UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Marleson Graf

GERAÇÃO DE TESTES DE MEMÓRIA COMPARTILHADA COERENTE: UMA AVALIAÇÃO DE COBERTURA E EFICÁCIA

Florianópolis

2017

Marleson Graf

GERAÇÃO DE TESTES DE MEMÓRIA COMPARTILHADA COERENTE: UMA AVALIAÇÃO DE COBERTURA E EFICÁCIA

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

2017

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

Graf, Marleson
Geração de testes de memória compartilhada coerente: uma avaliação de cobertura e eficácia / Marleson Graf ; orientador, Luiz Cláudio Villar dos Santos, 2017.
78 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2017.

Inclui referências.

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

Dedico este trabalho a minha família.

AGRADECIMENTOS

Agradeço a todos que contribuíram de alguma forma para a elaboração e execução do presente Trabalho de Conclusão de Curso, em especial ao orientador e demais membros do grupo de pesquisa, os quais estiveram diretamente envolvidos com o trabalho desenvolvido. Também agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pela concessão de bolsa de Iniciação Científica (PIBIC), cujas atividades relacionadas fomentaram o desenvolvimento desse trabalho.

Do what thou wilt shall be the whole of the Law.

Aleister Crowley

RESUMO

Multiprocessadores em chip (CMP) vêm sendo desafiados pela crescente complexidade de seus subsistemas de memória compartilhada. Regras de consistência e requisitos de coerência especificados por um modelo de memória devem ser atendidos pelo *hardware* para que o funcionamento da memória ocorra como esperado pelo programador. O uso de regras de consistência relaxadas – para abrir espaço a novas otimizações – e o crescente número de núcleos – o que aumenta a complexidade dos protocolos de coerência – torna o projeto do subsistema de memória suscetível a erros. Para evitar que esses erros se propaguem até a fase de prototipação, aplicam-se técnicas de verificação sobre uma versão executável do projeto (por meio de simulações). Essas técnicas dependem da execução de programas de teste sobre a plataforma verificada, a fim de estimular o sistema e expor quaisquer erros que possam existir no projeto. Nesse trabalho é proposta uma métrica de cobertura para avaliar os programas de testes gerados por diferentes técnicas, e assim descobrir quais propriedades e características dos geradores têm mais impacto na capacidade de exposição de erros. Com essas informações espera-se fomentar as bases de um futuro gerador adaptativo de testes. Complementarmente, foram desenvolvidos novos erros de coerência, a partir dos quais criaram-se representações de projeto para avaliar o esforço e eficácia dos testes gerados pelas diferentes técnicas. Para a representação dos projetos, foi utilizado um simulador de domínio público (gem5). Ao todo, foram executados 14400 programas de teste para a avaliação de cobertura e 86400 casos de uso para a avaliação de esforço e eficácia (6 erros $\times$ 14400 programas de teste), em arquiteturas de 8, 16 e 32 núcleos. Os resultados mostram que a combinação de duas diferentes técnicas de geração de testes leva aos melhores valores de cobertura e alcança os melhores resultados de esforço e eficácia na maiorias dos cenários de verificação.

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

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

Multiprocessadores em chip (CMP), como o próprio nome sugere, são chips que contém múltiplos núcleos de processamento em seu interior. São amplamente utilizados em computadores pessoais e servidores, onde a programação de propósito geral conta com o uso de modelos de memória para abstrair detalhes do uso da hierarquia de memória. Em termos gerais, um modelo de memória restringe quais valores a leitura de uma posição de memória compartilhada pode retornar (ADVE; GHARACHORLOO, 1996). Essas restrições são capturadas por regras de consistência, as quais abordam dois aspectos principais: o quanto a ordem de execução de operações de leitura e escrita para endereços de memória distintos pode ser relaxada, i.e., executada fora da ordem originalmente especificada, e o grau de atomicidade das operações de escrita para um mesmo endereço (ADVE; GHARACHORLOO, 1996). O *hardware* subjacente garante que o comportamento do subsistema de memória seja compatível com o modelo adotado.

A busca por maior desempenho leva ao uso de modelos de memória que permitem ordens de execução mais relaxadas. Em uma execução sequencial, a ordem em que as operações são emitidas e executadas é a exatamente a mesma definida pelo programa. Ao relaxar a ordem de execução, otimizações podem ser exploradas na emissão de operações de leitura e escrita ao subsistema de memória. Embora essa relaxação abra a possibilidade de resultados diferentes na execução de programas paralelos, o amplo uso de bibliotecas de sincronização acaba escondendo as regras de consistência da visão do programador (HENNESSY; PATTERSON, 2011), o que permite o uso de ordens mais relaxadas sem adicionar complexidade à programação.

Para suportar emissões fora de ordem, *buffers* de leitura e escrita são necessários em cada processador e entre *caches* de diferentes níveis hierárquicos. Além disso, sistemas que utilizam *caches* privativas precisam manter cópias válidas de um mesmo dado em múltiplas *caches* ao longo da hierarquia. Para tanto, adotam-se protocolos de coerência baseados em diretório os quais, apesar da complexidade, acredita-se serem escaláveis a centenas de processadores (MARTIN; HILL; SORIN, 2012). Essa complexidade, intensificadas pelo aumento do número de processadores em um único *chip*, faz com que projetos de multiprocessadores sejam suscetíveis a erros em sua especificação, tornando a aplicação de técnicas de verificação funcional fundamentais para validar o projeto sob desenvolvimento.

1.1 TÉCNICAS DE VERIFICAÇÃO

O uso de modelos de memória como base para a construção de *checkers* foi inicialmente proposto por Hangal et al. (2004) e inspirou muitos outros trabalhos em verificação pós-silício, como Manovit e Hangal (2006), Roy et al. (2006), Chen et al. (2009) e Hu et al. (2012). A grande vantagem de basear-se no modelo de memória reside em desacoplar a técnica de verificação da implementação específica sendo testada, de forma que esta possa ser reutilizada para diferentes projetos derivados de uma mesma arquitetura, um projeto completamente novo para uma arquitetura que adote o mesmo modelo (HANGAL et al., 2004) ou mesmo uma arquitetura que suporte-o entre múltiplos modelos (SHACHAM et al., 2008).

Checkers pós-silício são executados diretamente sobre um protótipo do projeto desenvolvido, o que permite a execução de longos programas de teste, ou até mesmo aplicações reais. A maioria desses *checkers* são *postmortem*, i.e., primeiro realizam a coleta de dados (conhecidos como *traces*) sobre as operações de memória disparadas pelos programas de teste e só ao fim da execução utilizam esses dados para a detecção de eventuais erros. Apesar de sugestões terem sido feitas de que verificadores pós-silício poderiam ser utilizados para verificação pré-silício (HANGAL et al., 2004; HU et al., 2012), apenas suas versões de melhor esforço podem ser eficientemente utilizadas, deixando de lado suas versões completas que fornecem garantias de verificação para cada teste (MANOVIT; HANGAL, 2006; HU et al., 2012).

A verificação pré-silício baseia-se na execução de testes sobre uma simulação do sistema com base em sua implementação. Isso permite validar o projeto durante seu desenvolvimento, sem o custo de sintetizar protótipos para a execução dos testes. Por outro lado, a vazão de instruções pode ser ordens de magnitude menor que a de um protótipo em *hardware*, o que torna testes extensos inviáveis. Em contraste aos verificadores pós-silício, apenas um verificador *postmortem* para pré-silício é relatado na literatura (RAMBO; HENSCHEL; SANTOS, 2011). Outros dois verificadores pré-silício, os quais atuam em tempo de execução (*runtime checkers*), são relatados. Um deles, proposto por Shacham et al. (2008), permite rápida verificação em tempo de execução, mas admite-se que possa levantar falsos negativos. Outro trabalho (FREITAS; RAMBO; SANTOS, 2013), mais recente, fornece garantias de ausência de falsos diagnósticos – falsos positivos e falsos negativos – para um dado programa de testes. Além disso, assim que um erro é encontrado, a execução do teste é terminada.

1.2 A GERAÇÃO DE TESTES

As técnicas de verificação dependem da execução de programas de teste sobre o projeto sendo testado. Alguns verificadores permitem o uso de programas reais, enquanto outros utilizam programas pseudo-aleatórios com valores pré-determinados para as leituras e escritas, a fim de facilitar a análise do verificador (FREITAS; RAMBO; SANTOS, 2013). Tem sido observado tanto em ambiente acadêmico (SHACHAM et al., 2008) quanto em ambiente industrial (HANGAL et al., 2004) que testes devem ser programas curtos, paralelos, com condições de corrida para um número pequeno de posições de memória, visto que condições de corrida agressivas tendem a expor erros em multiprocessadores mais rápido (MANOVIT; HANGAL, 2006).

Como a verificação pré-silício pode ser ordens de magnitude mais lenta que um protótipo em *hardware*, um trabalho recente (ANDRADE; GRAF; SANTOS, 2016) buscou tornar a geração pseudo-aleatória mais eficiente, baseando em uma especificação formal de cadeias canônicas de dependência de dados (GHARACHORLOO, 1995). Um trabalho ainda mais recente (ANDRADE, 2017) estende a ideia, propondo um mecanismo que explora a escolha dos endereços de memória utilizados pelos testes. Outros trabalhos propõem técnicas de geração de testes adaptativa (WAGNER; BERTACCO, 2008; ELVER; NAGARAJAN, 2016), de forma a guiar a criação de um novo teste com base nas informações dos testes anteriores e, assim, diminuir o esforço de verificação. Qin e Mishra (2012) propõe uma técnica dirigida pela cobertura do produto das máquinas de estados (FSMs) do protocolo de coerência utilizado.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

A fim de elaborar as bases de um futuro gerador adaptativo de testes para a verificação pré-silício, este trabalho tem como objetivo principal avaliar o impacto da escolha de diferentes técnicas de geração pseudo-aleatórias – e seus parâmetros de geração – sobre três diferentes métricas (cobertura, eficácia e esforço) dos testes gerados. Com esses dados, espera-se identificar aspectos do problema que possam servir como guias na elaboração de um novo gerador adaptativo.

1.3.2 Objetivos Específicos

- **Propor uma métrica de cobertura funcional**: para avaliar o impacto de diferentes técnicas e parâmetros de geração, é necessário definir e implementar uma métrica de cobertura apropriada.
- **Elaborar erros artificiais de projeto:** erros de projeto são necessários para desafiar técnicas de geração de forma a expor seus aspectos positivos e negativos e de forma a determinar como explorá-los ou evitá-los na construção de geradores melhores.
- **Avaliar um conjunto de técnicas de geração:** com base na métrica de cobertura proposta e nos erros de projeto elaborados, torna-se possível avaliar as técnicas propostas em (ANDRADE; GRAF; SANTOS, 2016; ANDRADE, 2017) quando comparadas a um gerador convencional.

1.4 ORGANIZAÇÃO DA MONOGRAFIA

O restante dessa monografia está organizado da seguinte forma: o Capítulo 2 faz uma revisão dos geradores de testes encontrados na literatura, ao final apontando algumas limitações observadas. O Capítulo 3 aborda os aspectos-chave da infraestrutura utilizada, descreve o protocolo de coerência e os erros de projeto usados na avaliação experimental e apresenta os geradores de testes a serem avaliados. O Capítulo 4 propõe uma métrica de cobertura funcional. O Capítulo 5 apresenta os resultados de cobertura, eficácia e esforço obtidos com a execução dos experimentos para cada uma das técnicas de geração avaliada. Finalmente, o Capítulo 6 sumariza as conclusões gerais desta monografia e propõe trabalhos futuros com base no conhecimento adquirido sobre o problema.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo serão apresentadas algumas técnicas de geração de testes para a verificação de modelos de memória, a começar pela geração de testes pseudo-aleatórios, amplamente utilizada devido sua simplicidade e escalabilidade. Em seguida, será abordada uma técnica baseada em protocolos de coerência de cache que, explorando propriedades da estrutura espacial de estados da FSM produto do protocolo, é capaz de obter eficientemente cobertura completa de estados e transições. Por último, serão abordadas duas técnicas adaptativas de geração, a primeira baseando-se no uso de agentes cooperantes para cada processador e a última explorando o uso de algoritmo genético.

2.1 GERAÇÃO DE TESTES PSEUDO-ALEATÓRIOS

Geradores de testes pseudo-aleatórios para modelos de memória vêm sendo utilizados tanto por verificadores pós-silício (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; ROY et al., 2006; HU et al., 2012), no meio industrial, quanto por verificadores pré-silício (SHACHAM et al., 2008; FREITAS; RAMBO; SANTOS, 2013) em ambiente acadêmico. Apesar de sua difusão, a maioria dos trabalhos têm seu foco na descrição dos algoritmos de análise e verificação, sendo que apenas dois deles (RAMBO; HENSCHEL; SANTOS, 2011; ANDRADE; GRAF; SANTOS, 2016) descrevem o algoritmo de geração de testes. Alguns parâmetros comuns ao gerador são a frequência de tipos de instruções (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; HU et al., 2012), o número de posições de memória compartilhadas (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; SHACHAM et al., 2008), e até mesmo sequências de instruções para induzir comportamentos específicos conhecidos cuja ocorrência é incomum.

