canonical Dependence Chains and Address Biasing Constraints to Improve Random Test Generation for Shared-Memory Verification. Master's thesis, Universidade Federal de Santa Catarina, Florianópolis, SC, Brazil.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

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

Elver, M. (2016). Mcversi framework. `https://github.com/melver/mc2lib`.

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

Freitas, L. S., Rambo, E. A., and dos Santos, L. C. V. (2013). On-the-fly Verification of Memory Consistency with Concurrent Relaxed Scoreboards. In *Design, Automation, and Test in Europe (DATE)*, pages 631–636.

Gharachorloo, K. (1995). *Memory consistency models for shared-memory multiprocessors*. PhD thesis, Stanford University.

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

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

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

Patterson, D. A. and Hennessy, J. L. (2013). *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

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

Roy, A., Zeisset, S., Fleckenstein, C. J., and Huang, J. C. (2006). Fast and Generalized Polynomial Time Memory Consistency Verification. In Ball, T. and Jones, R. B., editors, *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 503–516. Springer Berlin Heidelberg.

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