Verificadores pós-silício podem contar com a alta vazão de instruções do protótipo em *hardware*, o que permite a execução de testes pseudo-aleatórios maiores a fim de obter uma gama satisfatória de comportamentos possíveis. O mesmo não é verdade para testes pré-silício, visto que são executados em um simulador do *hardware*. Para resolver esse problema, um trabalho recente (ANDRADE; GRAF; SANTOS, 2016) baseou-se em especificações formais (GHARACHORLOO, 1995) para produzir testes pseudo-aleatórios que contenham apenas sequências de instruções que sejam capazes de gerar estímulos significativos ao modelo

de memória sendo testado. Uma extensão desse trabalho (ANDRADE, 2017) apresenta uma técnica para definir quais endereços de memória efetivos serão utilizados pelos programas de teste, fator importante para controlar a ocorrência de substituições de blocos de *cache*.

Mesmo com refinamentos sobre a produção de testes pseudo-aleatórios, a configuração dos parâmetros de geração continua sendo responsabilidade do engenheiro de verificação. Nem sempre é trivial definir quais os melhores parâmetros para alcançar limites satisfatórios de cobertura para o projeto sob verificação. As técnicas apresentadas nas próximas seções procuram abordar esse problema.

2.2 GERAÇÃO DE TESTES DIRIGIDA POR COBERTURA

Qin e Mishra (2012) propõem um algoritmo de geração automática de testes dirigidos pela cobertura de transições do protocolo de coerência de *cache* adotado pelo sistema a ser testado. O trabalho é primariamente motivado pelo algoritmo de busca em largura (BFS), com o qual é possível gerar testes que ativam todos os estados e transições independente da estrutura espacial da FSM do protocolo utilizado. Essa técnica apresenta dois problemas. Boa parte das instruções de um teste são utilizadas para induzir as transições mais próximas do estado inicial, que nada agregam à cobertura já alcançada. Além disso, devido à necessidade de gravar todos os estados que foram visitados, os requisitos de memória em tempo de execução crescem exponencialmente.

Para tratar os problemas do BFS, Qin e Mishra (2012) sugerem decompor o espaço de estados da FSM produto do protocolo em componentes estruturalmente mais simples (cliques e hipercubos). Com base nesses componentes, o trabalho apresenta um método *on-the-fly* (dinâmico) de geração de testes dirigidos baseado em caminho euleriano, o qual requer espaço linear em respeito ao número de núcleos. Para cada transição que se deseja estimular, o algoritmo define operações de leitura, escrita e substituição de blocos de *cache* (através de leituras em endereços que mapeiam para o mesmo bloco), de forma a criar programas paralelos (utilizando instruções *load-linked* e *store-conditional* do conjunto de instruções *Alpha*) capazes de obter 100% de cobertura sobre estados e transições. Em comparação com BFS, os experimentos realizados por Qin e Mishra (2012) mostraram que a técnica proposta gera testes de 48% a 60% menores.

2.3 GERAÇÃO ADAPTATIVA DE TESTES

Geradores adaptativos de testes baseiam-se no uso de metaheurísticas para definir os próximos testes a serem gerados. Para isso, requerem um sistema de retroalimentação, onde dados sobre o último teste (ou conjunto de testes) são coletados e utilizados para definir os próximos testes. Dois trabalhos (WAGNER; BERTACCO, 2008; ELVER; NAGARAJAN, 2016) utilizam essa abordagem para a geração automática de testes. As próximas duas subseções trataram essas técnicas em detalhe.

2.3.1 MCjammer

MCJammer, nas palavras de Wagner e Bertacco (2008), é uma ferramenta de verificação adaptativa para projetos de multiprocessadores que utiliza um ciclo fechado de *feedback* para dinamicamente ajustar a simulação para efetivamente estimular *corner cases* do comportamento do projeto. A técnica baseia-se no uso de uma rede de agentes cooperantes, cada agente associado a um núcleo de processamento. Ao fim de cada execução, dados sobre cobertura e *pressão* são coletados e utilizados para guiar a produção das instruções para a próxima execução. A *pressão* é definida como o tempo médio entre eventos conflitantes nas *caches* e controladores de memória, servindo para maximizar o "estresse" sobre o sistema de memória. Agentes podem decidir cumprir objetivos próprios sobre a cobertura do sistema ou ajudar outro agente a cumprir um de seus objetivos. O sistema é configurado de forma que agentes com mais objetivos próprios cumpridos têm mais chance de optar por ajudar outro agente.

Para medir a cobertura, Wagner e Bertacco (2008) definiram uma estrutura chamada máquina de estados finita dicotômica (DFSM), construída com base no protocolo de coerência adotado (no caso do trabalho, o protocolo MESI). Essa estrutura representa uma FSM produto de dois processadores, com um vetor adicional de cobertura de tamanho n em cada aresta, onde n é o número de processadores no *chip*. Cada agente tem sua própria DFSM e, dessa forma, é possível capturar todas as transições da combinação de dois a dois de todos os processadores. Como essa representação não cobre todas as transições da FSM produto do protocolo, cada transição, para cada elemento do vetor de cobertura, é executada um certo número de vezes até ser considerada coberta, a fim de compensar o efeito da abstração da FSM produto.

2.3.2 McVerSi

Elver e Nagarajan (2016) apresentam uma ferramenta de geração de testes batizada de McVerSi, a qual utiliza um algoritmo genético para guiar a geração adaptativa de testes. Cada programa de teste é tratado como um *cromossomo* pelo algoritmo, sendo representado como um grafo orientado acíclico (onde os vértices são operações e as arestas representam sua relação de ordem). Cada vértice do grafo é tratado como um gene, e representa uma operação em alto-nível de uma *thread*. A cobertura estrutural (código-fonte) da especificação do protocolo de coerência é utilizada como função-objetivo (*fitness function*) a ser maximizada. Cada teste é executado um número específico de iterações, após as quais tem seu valor calculado para a função-objetivo.

Para a realização do cruzamento (*crossover*), é proposta uma técnica de cruzamento seletivo, a qual busca as operações (*genes*) que mais contribuem para o não-determinismo médio do teste (*cromossomo*). Por não-determinismo médio de um teste entende-se o número médio de eventos que estão em *ordem de conflito*[1] antes de qualquer ocorrência de um dado evento no teste. Isso faz com que operações de memória envolvidas em condições de corrida sejam favorecidas. Sendo assim, o algoritmo busca, de forma geral, ao final de cada teste executado produzir um novo teste que conduza a uma porcentagem maior de cobertura e ao mesmo tempo favoreça operações que contribuam com condições de corrida, as quais, como já dito anteriormente, ajudam a expor erros mais rápido (MANOVIT; HANGAL, 2006).

2.4 LIMITAÇÕES OBSERVADAS

Verificadores pensados para validação pós-silício não precisam se preocupar em utilizar testes dirigidos por cobertura, pois podem rodar testes maiores e em maior quantidade sobre o protótipo do próprio *hardware*. A verificação pré-silício não pode contar com a mesma vazão de testes e, portanto, precisa usar técnicas que melhorem a eficácia de testes pseudo-aleatórios (ANDRADE; GRAF; SANTOS, 2016; ANDRADE, 2017) ou contar com testes dirigidos (QIN; MISHRA, 2012; WAGNER; BERTACCO, 2008; ELVER; NAGARAJAN, 2016). Geradores dirigidos que garantem plena cobertura global explorando propriedades do espaço de verificação (QIN; MISHRA, 2012) têm como grande desvantagem sua de-

[1] Duas operações de memória são conflitantes quando acessam a mesma posição de memória e pelo menos uma delas é uma escrita.

pendência do protocolo específico de coerência do projeto sob verificação. Portanto, não são reusáveis para projetos que utilizem protocolos diferentes, requerendo uma nova análise da FSM associada ao protocolo e subsequente divisão da mesma em componentes adequados para a travessia euleriana. Da mesma forma, protocolos mais complexos, envolvendo múltiplos níveis de *cache*, requerem extensões no algoritmo, que foram apenas indicadas em (QIN; MISHRA, 2012), não tendo sido dadas garantias de que uma FSM produto que combine múltiplos níveis hierárquicos conserve as mesmas propriedades demonstradas para um único nível de cache. Sem essas garantias, não se pode estar certo de que se possa atingir plena cobertura para protocolos com múltiplos níveis de cache com a mesma eficiência computacional.

Por outro lado, geradores adaptativos baseados no uso de metaheurísticas para maximizar a cobertura (WAGNER; BERTACCO, 2008; ELVER; NAGARAJAN, 2016) não exploram propriedades do espaço global de verificação, o que torna seus algoritmos complexos ou requer o uso de estimativas simples de cobertura (para que possam ser obtidas *on-the-fly*). Além disso, as DFSM propostas por (WAGNER; BERTACCO, 2008) apenas abordam protocolos com um nível de *cache*, apontando o suporte para hierarquias de memória *cache* como trabalho futuro. Outro detalhe interessante a ser notado é que em dois dos trabalhos (WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012) apenas transições de estados estáveis são consideradas. Estados e transições estáveis são aqueles originalmente especificados pelo protocolo abstrato. Entretanto, implementações reais desses protocolos requerem o uso de estados transientes (e.g. para a sincronização de operações) que garantem o correto funcionamento do protocolo. Apenas na cobertura estrutural adotada por (ELVER; NAGARAJAN, 2016) tais transições são contempladas.

Portanto, alguns trabalhos correlatos baseiam-se na FSM produto para definir uma métrica *global* de cobertura funcional que explicita todos os comportamentos *implicitamente* especificados pelo protocolo de coerência. Como o número de estados da FSM produto cresce exponencialmente com o número de núcleos, outros trabalhos preferem não enumerar explicitamente todos os comportamentos, mas basear-se em uma métrica *local* de cobertura funcional que capture 1) eventos induzidos localmente pelo processador e pelos controladores de cache privativas (*loads, stores, evictions*) e 2) eventos locais induzidos remotamente por outro processador (requisições de invalidação e requisições de coerência repassadas ao controlador de uma cache privativa). O trabalho descrito nesta monografia adota esta última abordagem, ou seja, propõe-se a avaliar o impacto de diferentes técnicas e parâmetros

de geração no espaço *local* de verificação para definir propriedades que possam ser utilizadas por um algoritmo adaptativo, o qual se manterá reusável para projetos que utilizem diferentes protocolos. Esta monografia supõe ser verdadeira a seguinte hipótese: a de que a limitação de se adotar o espaço local de verificação para determinar os valores dos parâmetros de geração mais adequados possa ser compensada (em parte) pelo uso de uma técnica (ANDRADE; GRAF; SANTOS, 2016) que restringe o espaço global de verificação por meio de restrições embutidas em cadeias canônicas de dependências, as quais se estendem ao longo de múltiplos núcleos (GHARACHORLOO, 1995). Esta hipótese justifica-se pelo fato de que essa técnica, em princípio, tem o potencial de evitar a visita de transições da FSM produto que seriam incapazes de expor comportamentos inválidos. A validação dessa hipótese, entretanto, será objeto de trabalho futuro.

3 INFRAESTRUTURA ADOTADA

Para a simulação, foi utilizada uma infraestrutura de domínio público, o *gem5 simulator* (BINKERT et al., 2011), configurado com um modelo de CPU *out-of-order* (O3). Além disso, foi adotado o conjunto de instruções (ISA) do SPARC e o modelo de memória do processador Alpha (GHARACHORLOO, 1995). A hierarquia de memória utilizada consiste em caches L0 privativas (separadas em dados e instruções), caches L1 privativas (unificadas), e uma cache L2 compartilhada, com tamanhos de 4 KiB (mapeamento direto), 64 KiB (2-*way*) e 2 MiB (8-*way*), respectivamente. Todos os níveis operam com o mesmo tamanho de bloco (64 bytes) e utilizam a mesma política de substituição (LRU).

A escolha dessas configurações está diretamente relacionada com a escolha do *checker* e dos geradores, uma vez que a implementação dos mesmos foi realizada nessas condições. Além disso, o modelo de memória do Alpha possui um grau de relaxação maior, tornando-o um alvo mais desafiador para a verificação. As próximas seções abordarão em mais detalhes o protocolo de coerência, os erros de projeto utilizados, os geradores de testes e o *checker* escolhido, respectivamente.

3.1 O PROTOCOLO MESI

Para garantir a validade das cópias de dados distribuídos pelas diferentes caches da hierarquia, um protocolo de coerência é necessário. O gem5 fornece a implementação de uma variante do protocolo MESI (PAPAMARCOS; PATEL, 1984). Nessa implementação, a hierarquia de caches é inclusiva: se um bloco está presente na cache de um determinado nível, também estará presente nas caches dos níveis inferiores. O controlador da cache de último nível (LLC), i.e., o controlador da cache L2, exerce também a função de diretório, mantendo uma tabela de informações sobre quais caches privativas possuem um dado bloco de memória. Dessa forma, não há necessidade de *snoopers* para cada cache privada a fim de manter a coerência: o diretório funciona como um roteador das mensagens de coerência, direcionando-as de acordo com as informações guardadas na tabela.

A sigla MESI refere-se aos quatro possíveis estados que um bloco pode se encontrar sob a perspectiva do controlador da cache de um dado processador. O estado M (*Modified*) garante permissão de leitura/escrita e indica que o bloco já foi modificado. O estado E (*Exclusive*)

garante permissão de apenas leitura, mas admite a realização de um *silent upgrade* (i.e., sem notificar os níveis superiores da hierarquia) para o estado M. Isso é possível pois ambos os estados garantem a *ownership* do bloco, i.e., apenas a cache privada do processador em questão tem o bloco carregado. O estado S (*Shared*) garante permissão de apenas leitura e indica que uma ou mais caches privativas contém cópia de um mesmo bloco de memória. O estado I (*Invalid*) indica que o bloco é inválido, ou seja, seu acesso não é permitido (nem leituras, nem escritas). A Figura 1 apresenta a FSM do controlador da L0. As transições de um estado para si próprio e as ações de saída foram omitidas para simplificar o diagrama. As FSMs dos outros controladores podem ser consultadas no Anexo A.

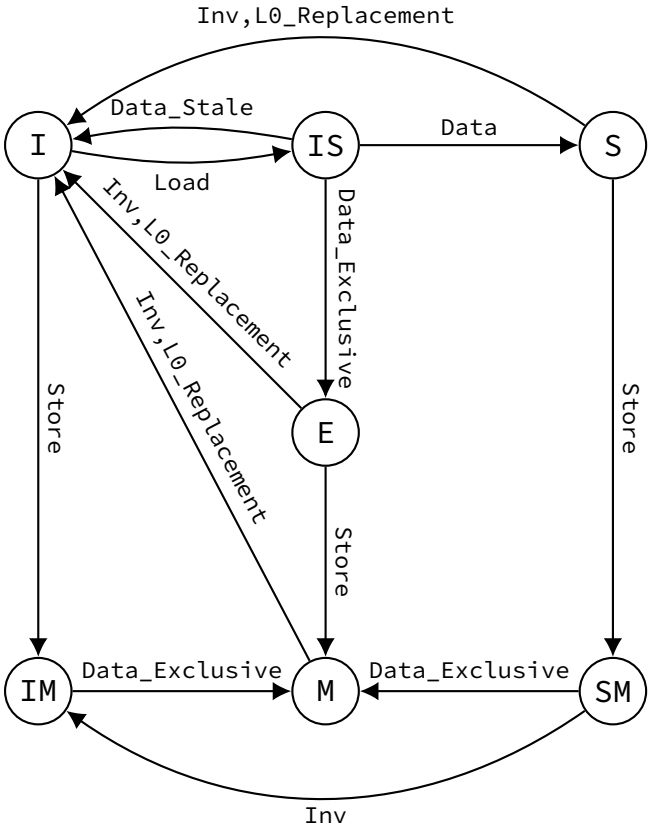


Figura 1 – FSM do controlador de cache do nível L0

A partir da Figura 1 nota-se que a FSM do protocolo não se resume a apenas os quatro estados definidos pela sigla, mas também outros estados auxiliares. Esses estados são chamados de estados transientes e tem como funcionalidade bloquear o acesso ao bloco – sem permissões de leitura ou escrita – enquanto aguarda uma resposta de outra cache. Ou seja, esses estados são necessários para sincronizar ações que envolvem mais de uma cache. Também vale ressaltar que a cache L0 se comunica diretamente com a cache L1, sendo que é a cache L1 quem exerce o papel de *owner* (ou *sharer*) e comunica-se com o diretório (LLC) e as caches L1 de outros processadores. O resultado prático dessa abordagem é a simplificação da FSM do nível L0.

Para acessar um bloco de memória, os controladores emitem requisições para adquirir permissões somente leitura, identificadas no gem5 como GETS, e permissões de leitura/escrita, identificadas como GETX e UPG (GETX quando o bloco está inválido e UPG quando este já tem permissão de leitura). Quando essas requisições são provenientes de processadores remotos, são identificadas por FWD_GETX e FWD_GETS. A ocorrência de *evictions* somente é notificada aos níveis inferiores quando o bloco está *dirty* (modificado), pois nesse caso é necessário atualizar as cópias desatualizadas (eventos PUTX). Outro evento comum ao protocolo é o *Invalidate*, o qual pode ser causado por concorrência de diversos blocos por uma mesma linha de cache ou requisições de leitura/escrita remotas.

3.2 OS ERROS DE PROJETO

A criação de erros artificiais é necessária para colocar os geradores à prova, verificando sua capacidade de expor erros. Cada erro criado é injetado em um caso de projeto separado, de forma que, quando ocorre uma detecção, é possível afirmar que esta foi relativa apenas ao erro criado. A Tabela 1 apresenta uma seleção de 6 erros de projeto (nomeados de E1 a E6). Nela são identificados o nível em que o erro foi criado, o estado atual da FSM, o evento que dispara a transição, o estado destino da transição e, por fim, se houve alguma ação de saída removida. Esses erros foram desenvolvidos em colaboração com outros membros do grupo de pesquisa e fazem parte de um acervo maior.

Os erros foram criados a partir de alterações nas máquinas de estados do protocolo de coerência. São feitas modificações de dois tipos básicos: a mudança do estado destino da transição – representado na tabela pelo uso da expressão *instead of* na coluna "Próximo estado" – e a

Tabela 1 – Erros de projeto selecionados

ID	Nível	Estado atual	Evento de entrada	Próximo estado	Ação de saída removida
E1	L0	E	Store	E instead of M	nenhuma
E2	L1	IS	Data_Exclusive	E	writeDataFromL2Response
E3	L1	E	WriteBack	MM	writeDataFromL0Request
E4	L1	IS_I	DataS_fromL1	I	writeDataFromL2Response
E5	L1	EE	Fwd_GETX	SS	data block of sendDataToRequestor
E6	L1	M_IL0	WriteBack	MM_IL0	writeDataFromL0Request

remoção de uma ou mais ações de saída de uma transição – informado na coluna "Ação de saída removida". Erros mais complexos podem requerer a modificação de mais transições, ou até mesmo a criação de novos estados, mas esse não é o caso dos erros aqui selecionados.

Cada erro tem um cenário de exposição, onde as ações de leitura/escrita dos processadores induzem a sequência de transições necessária para que o erro ocorra e seja detectado. O erro E1 é estimulado por *conflito intra-processador*, uma leitura seguida de uma escrita, onde a leitura recebeu acesso exclusivo ao bloco (estado E), i.e., não há qualquer outra cache privativa com aquele bloco. O erro E2 é estimulado por uma única leitura, desde que essa também tenha recebido acesso exclusivo. Os erros E3 e E6 são estimulados por *evictions* no nível L0, com a diferença de que o último requer que primeiro tenha acontecido uma invalidação do bloco (resultante de conflito intra ou inter-processador). O erro E4 é estimulado por *conflitos inter-processador* onde, antes que uma leitura fosse concluída e transferida para o nível L0, uma escrita ocorre em outro processador e induz um sinal de invalidação do bloco. Isso significa que o bloco que o controlador da cache receberá será um bloco *stale*, o qual só pode ser utilizado para aquela leitura. Por fim, erro E5 é estimulado por duas leituras em um processador e uma terceira em outro: a primeira leitura é de um endereço no bloco em que ocorrerá o erro, a segunda leitura é de outro endereço que cause a *eviction* do bloco anterior no nível L0, e a terceira leitura (no processador remoto) é do mesmo endereço da primeira leitura. Todos esses erros são detectados por induzirem a violação de um invariante de protocolos de coerência[1], o qual é capturado por um axioma do modelo de memória (comumente denominado de Axioma de Valor).

[1] O assim-chamado *Data Value Invariant* especifica que o valor atribuído a uma posição de memória ao final de uma época, precisa coincidir com o valor lido dessa posição em época imediatamente posterior.

3.3 OS GERADORES DE TESTES

Quatro geradores diferentes foram utilizados para a geração de testes. PLAIN- refere-se a um gerador pseudo-aleatório convencional, similar aos utilizados em outros trabalhos de verificação de modelos de memória (HANGAL et al., 2004; SHACHAM et al., 2008). Como não foi encontrada nenhuma implementação disponível em domínio público, um protótipo foi construído com base no pseudocódigo reportado em Rambo, Henschel e Santos (2011). Nesse gerador, operações de leitura, escrita e barreiras de memória são distribuídas aleatoriamente entre as diferentes *threads* de execução, uma para cada núcleo de processamento. Todas as *threads* possuem o mesmo número de instruções e os endereços sobre os quais as operações são realizadas também são escolhidos aleatoriamente. Cada operação de escrita utiliza um valor diferente das outras para fins de verificação.

Outro gerador, denominado PLAIN+, é similar ao gerador aleatório, diferenciando-se apenas na forma como os endereços de memória são escolhidos. Nesse gerador, a escolha dos endereços de memória compartilhados que são utilizados pelo programa de testes é realizada por uma técnica (ANDRADE, 2017) onde os parâmetros especificados definem quais padrões de bits são aceitáveis para a formação dos endereços. Nesse sentido, são controlados os índices, *tags* e partes do *offset* para garantir que os parâmetros sejam satisfeitos. A maior vantagem desse mecanismo é a capacidade de controlar o nível de competição dos blocos de memória por uma mesma linha (ou conjunto associativo) de cache, o que permite controlar a ocorrência de *evictions*.

O terceiro gerador, CHAIN-, utiliza a noção de cadeias canônicas de dependência (GHARACHORLOO, 1995) para guiar a geração das sequências de instruções. Proposta por Andrade, Graf e Santos (2016), essa técnica procura diminuir o espaço de geração de testes, reduzindo a ocorrência de sequências de operações redundantes para a verificação do modelo de memória. A implementação utilizada foi especificamente projetada com base em quatro cadeias definidas para o modelo de memória do processador Alpha, através das quais definem-se sequências de operações intra-processador (categoria 0) e inter-processador (categorias 1, 2 e 3). O quarto gerador, CHAIN+, é composto pelo gerador CHAIN- e a técnica de escolha de endereços dirigida.

Os quatro geradores possuem alguns parâmetros de geração em comum: o número de *threads* (p), uma para cada processador do projeto sendo testado, o tamanho do programa de teste (n), o número de posições de memória compartilhadas (s) e um número para inici-

alizar o gerador de números pseudoaleatórios utilizado (*seed*). Com base nos três primeiros parâmetros, define-se o *cenário de verificação* como o conjunto de testes que compartilham uma mesma configuração $v = (p, n, s)$.

3.4 O CHECKER

A verificação sobre os testes é realizada por um *checker* pré-silício *on-the-fly* (FREITAS; RAMBO; SANTOS, 2013), i.e., a verificação ocorre simultaneamente à execução do teste e assim que um erro é detectado a execução é finalizada. Dessa forma, evita-se a execução de instruções que não têm utilidade para a detecção do erro, diminuindo o esforço de verificação. O *checker* trabalha com três pontos de monitoramento por núcleo e garante que não ocorram falsos negativos nem falsos positivos quando o teste é capaz de estimular um erro, tornando-o apropriado para a avaliação das técnicas de geração de testes.

4 A MÉTRICA DE COBERTURA

Um modelo de memória especifica múltiplos comportamentos válidos para a execução de um programa de teste do subsistema de memória compartilhada, o que o torna inapropriado para estabelecer uma estimativa de cobertura funcional. Para contornar esse problema, algumas técnicas de verificação de modelos de memória utilizam cobertura estrutural (ELVER; NAGARAJAN, 2016), enquanto verificações baseadas no protocolo de coerência (WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012) podem utilizar diretamente a FSM do protocolo para estabelecer a métrica de cobertura. Portanto, para estimar a cobertura funcional, foi necessário adotar uma métrica dependente do protocolo de coerência utilizado no projeto sob verificação. As próximas seções apresentam dois tipos de cobertura de transições (relativas às máquinas de estados do protocolo de coerência), definem formalmente a métrica adotada, e fazem uma breve explicação de como a coleta da informação de cobertura foi implementada.

4.1 COBERTURA DE TRANSIÇÕES

Na especificação do protocolo de coerência do controlador de um nível de cache qualquer, uma transição da FSM que captura o comportamento do protocolo pode ser definida como um par (A, E), onde A refere-se ao estado atual e E ao evento de entrada que dispara a transição. Como a implementação dos protocolos utilizam FSMs determinísticas, não há motivos para especificar o estado destino da transição representada. Sendo assim, para determinar a cobertura de transições de uma FSM é necessário registrar todos os pares (A, E) que são estimulados com a execução dos testes.

Cada nível da hierarquia de memória possui uma máquina de estados específica que implementa sua respectiva parte do protocolo de coerência. Cada instância de memória cache tem seu próprio controlador e cada controlador tem sua própria instância da FSM do nível correspondente. O protocolo de coerência é definido pela interação das FSMs especificadas para cada controlador de cache em um determinado nível da hierarquia. Os comportamentos implicitamente especificados podem ser explicitamente enumerados através da construção de uma FSM produto. Isso dá origem a abordagens de cobertura local ou global, como explicado no Capítulo 2. Por exemplo, Qin e Mishra (2012)

utilizam uma métrica global de cobertura, enquanto Wagner e Bertacco (2008) usam uma simplificação que é um meio-termo entre a cobertura global e a cobertura local, pois captura a interação entre pares de processadores.

A cobertura local tem a desvantagem de não capturar de forma direta as informações resultantes da interação entre os diversos processadores e níveis de cache, mas, em contrapartida, provê uma forma simples de estimar cobertura, requerendo espaço que cresce linearmente com o o produto entre o número de processadores e o número de níveis hierárquicos privativos. A cobertura estrutural adotada por Elver e Nagarajan (2016) assemelha-se a uma métrica de cobertura local, visto que se refere diretamente ao código que implementa cada FSM. Este trabalho também adota uma métrica de cobertura local, pois sua simplicidade será provavelmente benéfica ao desenvolvimento de geradores adaptativos no futuro.

4.2 DEFINIÇÃO DA MÉTRICA ADOTADA

Dado um projeto livre de erros, a *cobertura funcional* é medida como a fração de transições cobertas na máquina de estados de cada controlador de cache. São rastreados todos os blocos de memória referenciados pela *coleção de testes* pseudoaleatórios de um dado cenário de verificação $v = (p, n, s)$ (como definido na Seção 3.3). Para cada bloco referenciado correspondente a uma posição de memória a sob a perspectiva da cache de um processador i no nível L, determinam-se quantas transições *distintas* foram exercitadas na respectiva máquina de estados do protocolo como resultado da execução da coleção de testes. Dessa forma, obtém-se o número cumulativo de transições cobertas, denotado como $TRAN_a^{i,L}(v)$.

Seja $total(L)$ o número total de transições da máquina vinculada ao nível L. Computa-se a *cobertura de transições* como $TC_a^{i,L}(v) = TRAN_a^{i,L}(v)/total(L)$. Em seguida, obtém-se a distribuição de coberturas de transições induzidas por v no nível L, denotado como $TC(v, L)$, i.e., a distribuição de $TC_a^{i,L}(v)$ sobre todos os processadores (i) e todas as posições de memória (a). Também foi obtida uma distribuição simular no nível compartilhado de cache (LLC), denotado como $TC(v, L2)$. Por fim, para cada nível L, são definidas a *mediana*, o valor *mínimo* e o valor *máximo* de $TC(v, L)$ sobre a coleção de todos os cenários v, escritos $\widehat{TC}(L)$, $TC^{min}(L)$ e $TC^{max}(L)$, respectivamente.

4.3 IMPLEMENTAÇÃO

Para avaliar o comportamento da cobertura local proposta, um módulo de coleta foi produzido em *C++* e integrado ao simulador gem5. Cada controlador de cache possui sua própria instância do módulo, o qual registra as transições exercitadas para cada endereço de memória compartilhado. Ao fim da execução, um arquivo com as informações coletadas é criado para cada instância. Esses arquivos utilizam a sintaxe *JSON* para guardar as informações, os quais podem ser manipulados por *scripts* auxiliares (escritos em *Python*) para agrupar as informações (de acordo com o que foi definido na seção anterior) e criar gráficos.

5 AVALIAÇÃO EXPERIMENTAL

Nesse capítulo, são apresentados os resultados experimentais resultantes da execução dos conjuntos de testes. Os experimentos para a análise de cobertura, realizados em um projeto livre de erros, totalizaram 14400 testes, enquanto os experimentos para a análise de eficácia e esforço totalizaram 86400 casos de uso (6 erros $\times$14400 testes). Nas próximas seções são apresentados os parâmetros utilizados para a geração dos programas de testes, a análise de cobertura e a análise de eficácia e esforço. Por conveniência, serão utilizados os termos *chaining* e *biasing* para se referir as técnicas de CHAIN (presentes em CHAIN- e CHAIN+) e da atribuição de endereços sob restrições (presente em PLAIN+ e CHAIN+), respectivamente.

5.1 PARÂMETROS DOS GERADORES

Foram simuladas arquiteturas de 8, 16 e 32 núcleos (parâmetro p) para a execução dos testes. Para cada arquitetura, foram gerados programas com quinze sementes aleatórias (1, 2, ..., 15), quatro quantias diferentes de posições de memória compartilhadas (s = 4, 8, 16 e 32), e cinco tamanhos de teste (n = 1k, 2k, 4k, 8k, 16k), onde k representa 2^{10} operações. Além disso, os geradores PLAIN- e PLAIN+ aceitam como parâmetro um *mix* de instruções o qual determina a porcentagem aproximada de cada tipo de instrução, enquanto os geradores CHAIN- e CHAIN+ aceitam um *mix* de categorias, determinando o quanto de cada categoria é aplicado na geração do programa de teste. Foram usados quatro *mixes* diferentes para cada gerador, apresentados na Tabela 2.

Tabela 2 – *Mixes* utilizados

***Mix* de instruções**			***Mix* de categorias**			
Load	Store	Membar	$\mathcal{C}_0$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$
0.30	0.66	0.04	0.40	0.60	0.00	0.00
0.48	0.48	0.04	0.00	1.00	0.00	0.00
0.66	0.30	0.04	0.00	0.80	0.20	0.00
0.80	0.16	0.04	0.00	0.80	0.00	0.20

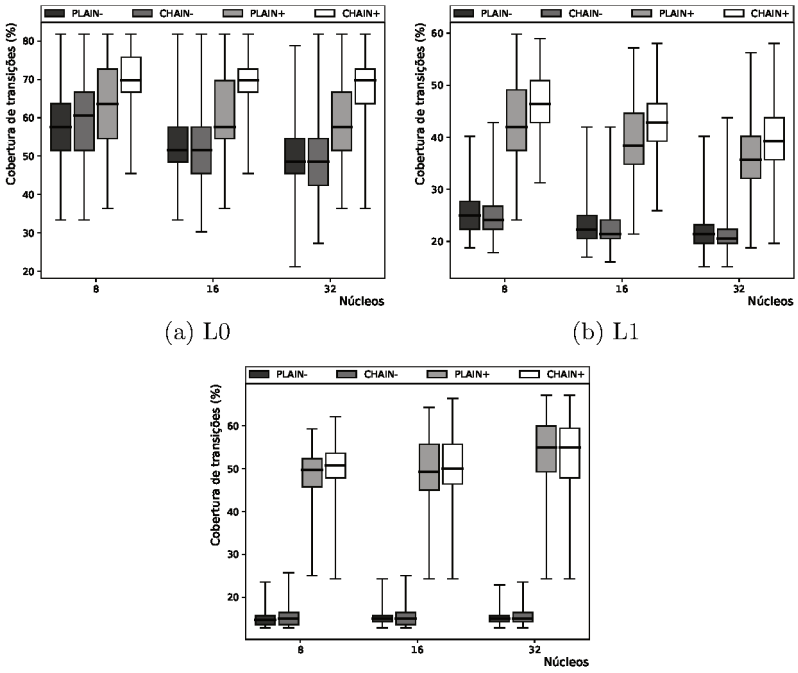
Os geradores que utilizam *biasing* também recebem três parâmetros para determinar o comportamento do módulo de atribuição de endereços. O parâmetro *abc* (*alignment biasing constraint*) especifica que o endereço é alinhado a 2^{abc} bytes. O parâmetro *sbc* (*sharing biasing constraint*) é um valor booleano que garante *true sharing* ou não. O último parâmetro, *cbc* (*competition biasing constraint*) é um par (κ, χ), onde κ indica o número de linhas de cache distintas para onde são mapeadas as posições compartilhada usadas no teste e χ indica o máximo de competição por uma mesma linha de cache. Esse terceiro parâmetro é especialmente útil para controlar a ocorrência de *evictions*. Para a geração dos experimentos, foram adotados $abc = 2^6$, *true sharing* ($sbc = true$) e $cbc = (1, s)$. Esse *cbc* foi escolhido por gerar o máximo possível de *evictions* para a faixa de valores usados de s, permitindo fazer uma avaliação bastante independente da associatividade adotada em cada nível hierárquico.

5.2 ANÁLISE DE COBERTURA

Os resultados de cobertura foram sumarizados na Figura 2, na forma de diagramas de caixa representando a cobertura para cada arquitetura testada (8, 16 e 32 núcleos) em cada nível da hierarquia. Os *whiskers* representam o mínimo e o máximo dos dados, de forma que os gráficos apresentam $\widehat{TC}(L)$, $TC^{min}(L)$, $TC^{max}(L)$ e a dispersão da cobertura. Os gráficos mostram que a cobertura mediana de CHAIN- não melhora em relação a de PLAIN- (com exceção do nível L0 da arquitetura de 8 núcleos). Entretanto, *biasing* melhora a cobertura de forma significativa, principalmente nos níveis hierárquicos mais baixos (L1 e L2). O fato de a melhoria aumentar do maior nível para o menor nível hierárquico mostra o quão inefetiva é a atribuição aleatória de endereços tendo em vista as associatividades progressivamente maiores em direção aos níveis mais baixos da hierarquia de memória.

O gerador CHAIN+, o qual combina *chaining* e *biasing*, alcançou os maiores valores de cobertura mediana para todos os níveis e todos os números de núcleos, exceto o L2 de 32 núcleos, no qual a mediana empata com PLAIN+. Além disso, CHAIN+ alcançou os maiores valores de cobertura máxima e mínima, com exceção dos projetos de 8 núcleos nos níveis L1 e L2, respectivamente. Isso é uma primeira evidência de que as duas técnicas têm uma natureza complementar: *biasing* causa mais impacto no último nível da hierarquia (L2), enquanto *chaining* tem mais impacto no primeiro nível (L0).

Figura 2 Distribuições de cobertura para projetos de 8, 16 e 32 núcleos



(a) L0 (b) L1

(c) L2

As técnicas de *chaining* e *biasing* estimulam operações conflitantes e eventos de *eviction*, respectivamente. Essas classes diferentes de eventos comumente induzem transições *distintas* na máquina de estados controlando uma cache privativa, o que contribui para o aumento da cobertura com os testes gerados por CHAIN+. O *biasing* estimula transições relativas à *evictions* em todos os níveis. No nível L0 (Figura 2a), *biasing* também estimula transições a partir do estado I, enquanto *chaining* estimula transições dos estados E, S e M, seja através de conflitos intra-processador (transições (S,M) e (E, M)) ou através de conflitos inter-processador (transições (M,I), (E,I), e (S,I)).

No nível L1 (Figura 2b), *biasing* estimula transições induzidas por requisições GETS e GETX do nível L0 (vide Seção 3.1), enquanto *chaining* estimula transições induzidas não somente por requisições UPG do nível L0, como também requisições de outros núcleos (Invalidate, FWD_GETS, FWD_GETX). Nota-se que o impacto das técnicas

quando combinadas é maior na L0 que na L1. Uma razão para isso é a exploração das caches inclusivas, onde operações conflitantes, apesar de gerarem *diferentes* tipos de requisições, acabam disparando a *mesma* transição na L1. Outra razão é a menor frequência de *evictions* devido a maior associatividade na L1.

No nível L2 (Figura 2c), a cobertura mediana foi cerca de 15% para PLAIN- e CHAIN- e 53% para *ambos* PLAIN+ e CHAIN+. Isso é mais uma evidência de que o impacto de *chaining* é marginal no último nível de cache, onde o diretório também se encontra. A explicação encontra-se no fato de que a maioria das requisições resultantes de conflitos inter-processador (Invalidate, FWD_GETS, FWD_GETX) não induzem transições na L2: elas são, na verdade, ações de saída. Apenas 4 das 140 transições do nível L2 são induzidas por conflitos intra-processador (UPG) ou inter-processador (GETX, GETS). Por isso, a maioria das transições no nível L2 são estimuladas apenas pelo *biasing*.

5.3 ANÁLISE DE EFICÁCIA E ESFORÇO

Para avaliar a *eficácia* de um gerador em expor um dado tipo de erro sob um cenário $v = (p, n, s)$, foi medida a fração $\varepsilon(v)$ de todos os testes induzidos por v para os quais violações foram detectadas. Ao assumir-se uma amostragem suficientemente grande, essa fração pode ser interpretada como a probabilidade de um gerador de expor aquele tipo de erro quando os parâmetros de geração estão configurados para $v = (p, n, s)$. Para estimar o *esforço de verificação* resultante da execução dos testes induzidos por um cenários $v = (p, n, s)$ na tentativa de expor um erro, combinou-se a eficácia e o tempo médio de execução dos testes do cenário v para um dado projeto. A métrica de esforço foi definida em trabalho correlato (ANDRADE, 2017) e é brevemente explicada abaixo.

Nos casos em que não houve detecção, o esforço no cenário foi dado diretamente pela soma de todos os tempos de execução dos testes. Caso contrário, o esforço médio no cenário foi calculado como $(\lceil 1/\varepsilon(v) \rceil - 1)\ \widehat{t^0(v)} + \widehat{t^1(v)}$, onde $\varepsilon(v)$ é a eficácia medida para o cenário v, $\widehat{t^0(v)}$ é o tempo médio de execução dos testes que não expuseram o erro e $\widehat{t^1(v)}$ é o tempo médio dos testes que expuseram. Dessa forma, contabilizou-se todo o tempo gasto por testes que não levaram à detecção do erro mais o tempo médio que um teste levou para expô-lo.

A Tabela 3 mostra se um gerador foi ou não capaz de expor um erro com um dado tamanho de teste em um projeto de 32 núcleos

(utilizando 32 variáveis compartilhadas). Entradas com o fundo preto indicam que naquele cenário nenhum teste foi capaz de expor o erro. Ela também mostra o esforço gasto (expresso em segundos) para expor um erro ou tentando expô-lo. Além disso, a eficácia de cada cenário é mostrada entre parênteses abaixo do esforço, e a tabela tem uma divisão entre *unbiased* e *biased*, referindo-se aos geradores que não utilizam e que utilizam *biasing*, respectivamente.

Tabela 3 – Esforço e eficácia para projetos de 32 núcleos e 32 variáveis compartilhadas

n	Unbiased											
	PLAIN-						CHAIN-					
	E1	**E2**	**E3**	**E4**	**E5**	**E6**	**E1**	**E2**	**E3**	**E4**	**E5**	**E6**
1k	320 (0.03)	645 (0.00)	719 (0.00)	658 (0.00)	719 (0.00)	159 (0.07)	155 (0.07)	629 (0.00)	702 (0.00)	641 (0.00)	702 (0.00)	311 (0.03)
2k	706 (0.00)	705 (0.00)	778 (0.00)	717 (0.00)	778 (0.00)	68 (0.17)	338 (0.03)	682 (0.00)	755 (0.00)	694 (0.00)	755 (0.00)	65 (0.18)
4k	779 (0.00)	191 (0.07)	847 (0.02)	154 (0.08)	852 (0.00)	48 (0.28)	84 (0.15)	146 (0.08)	819 (0.00)	756 (0.02)	816 (0.02)	46 (0.27)
8k	898 (0.00)	443 (0.03)	970 (0.00)	908 (0.00)	962 (0.02)	40 (0.48)	111 (0.13)	867 (0.00)	939 (0.00)	876 (0.00)	938 (0.00)	38 (0.42)
16k	1110 (0.00)	363 (0.05)	1185 (0.00)	1109 (0.02)	386 (0.05)	30 (0.58)	67 (0.27)	181 (0.10)	1190 (0.02)	1127 (0.02)	1191 (0.00)	49 (0.38)

n	Biased											
	PLAIN+						CHAIN+					
	E1	**E2**	**E3**	**E4**	**E5**	**E6**	**E1**	**E2**	**E3**	**E4**	**E5**	**E6**
1k	147 (0.10)	8 (1.00)	78 (0.20)	926 (0.00)	79 (0.20)	23 (0.82)	23 (0.82)	8 (1.00)	26 (0.82)	876 (0.00)	152 (0.10)	23 (0.77)
2k	69 (0.25)	8 (1.00)	78 (0.30)	290 (0.07)	79 (0.27)	27 (0.95)	27 (0.95)	8 (1.00)	31 (0.97)	1119 (0.02)	74 (0.30)	29 (0.97)
4k	114 (0.20)	8 (1.00)	152 (0.18)	1541 (0.00)	154 (0.18)	10 (1.00)	33 (0.83)	8 (1.00)	37 (0.95)	1510 (0.02)	198 (0.13)	9 (1.00)
8k	217 (0.15)	8 (1.00)	86 (0.37)	2010 (0.00)	341 (0.10)	40 (0.98)	41 (0.95)	8 (1.00)	42 (0.98)	518 (0.07)	352 (0.10)	9 (1.00)
16k	165 (0.28)	8 (1.00)	172 (0.32)	1451 (0.03)	383 (0.13)	10 (1.00)	51 (0.97)	8 (1.00)	12 (1.00)	3229 (0.02)	305 (0.17)	10 (1.00)

Em uma análise geral da tabela, percebe-se que as técnicas sem *biasing* têm muito mais dificuldade em expor os erros comparadas às técnicas que utilizam *biasing*, o que entra em concordância com o observado nos dados de cobertura. Considerando apenas os geradores PLAIN- e CHAIN-, a maior diferença encontra-se no erro E1, o qual é estimulado por conflitos intra-processador, propriedade que é estimulada com a categoria 0 das cadeias de CHAIN. Essa diferença na eficácia e esforço se mantém mesmo com a aplicação do *biasing* (entre PLAIN+ e CHAIN+). No restante dos cenários, CHAIN- sofre pequenas degradações em eficácia e esforço em comparação com PLAIN-, o que também se correlata com o observado na cobertura.

Ao analisar-se a segunda metade da tabela (*Biased*), percebe-se

algumas informações mais interessantes. O erro E2 necessita que uma leitura ganhe acesso exclusivo ao bloco, o que só acontece quando nenhuma outra cache privativa tem aquele bloco válido. Sua detecção foi trivializada. Em todos os cenários, tanto PLAIN+ quanto CHAIN+ foram capazes de detectá-lo. Isso acontece devido ao aumento da frequência de *evictions* forçado pelo *biasing*: com mais *evictions* acontecendo, maior é a chance de um bloco estar inválido em todas as caches privativas.

Os erros E3 e E6, embora dependam diretamente da ocorrência de *evictions* no nível L0, para que o erro seja descoberto são necessárias outras operações de leitura sobre os endereços. Isso explica o porquê de sua detecção não ter sido trivializada como o erro E2. A vantagem que CHAIN+ exibe sobre PLAIN+ é explicada pelo uso das cadeias: elas facilitam a ocorrência da sequência de operações necessárias para que os erros sejam detectados pelo *checker*. Já o erro E4 depende de conflitos inter-processador. Por fim, o erro E5 não depende especificamente de conflitos, mas sim de interações entre diferentes processadores, algo que o *chaining* procura aumentar. Em poucos casos CHAIN+ apresentou esforço e eficácia inferiores aos de PLAIN+, o que indica que o uso inteligente dessas técnicas pode levar a melhores resultados do que simplesmente aplicar a geração de testes pseudoaleatórios com restrições convencionais.

6 CONCLUSÕES

Esta monografia contribuiu para a avaliação de quatro geradores de testes para expor erros de projeto no subsistema de memória compartilhada: um gerador de testes randômicos com restrições convencionais e três geradores construídos com as técnicas descritas em (ANDRADE; GRAF; SANTOS, 2016) e (ANDRADE, 2017). A principal contribuição do trabalho reportado nesta monografia foi a proposta de uma métrica de cobertura funcional, sua implementação e a medida experimental de seu valor para diferentes geradores sob diferentes parâmetros de geração.

Uma contribuição secundária, mas crucial para a avaliação da eficácia e do esforço de verificação foi a elaboração de erros artificiais de projeto para desafiar os geradores. Embora mais de uma centena de erros tenha sido desenvolvida em colaboração com um doutorando, apenas alguns foram selecionados para ilustrar esta monografia. A síntese de novos erros demanda um conhecimento profundo e detalhado do protocolo de coerência. Essa experiência deixou clara a necessidade de que propriedades específicas do subsistema de memória (tais como princípios gerais de coerência e regras de consistência) sejam exploradas para melhorar a qualidade dos testes, norteando assim o desenvolvimento de novos geradores, ao invés do simples uso de meta-heurísticas de propósitos gerais.

Apesar de a métrica de cobertura proposta basear-se na FSM local de cada controlador de cache, o nível de informação que ela provê parece adequado para servir de diretriz para a geração de testes. A correlação entre os resultados de cobertura, esforço e eficácia tornam mais sólidas as conclusões em relação às propriedades dos programas de testes, o que pode ser explorado no desenvolvimento de um algoritmo adaptativo para geração dirigida de testes. Além disso, ao contrário de trabalhos (WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012) que realizam verificação orientada por protocolo, onde é necessária uma análise do protocolo de coerência utilizado para quebrá-lo em subespaços (QIN; MISHRA, 2012) ou criar uma modelagem de máquina de estados dicotômica (WAGNER; BERTACCO, 2008), a adoção da cobertura de transições não prejudica o reuso dessa métrica para variantes do mesmo protocolo ou diferentes protocolos de coerência. Sendo assim, a utilização da métrica proposta para dirigir um novo gerador não limita a reusabilidade do checker, que é consequência de se usar um modelo de memória baseado em axiomas independentes do protocolo e da implementação específica. Em resumo, a métrica proposta parece ser adequada para a

geração dirigida de testes a serem usados na verificação de projetos cuja correção será avaliada através da aderência a um modelo de memória.

6.1 TRABALHOS FUTUROS

Como tópico de investigação científica futura pretende-se propor um gerador de testes dirigidos que use as técnicas avaliadas dentro de um laço de realimentação através do qual, dependendo da cobertura medida, parâmetros do programa (n, s) e restrições de endereços sejam gerados automaticamente. Os resultados de cobertura reportados nesta monografia deixam claro que CHAIN+ deve ser escolhido para um módulo *gerador* e que um módulo *diretor* deve ser desenvolvido levando em conta o aprendizado obtido neste trabalho.

Para que CHAIN+ seja usado como módulo gerador desse futuro gerador, é desejável ampliar a sua validação experimental, expandindo ainda mais a faixa de valores para os parâmetros. Por exemplo, outros *cbs* precisam ser testados para quantificar melhor o impacto desse parâmetro na cobertura de diferentes *tipos* de transições. Ou seja, é desejável avaliar a complementariedade de se usar diferentes *cbcs* para alcançar maiores níveis de cobertura, reduzindo a cobertura de transições induzidas por *evictions* para permitir que sejam tomadas transições induzidas por outros tipos de eventos, aumentando assim a cobertura cumulativa. Para isso, seria útil uma análise mais detalhada de quais transições foram efetivamente estimuladas para identificar quais transições os testes não são capazes de estimular.

Por fim, a validação da hipótese assumida no Capítulo 2 também será objeto de trabalho futuro, auxiliada pela ampliação já mencionada dos resultados experimentais.

REFERÊNCIAS

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: *IEEE 34th 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, Aug 2011. ISSN 0163-5964.

CHEN, Y. et al. Fast Complete Memory Consistency Verification. In: *IEEE Int. Symposium on High Performance Computer Architecture (HPCA)*. [S.l.: s.n.], 2009. p. 381–392.

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.

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.

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.

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.

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

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

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>.

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.

RAMBO, E.; HENSCHEL, O.; SANTOS, L. dos. Automatic generation of memory consistency tests for chip multiprocessing. In: *IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS)*. [S.l.: s.n.], 2011. p. 542–545.

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.

SHACHAM, O. et al. Verification of chip multiprocessor memory systems using a relaxed scoreboard. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. [S.l.: s.n.], 2008. p. 294–305. ISBN 978-1-4244-2836-6.

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 – Protocolo MESI

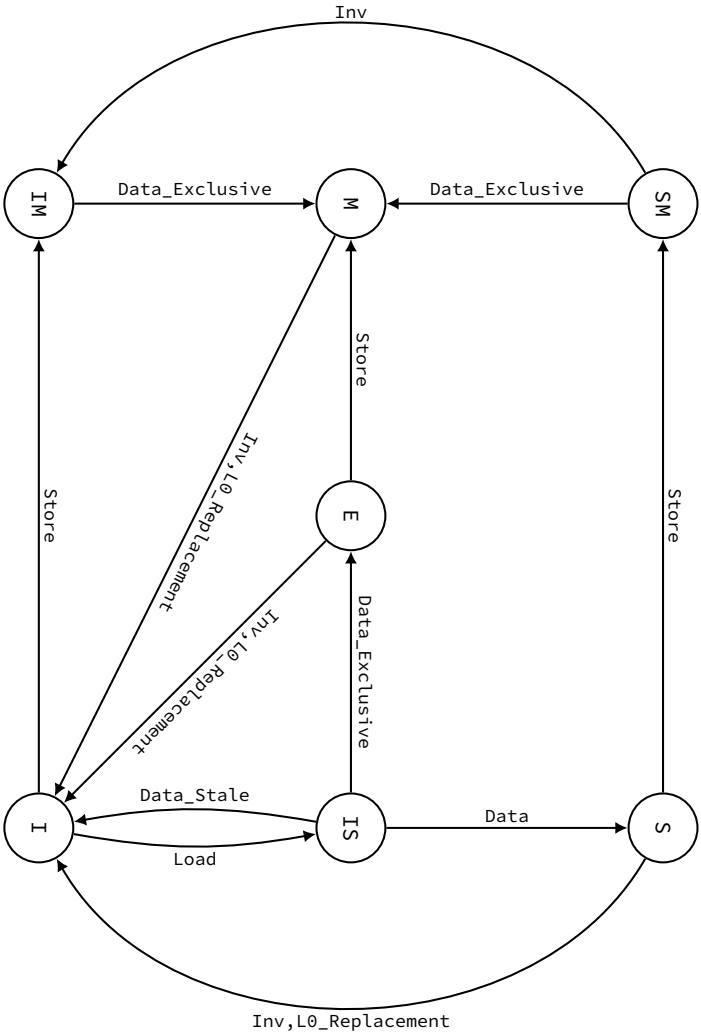


Figura 3 – FSM do controlador de cache do nível L0

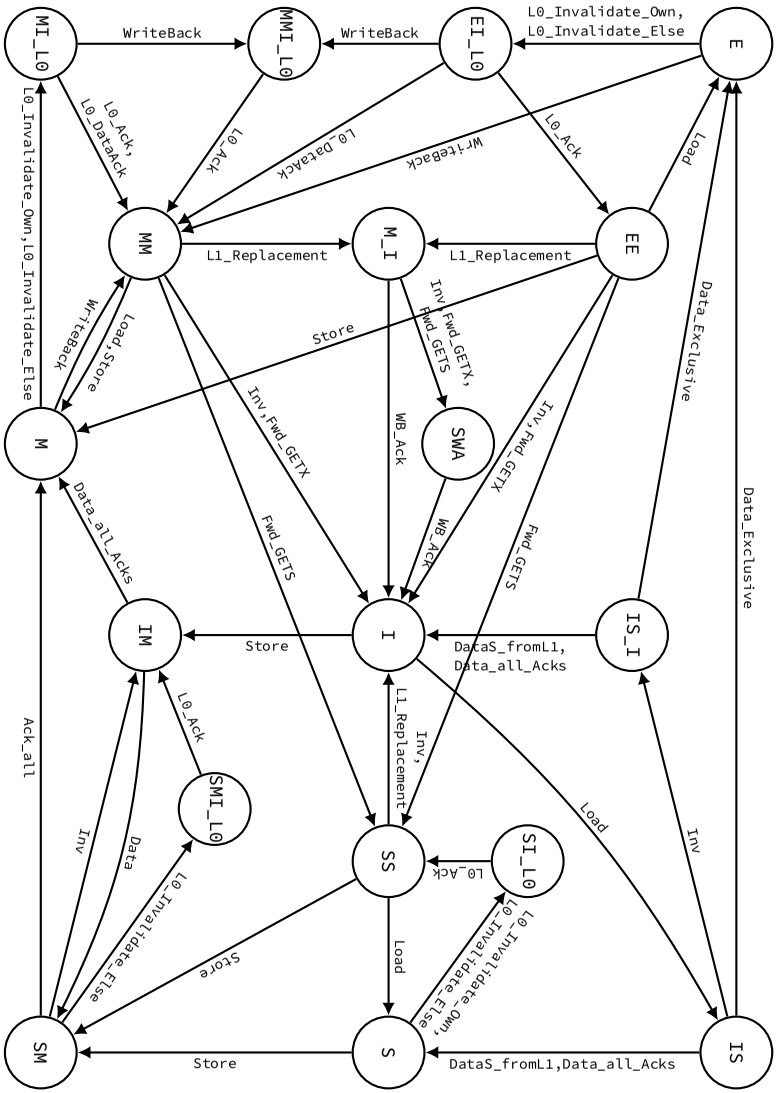


Figura 4 – FSM do controlador de cache do nível L1

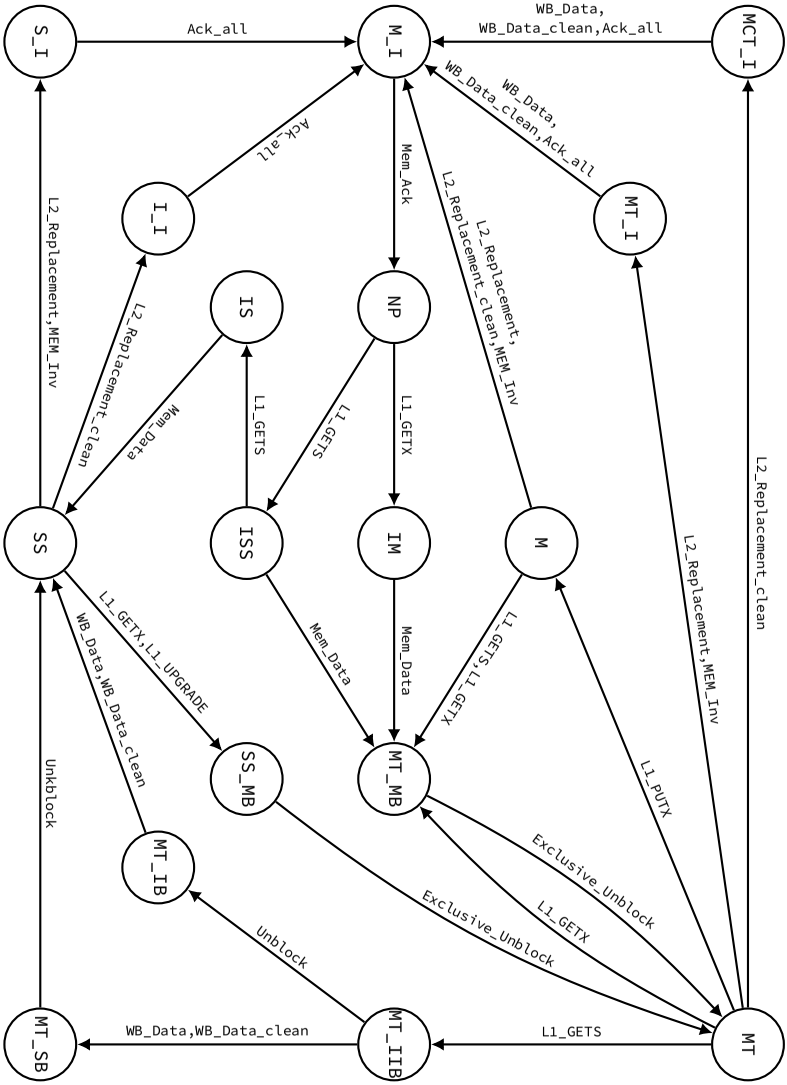


Figura 5 – FSM do controlador de cache do nível L2

ANEXO B – Código-fonte

gem5/src/monitors/coverage_logger.hh

```
/*
 * coverage_logger.hh
 *
 *  Created on: November, 01, 2016
 *      Author: Marleson <aszdrick> Graf
 */

#ifndef MEMORY_MODEL_COVERAGE_LOGGER_HH
#define MEMORY_MODEL_COVERAGE_LOGGER_HH

#include <fstream>
#include <map>

#include "mem/ruby/common/Address.hh"
#include "monitors/monitor.hh"
#include "params/CoverageLogger.hh"
#include "sim/sim_object.hh"

namespace MemoryModel {
    class CoverageLogger : public SimObject {
        // SimObject-compatible parameters
        using Params = CoverageLoggerParams;
     public:
        // SimObject-compatible constructor
        CoverageLogger(const Params*);
        // For a given cache block, event and current state,
             logs this transition
        void log(Address, const std::string&, const std::
            string&);
        // Indicates if logger is enabled
        bool enabled() const { return enable; }
        // SimObject ugly stuff
        const Params *params() const {
            return dynamic_cast<const Params*>(_params);
        }
        void finalize();
     private:
        // Identifier of cache (namely L0, L1 or L2)
        std::string cache_id;
        // Identifier of processor (e.g. 0, 1, 2...)
        Monitor::ProcessorID pid;
        // Transition counters organized by address
        std::map<uint64_t, std::map<std::string, size_t>>
            addr_transitions;
        // Output file to register the logs
        std::ofstream output_file;
        bool initialized = false;
        // Defines if logger is enabled
        bool enable;

```

```
48          void initialize();
49          bool is_monitored(Addr);
50      };
51  }
52
53  #endif /* MEMORY_MODEL_COVERAGE_LOGGER_HH */
```

gem5/src/monitors/coverage_logger.cc

```
1
2   #include "coverage_logger.hh"
3
4   #include "base/callback.hh"
5   #include "mem/ruby/system/System.hh"
6   #include "sim/sim_exit.hh"
7
8   MemoryModel::CoverageLogger * CoverageLoggerParams::create()
        {
9       return new MemoryModel::CoverageLogger(this);
10  }
11
12  MemoryModel::CoverageLogger::CoverageLogger(const Params* p)
        :
13   SimObject(p),
14   cache_id(p->cache_id),
15   pid(p->pid),
16   enable(p->enable) {
17      if (enable) {
18          Callback* callback = new MakeCallback<
19              CoverageLogger, &CoverageLogger::finalize
20          >(this, true);
21          registerExitCallback(callback);
22      }
23  }
24
25  void MemoryModel::CoverageLogger::finalize() {
26      auto processor = std::to_string(pid);
27      auto fname = "m5out/cv_" + cache_id + "_p" + processor +
            ".log";
28
29      output_file.open(fname);
30      output_file << "[";
31
32      auto first = true;
33
34      for (const auto& pair : addr_transitions) {
35          if (!first) {
36              output_file << ",";
37          } else {
38              first = false;
39          }
```

```
40          output_file << "[\"0x" << std::hex << pair.first <<
                std::dec << "\", [";
41          auto inner_first = true;
42          for (auto counter : pair.second) {
43              if (!inner_first) {
44                  output_file << ",";
45              } else {
46                  inner_first = false;
47              }
48              output_file << "[\"" << counter.first << "\","
                    << counter.second << "]";
49          }
50          output_file << "]]";
51      }
52
53      output_file << "]" << std::endl;
54  }
55
56  void MemoryModel::CoverageLogger::initialize() {
57      auto& addrs = Monitor::getSharedAddresses();
58      for (auto addr : addrs) {
59          addr_transitions[addr] = {};
60      }
61  }
62
63  bool MemoryModel::CoverageLogger::is_monitored(Addr addr) {
64      return Monitor::isActive(pid)
65          && Monitor::getSharedAddresses().count(addr);
66  }
67
68  void MemoryModel::CoverageLogger::log(Address addr,
69                                        const std::string& s,
70                                        const std::string& e)
                                            {
71      if (enable) {
72          auto true_addr = addr.getAddress() | addr.
                getLowerBits();
73          if (is_monitored(true_addr)) {
74              if (!initialized) {
75                  initialize();
76                  initialized = true;
77              }
78              auto key = s + "+" + e;
79              if (addr_transitions[true_addr].count(key)) {
80                  addr_transitions[true_addr][key] += 1;
81              } else {
82                  addr_transitions[true_addr][key] = 1;
83              }
84          }
85      }
86  }
```

gem5/src/monitors/CoverageLogger.py

```
from m5.params import *
from m5.SimObject import SimObject

class CoverageLogger(SimObject):
    type = 'CoverageLogger'
    cxx_class = "MemoryModel::CoverageLogger"
    cxx_header = "monitors/coverage_logger/coverage_logger.
        hh"
    cache_id = Param.String("", "Target cache to log")
    pid = Param.Int(-1, "Target processor")
    enable = Param.Bool(False, "Enable logger")
```

Modificações em gem5/src/mem/protocol/RubySlicc_Types.sm

```
...
structure (MemoryModel::CoverageLogger, external = "yes",
    primitive = "yes") {

}
...
```

Modificações em gem5/src/mem/protocol/MESI_Three_Level-L0cache.sm

```
...
    MemoryModel::CoverageLogger * cv_logger,
...
```

Modificações em gem5/src/mem/protocol/MESI_Three_Level-L1cache.sm

```
...
    MemoryModel::CoverageLogger * cv_logger,
...
```

Modificações em gem5/src/mem/protocol/MESI_Two_Level-L2cache.sm

```
...
    MemoryModel::CoverageLogger * cv_logger,
...
```

Modificações em gem5/configs/ruby/MESI_Three_Level.py

```
...
            l0_cntrl.cv_logger = CoverageLogger(
                cache_id = "l0",
```

```
                pid = i * num_cpus_per_cluster + j,
                enable = options.log_cv == True
            )
...
            l1_cntrl.cv_logger = CoverageLogger(
                cache_id = "l1",
                pid = i * num_cpus_per_cluster + j,
                enable = options.log_cv == True
            )
...
            l2_cntrl.cv_logger = CoverageLogger(
                cache_id = "l2",
                pid = i * num_l2caches_per_cluster + j,
                enable = options.log_cv == True
            )
...
```

Modificações em gem5/src/mem/slicc/symb/StateMachine.py

```
...
        if ident == 'L0Cache' or ident == 'L1Cache' or ident
             == 'L2Cache':
            code('''
        if (m_cv_logger_ptr->enabled()) {
            auto cv_state = ${ident}_State_to_string(state);
            auto cv_event = ${ident}_Event_to_string(event);
            m_cv_logger_ptr->log(addr, cv_state, cv_event);
        }
''')
...
```

Modificações em gem5/configs/example/se.py

```
...
parser.add_option("--log_cv", action="store_true", default=
    False,
                  help="Activate coverage logging")
...
```

ANEXO C – Artigo

Geração de Testes de Memória Compartilhada Coerente: Uma Avaliação de Cobertura e Eficácia

Marleson Graf

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

marleson.graf@grad.ufsc.br

Resumo. *Multiprocessadores em chip (CMP) são desafiados pela crescente complexidade de seus subsistemas de memória compartilhada, o que torna seus projetos suscetíveis a erros. Para evitar que esses erros se propaguem até a fase de prototipação, técnicas de verificação são aplicadas sobre simulações do projeto. Programas de teste precisam ser executados sobre a plataforma simulada para estimular o sistema e expor potenciais erros. Nesse trabalho é proposta uma métrica de cobertura para expandir a avaliação dos programas de testes gerados por quatro diferentes técnicas de geração. Complementarmente, foram desenvolvidos novos erros de coerência, a partir dos quais criaram-se representações de projeto para avaliar o esforço e eficácia dos testes gerados pelas técnicas. Ao todo, foram executados 14400 programas de teste para a avaliação de cobertura e 86400 casos de uso para a avaliação de esforço e eficácia (6 erros $\times$ 14400 programas de testes), em arquiteturas de 8, 16 e 32 núcleos. Os resultados mostram que a combinação de duas diferentes técnicas de geração de testes leva aos melhores valores de cobertura e alcança os melhores resultados de esforço e eficácia na maiorias dos cenários de verificação.*

1. Introdução

Multiprocessadores em chip (CMP – *Chip Multiprocessors*), como o próprio nome sugere, são chips que contém múltiplos núcleos de processamento em seu interior. São amplamente utilizados em computadores pessoais e servidores, onde a programação de propósito geral conta com o uso de modelos de memória para abstrair detalhes do uso da hierarquia de memória. Em termos gerais, um modelo de memória restringe quais valores a leitura de uma posição de memória compartilhada pode retornar [Adve and Gharachorloo 1996]. Essas restrições são capturadas por regras de consistência, as quais abordam dois aspectos principais: o quanto a ordem de execução de operações de leitura e escrita para endereços de memória distintos pode ser relaxada, i.e., executada fora da ordem originalmente especificada, e o grau de atomicidade das operações de escrita para um mesmo endereço [Adve and Gharachorloo 1996]. O *hardware* subjacente garante que o comportamento do subsistema de memória seja compatível com o modelo adotado.

A busca por maior desempenho leva ao uso de modelos de memória que permitem ordens de execução mais relaxadas. Em uma execução sequencial, a ordem em que as operações são emitidas e executadas é a exatamente a mesma definida pelo programa. Ao relaxar a ordem de execução, otimizações podem ser exploradas na emissão de operações de leitura e escrita ao subsistema de memória. Embora essa relaxação abra a

possibilidade de resultados diferentes na execução de programas paralelos, o amplo uso de bibliotecas de sincronização acaba escondendo as regras de consistência da visão do programador [Hennessy and Patterson 2011], o que permite o uso de ordens mais relaxadas sem adicionar complexidade à programação. Para suportar emissões fora de ordem, *buffers* de leitura e escrita são necessários em cada processador e entre *caches* de diferentes níveis hierárquicos. Além disso, sistemas que utilizam *caches* privativas precisam manter cópias válidas de um mesmo dado em múltiplas *caches* ao longo da hierarquia. Para tanto, adotam-se protocolos de coerência baseados em diretório os quais, apesar da complexidade, acredita-se serem escaláveis a centenas de processadores [Martin et al. 2012]. Essa complexidade, intensificadas pelo aumento do número de processadores em um único *chip*, faz com que projetos de multiprocessadores sejam suscetíveis a erros em sua especificação, tornando a aplicação de técnicas de verificação funcional fundamentais para validar o projeto sob desenvolvimento.

As técnicas de verificação dependem da execução de programas de teste sobre o projeto sendo testado. Alguns verificadores permitem o uso de programas reais, enquanto outros utilizam programas pseudo-aleatórios com valores pré-determinados para as leituras e escritas, a fim de facilitar a análise do verificador [Freitas et al. 2013]. Tem sido observado tanto em ambiente acadêmico [Shacham et al. 2008] quanto em ambiente industrial [Hangal et al. 2004] que testes devem ser programas curtos, paralelos, com condições de corrida para um número pequeno de posições de memória, visto que condições de corrida agressivas tendem a expor erros em multiprocessadores mais rápido [Manovit and Hangal 2006]. Como a verificação pré-silício pode ser ordens de magnitude mais lenta que um protótipo em *hardware*, um trabalho recente [Andrade et al. 2016] buscou tornar a geração pseudo-aleatória mais eficiente, baseando em uma especificação formal de cadeias canônicas de dependência de dados [Gharachorloo 1995]. Um trabalho ainda mais recente [Andrade 2017] estende a ideia, propondo um mecanismo que explora a escolha dos endereços de memória utilizados pelos testes. Outros trabalhos propõem técnicas de geração de testes adaptativa [Wagner and Bertacco 2008, Elver and Nagarajan 2016], de forma a guiar a criação de um novo teste com base nas informações dos testes anteriores e, assim, diminuir o esforço de verificação. [Qin and Mishra 2012] propõe uma técnica dirigida pela cobertura do produto das máquinas de estados (FSMs) do protocolo de coerência utilizado.

2. Trabalhos correlatos

Geradores de testes pseudo-aleatórios para modelos de memória vêm sendo utilizados tanto por verificadores pós-silício [Hangal et al. 2004, Manovit and Hangal 2006, Roy et al. 2006, Hu et al. 2012], no meio industrial, quanto por verificadores pré-silício [Shacham et al. 2008, Freitas et al. 2013] em ambiente acadêmico. Verificadores pós-silício contam com a alta vazão de instruções do protótipo em *hardware*, permitindo a execução de testes maiores para estimular o sistema. O mesmo não é verdade para testes pré-silício, os quais são executados em simuladores do *hardware*. Para tratar esse problema, um trabalho recente [Andrade et al. 2016] baseou-se em especificações formais [Gharachorloo 1995] para produzir programas apenas com sequências de instruções significativas ao teste do modelo de memória. Uma extensão desse trabalho [Andrade 2017] apresenta uma técnica para restringir os endereços de memória associados aos programas de teste, permitindo controlar a ocorrência de substituições de blocos de *cache*.

Mesmo com refinamentos sobre a produção de testes pseudo-aleatórios, a configuração dos parâmetros de geração continua sendo responsabilidade do engenheiro de verificação. Nem sempre é trivial definir quais os melhores parâmetros para alcançar limites satisfatórios de cobertura para o projeto sob verificação. Diferentes técnicas dirigidas [Wagner and Bertacco 2008, Qin and Mishra 2012, Elver and Nagarajan 2016] foram propostas para tratar esse problema. Um desses trabalhos [Qin and Mishra 2012], propõe um algoritmo dirigido pela cobertura de transições do protocolo de coerência de *cache* adotado pelo sistema a ser testado. Ele decompõe o espaço de estados da máquina produto do protocolo em componentes estruturalmente mais simples (cliques e hipercubos). Com base nesses componentes, o trabalho apresenta um método *on-the-fly* (dinâmico) de geração de testes dirigidos baseado em caminho euleriano.

Outros trabalhos [Wagner and Bertacco 2008, Elver and Nagarajan 2016] propõem geradores adaptativos de testes baseados no uso de meta-heurísticas para definir os próximos testes a serem gerados. Um deles, MCJammer [Wagner and Bertacco 2008], é uma ferramenta de verificação adaptativa para projetos de multiprocessadores que utiliza um ciclo fechado de *feedback* para dinamicamente ajustar a simulação para efetivamente estimular *corner cases*. A técnica baseia-se no uso de uma rede de agentes cooperantes, cada agente associado a um núcleo de processamento. Outro gerador adaptativo proposto é o McVerSi [Elver and Nagarajan 2016], uma ferramenta baseada em algoritmo genético para guiar a geração. A cobertura estrutural (código-fonte) do protocolo de coerência é utilizada como função-objetivo (*fitness function*). Para a função de *crossover*, é proposta uma técnica de cruzamento seletivo, a qual favorece operações de memória envolvidas em condições de corrida.

Apesar de geradores dirigidos garantirem plena cobertura global explorando propriedades do espaço de verificação [Qin and Mishra 2012], eles têm como grande desvantagem sua dependência ao protocolo específico de coerência do projeto sob verificação. Portanto, não são reusáveis para projetos que utilizem protocolos diferentes, requerendo uma nova análise da FSM associada ao protocolo e subsequente divisão da mesma em componentes adequados para a travessia euleriana. Por outro lado, geradores adaptativos baseados no uso de meta-heurísticas para maximizar a cobertura [Wagner and Bertacco 2008, Elver and Nagarajan 2016] não exploram propriedades do espaço global de verificação, o que torna seus algoritmos complexos ou requer o uso de estimativas simples de cobertura (para que possam ser obtidas *on-the-fly*).

3. Os geradores de testes

Quatro geradores diferentes foram utilizados para a geração de testes. PLAIN- refere-se a um gerador pseudo-aleatório convencional, similar aos utilizados em outros trabalhos de verificação de modelos de memória [Hangal et al. 2004, Shacham et al. 2008]. Um protótipo foi construído com base no pseudocódigo reportado na literatura [Rambo et al. 2011]. Nesse gerador, operações de leitura, escrita e barreiras de memória são distribuídas aleatoriamente entre as diferentes *threads* de execução, uma para cada núcleo de processamento. Todas as *threads* possuem o mesmo número de instruções e os endereços sobre os quais as operações são realizadas também são escolhidos aleatoriamente. Cada operação de escrita utiliza um valor diferente das outras para fins de verificação.

O segundo gerador, denominado PLAIN+, é similar ao PLAIN-, diferenciando-se na forma como os endereços de memória são escolhidos. Nesse gerador, a escolha dos endereços de memória compartilhados é realizada por uma técnica [Andrade 2017] onde os parâmetros especificados definem quais padrões de bits são aceitáveis para a formação dos endereços. Nesse sentido, são controlados os índices, *tags* e partes do *offset* para garantir que os parâmetros sejam satisfeitos. A maior vantagem desse mecanismo é a capacidade de controlar o nível de competição dos blocos de memória por uma mesma linha (ou conjunto associativo) de cache, o que permite controlar a ocorrência de *evictions*.

O terceiro gerador, CHAIN- [Andrade et al. 2016], utiliza a noção de cadeias canônicas de dependência [Gharachorloo 1995] para guiar a geração das sequências de instruções. Essa técnica procura diminuir o espaço de geração de testes, reduzindo a ocorrência de sequências de operações redundantes para a verificação do modelo de memória. A implementação utilizada foi especificamente projetada com base em quatro cadeias definidas para o modelo de memória do processador Alpha, através das quais definem-se sequências de operações intra-processador (categoria 0) e inter-processador (categorias 1, 2 e 3). O quarto gerador, CHAIN+, é composto pelo gerador CHAIN- e a técnica de escolha de endereços dirigida.

Os quatro geradores possuem alguns parâmetros de geração em comum: o número de *threads* (p), uma para cada processador do projeto sendo testado, o tamanho do programa de teste (n), o número de posições de memória compartilhadas (s) e um número para inicializar o gerador de números pseudoaleatórios utilizado ($seed$). Com base nos três primeiros parâmetros, define-se o *cenário de verificação* como o conjunto de testes que compartilham uma mesma configuração $v = (p, n, s)$.

4. Os erros de projeto

A criação de erros artificiais é necessária para colocar os geradores à prova, verificando sua capacidade de expor erros. Cada erro criado é injetado em um caso de projeto separado, de forma que, quando ocorre uma detecção, é possível afirmar que esta foi relativa apenas ao erro criado. A Tabela 1 apresenta uma seleção de 6 erros de projeto (nomeados de E1 a E6). Nela são identificados o nível em que o erro foi criado, o estado atual da FSM, o evento que dispara a transição, o estado destino da transição e, por fim, se houve alguma ação de saída removida. Os erros foram criados a partir de alterações nas máquinas de estados do protocolo de coerência. Existem dois tipos de alterações: a mudança do estado destino da transição – representado na tabela pelo uso da expressão *instead of* na coluna "Próximo estado" – e a remoção de uma ou mais ações de saída de uma transição – informado na coluna "Ação de saída removida".

Tabela 1. Erros de projeto selecionados

ID	Nível	Estado atual	Evento de entrada	Próximo estado	Ação de saída removida
E1	L0	E	Store	E instead of M	nenhuma
E2	L1	IS	Data_Exclusive	E	writeDataFromL2Response
E3	L1	E	WriteBack	MM	writeDataFromL0Request
E4	L1	IS_I	DataS_fromL1	I	writeDataFromL2Response
E5	L1	EE	Fwd_GETX	SS	data block of sendDataToRequestor
E6	L1	M_IL0	WriteBack	MM_IL0	writeDataFromL0Request

5. A métrica de cobertura

Dado um projeto livre de erros, a *cobertura funcional* é medida como a fração de transições cobertas na máquina de estados de cada controlador de cache. São rastreados todos os blocos de memória referenciados pela *coleção de testes* pseudoaleatórios de um dado cenário de verificação $v = (p, n, s)$ (como definido na Seção 3). Para cada bloco referenciado correspondente a uma posição de memória a sob a perspectiva da cache de um processador i no nível L, determinam-se quantas transições *distintas* foram exercitadas na respectiva máquina de estados do protocolo como resultado da execução da coleção de testes. Dessa forma, obtém-se o número cumulativo de transições cobertas, denotado como $TRAN_a^{i,L}(v)$.

Seja $total(L)$ o número total de transições da máquina vinculada ao nível L. Computa-se a *cobertura de transições* como $TC_a^{i,L}(v) = TRAN_a^{i,L}(v)/total(L)$. Em seguida, obtém-se a distribuição de coberturas de transições induzidas por v no nível L, denotado como $TC(v, L)$, i.e., a distribuição de $TC_a^{i,L}(v)$ sobre todos os processadores (i) e todas as posições de memória (a). Também foi obtida uma distribuição simular no nível compartilhado de cache (LLC), denotado como $TC(v, L2)$. Por fim, para cada nível L, são definidas a *mediana*, o valor *mínimo* e o valor *máximo* de $TC(v, L)$ sobre a coleção de todos os cenários v, escritos $\widehat{TC}(L)$, $TC^{min}(L)$ e $TC^{max}(L)$, respectivamente.

6. Avaliação experimental

Para a simulação foi adotada uma infraestrutura de domínio público, o *gem5 simulator* [Binkert et al. 2011], configurado com um modelo de CPU *out-of-order* (O3). Além disso, foi utilizado o conjunto de instruções (ISA) do SPARC e o modelo de memória do processador Alpha [Gharachorloo 1995]. A hierarquia de memória utilizada consiste em caches L0 privativas (separadas em dados e instruções), caches L1 privativas (unificadas), e uma cache L2 compartilhada, com tamanhos de 4 KiB (mapeamento direto), 64 KiB (2-*way*) e 2 MiB (8-*way*), respectivamente. Todos os níveis operam com o mesmo tamanho de bloco (64 bytes) e utilizam a mesma política de substituição (LRU). Os experimentos para a análise de cobertura, realizados em um projeto livre de erros, totalizam 14400 testes, enquanto os experimentos para a análise de eficácia e esforço totalizam 86400 testes (6 erros $\times$ 14400 testes). Por conveniência, serão utilizados os termos *chaining* e *biasing* para se referir as técnicas de CHAIN (presentes em CHAIN- e CHAIN+) e da atribuição de endereços sob restrições (presente em PLAIN+ e CHAIN+), respectivamente.

6.1. Parâmetros de geração

Foram simuladas arquiteturas de 8, 16 e 32 núcleos (parâmetro p) para a execução dos testes. Para cada arquitetura, foram gerados programas com quinze sementes aleatórias (1, 2, ..., 15), quatro quantias diferentes de posições de memória compartilhadas (s = 4, 8, 16 e 32), e cinco tamanhos de teste (n = 1k, 2k, 4k, 8k, 16k), onde k representa 2^{10} operações. Além disso, os geradores PLAIN- e PLAIN+ aceitam como parâmetro um *mix* de instruções o qual determina a porcentagem aproximada de cada tipo de instrução, enquanto os geradores CHAIN- e CHAIN+ aceitam um *mix* de categorias, determinando o quanto de cada categoria é aplicado na geração do programa de teste. Foram usados quatro *mixes* diferentes para cada gerador, apresentados na Tabela 2.

Os geradores que utilizam *biasing* também recebem três parâmetros para o módulo de atribuição de endereços. O parâmetro abc (*alignment biasing constraint*) especifica que

Tabela 2. *Mixes* utilizados

***Mix* de instruções**			***Mix* de categorias**			
Load	Store	Membar	$\mathcal{C}_0$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$
0.30	0.66	0.04	0.40	0.60	0.00	0.00
0.48	0.48	0.04	0.00	1.00	0.00	0.00
0.66	0.30	0.04	0.00	0.80	0.20	0.00
0.80	0.16	0.04	0.00	0.80	0.00	0.20

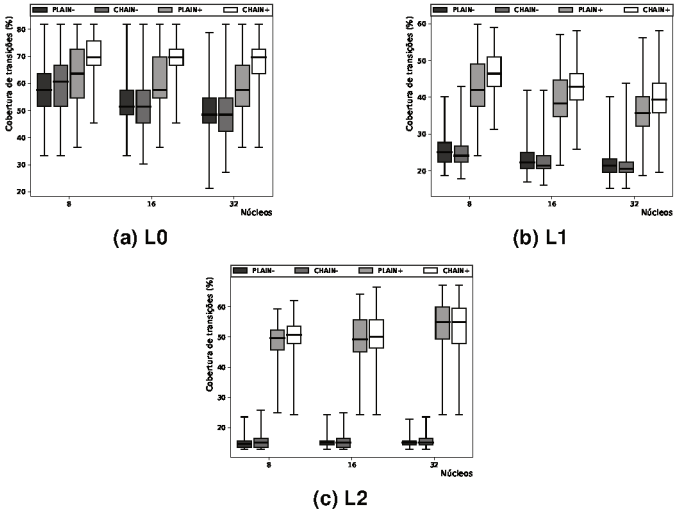
o endereço é alinhado a 2^{abc} bytes. O parâmetro sbc (*sharing biasing constraint*) é um valor booleano que garante *true sharing* ou não. O último parâmetro, cbc (*competition biasing constraint*) é um par (κ, χ), onde κ indica o número de linhas de cache distintas para onde são mapeadas as posições compartilhada usadas no teste e χ indica o máximo de competição por uma mesma linha de cache. Para a geração dos experimentos, foram adotados $abc = 2^6$, *true sharing* ($sbc = true$) e $cbc = (1, s)$. Esse cbc foi escolhido por gerar o máximo possível de *evictions* para a os valores usados de s.

6.2. Análise de cobertura

A Figura 1 sumariza os resultados de cobertura na forma de diagramas de caixa representando a cobertura para cada arquitetura testada (8, 16 e 32 núcleos) em cada nível da hierarquia. Os *whiskers* representam o mínimo e o máximo dos dados, de forma que os gráficos apresentam $\widehat{TC}(L), TC^{min}(L), TC^{max}(L)$ e a dispersão da cobertura. Os gráficos mostram que a cobertura mediana de CHAIN- não melhora em relação a de PLAIN- (com exceção do nível L0 da arquitetura de 8 núcleos). Entretanto, *biasing* melhora a cobertura de forma significativa nos níveis hierárquicos mais baixos (L1 e L2). O gerador CHAIN+, alcançou os maiores valores de cobertura mediana para todos os níveis e todos os números de núcleos, exceto o L2 de 32 núcleos, no qual a mediana empata com PLAIN+. Além disso, CHAIN+ alcançou os maiores valores de cobertura máxima e mínima, com exceção dos projetos de 8 núcleos nos níveis L1 e L2. Isso é uma primeira evidência de que as duas técnicas têm uma natureza complementar: *biasing* causa mais impacto no último nível da hierarquia (L2), enquanto *chaining* tem mais impacto no primeiro nível (L0).

As técnicas de *chaining* e *biasing* estimulam operações conflitantes e eventos de *eviction*, respectivamente. Esses tipos diferetnes de eventos induzem transições *distintas* na máquina de estados de caches privativas, contribuindo para o aumento da cobertura com o gerador CHAIN+. O *biasing* estimula transições relativas à *evictions* em todos os níveis. No nível L0 (Figura 1a), *biasing* também estimula transições a partir do estado I, enquanto *chaining* estimula transições dos estados E, S e M, seja através de conflitos intra-processador (transições (S,M) e (E, M)) ou através de conflitos inter-processador (transições (M,I), (E,I), e (S,I)). No nível L1 (Figura 1b), *biasing* estimula transições induzidas por requisições GETS e GETX do nível L0, enquanto *chaining* estimula transições induzidas não somente por requisições UPG do nível L0, como também requisições de outros núcleos (Invalidate, FWD_GETS, FWD_GETX). Nota-se que o impacto das técnicas quando combinadas é maior na L0 que na L1. Uma causa para isso é o uso de caches inclusivas, onde operações conflitantes, apesar de gerarem *diferentes* tipos de requisições, disparam a *mesma* transição na L1. Outra causa é a frequência menor de *evictions* devido

Figura 1. Distribuições de cobertura para projetos de 8, 16 e 32 núcleos



a maior associatividade na L1. No nível L2 (Figura 1c), a cobertura mediana foi cerca de 15% para PLAIN- e CHAIN- e 53% para *ambos* PLAIN+ e CHAIN+. Isso é mais uma evidência de que o impacto de *chaining* é marginal no último nível de cache. A explicação deve-se ao fato de que a maioria das requisições resultantes de conflitos inter-processador (Invalidate, FWD_GETS, FWD_GETX) não induzem transições na L2: elas são, na verdade, ações de saída. Apenas 4 das 140 transições do nível L2 são induzidas por conflitos intra-processador (UPG) ou inter-processador (GETX, GETS). Por isso, a maioria das transições no nível L2 são estimuladas apenas pelo *biasing*.

6.3. Análise de eficácia e esforço

Para avaliar a *eficácia* de um gerador em expor um dado tipo de erro sob um cenário $v = (p, n, s)$, foi medida a fração $\varepsilon(v)$ de todos os testes induzidos por v para os quais violações foram detectadas. Ao assumir-se uma amostragem suficientemente grande, essa fração pode ser interpretada como a probabilidade de um gerador de expor aquele tipo de erro no cenário v. A métrica de esforço foi definida em trabalho correlato [Andrade 2017], e utiliza uma combinação da eficácia e do tempo médio de execução dos testes de um cenário v. Nos casos em que não houve detecção, o esforço no cenário foi calculado diretamente pela soma de todos os tempos de execução dos testes. Caso contrário, o esforço médio no cenário foi calculado como $(\lceil 1/\varepsilon(v) \rceil - 1)\ \widehat{t^0(v)} + \widehat{t^1(v)}$, onde $\varepsilon(v)$ é a eficácia medida para o cenário v, $\widehat{t^0(v)}$ é o tempo médio de execução dos testes que não expuseram o erro e $\widehat{t^1(v)}$ é o tempo médio dos testes que expuseram. A Tabela 3 mostra se um gerador foi ou não capaz de expor um erro com um dado tamanho de teste em um projeto de 32 núcleos (utilizando 32 variáveis compartilhadas). Entradas com o fundo preto indicam que naquele cenário nenhum teste foi capaz de expor o erro. Ela também

mostra o esforço gasto (expresso em segundos) para expor um erro ou tentando expô-lo. Além disso, a eficácia de cada cenário é mostrada entre parênteses abaixo do esforço, e a tabela tem uma divisão entre *unbiased* e *biased*, referindo-se aos geradores que não utilizam e que utilizam *biasing*, respectivamente.

Tabela 3. Esforço e eficácia para projetos de 32 núcleos e 32 variáveis compartilhadas

n	Unbiased											
	PLAIN-						CHAIN-					
	E1	E2	E3	E4	E5	E6	E1	E2	E3	E4	E5	E6
1k	320 (0.03)	645 (0.00)	719 (0.00)	658 (0.00)	719 (0.00)	159 (0.07)	155 (0.07)	629 (0.00)	702 (0.00)	641 (0.00)	702 (0.00)	311 (0.03)
2k	706 (0.00)	705 (0.00)	778 (0.00)	717 (0.00)	778 (0.00)	68 (0.17)	338 (0.03)	682 (0.00)	755 (0.00)	694 (0.00)	755 (0.00)	65 (0.18)
4k	779 (0.00)	191 (0.07)	847 (0.02)	154 (0.08)	852 (0.00)	48 (0.28)	84 (0.15)	146 (0.08)	819 (0.00)	756 (0.02)	816 (0.02)	46 (0.27)
8k	898 (0.00)	443 (0.03)	970 (0.00)	908 (0.00)	962 (0.02)	40 (0.48)	111 (0.13)	867 (0.00)	939 (0.00)	876 (0.00)	938 (0.00)	38 (0.42)
16k	1110 (0.00)	363 (0.05)	1185 (0.00)	1109 (0.02)	386 (0.05)	30 (0.58)	67 (0.27)	181 (0.10)	1190 (0.02)	1127 (0.02)	1191 (0.00)	49 (0.38)

n	Biased											
	PLAIN+						CHAIN+					
	E1	E2	E3	E4	E5	E6	E1	E2	E3	E4	E5	E6
1k	147 (0.10)	8 (1.00)	78 (0.20)	926 (0.00)	79 (0.20)	23 (0.82)	23 (0.82)	8 (1.00)	26 (0.82)	876 (0.00)	152 (0.10)	23 (0.77)
2k	69 (0.25)	8 (1.00)	78 (0.30)	290 (0.07)	79 (0.27)	27 (0.95)	27 (0.95)	8 (1.00)	31 (0.97)	1119 (0.02)	74 (0.30)	29 (0.97)
4k	114 (0.20)	8 (1.00)	152 (0.18)	1541 (0.00)	154 (0.18)	10 (1.00)	33 (0.83)	8 (1.00)	37 (0.95)	1510 (0.02)	198 (0.13)	9 (1.00)
8k	217 (0.15)	8 (1.00)	86 (0.37)	2010 (0.00)	341 (0.10)	40 (0.98)	41 (0.95)	8 (1.00)	42 (0.98)	518 (0.07)	352 (0.10)	9 (1.00)
16k	165 (0.28)	8 (1.00)	172 (0.32)	1451 (0.03)	383 (0.13)	10 (1.00)	51 (0.97)	8 (1.00)	12 (1.00)	3229 (0.02)	305 (0.17)	10 (1.00)

Em uma análise geral da tabela, as técnicas sem *biasing* mostram muito mais dificuldade em expor os erros comparadas às técnicas que utilizam *biasing*, o que correlata com o observado nos dados de cobertura. Considerando apenas os geradores PLAIN- e CHAIN-, a maior diferença encontra-se no erro E1, o qual é estimulado por conflitos intra-processador, propriedade que é estimulada com a categoria 0 das cadeias de CHAIN. Essa diferença na eficácia e esforço se mantém mesmo com a aplicação do *biasing*. No restante dos cenários, CHAIN- sofre pequenas degradações em eficácia e esforço em comparação com PLAIN-, o que também se correlata com o observado na cobertura. Na segunda metade da tabela (*Biased*), percebe-se que o erro E2 foi trivializado. Isso ocorre devido ao aumento da frequência de *evictions* forçado pelo *biasing*: isso aumenta a chance de um bloco estar inválido em todas as caches privativas, o que permite que a próxima leitura adquira acesso exclusivo, cenário necessário para a exposição desse erro.

Os erros E3 e E6, embora dependam diretamente da ocorrência de *evictions* no nível L0, para que o erro seja descoberto são necessárias outras operações de leitura sobre os endereços. Isso explica o porquê de sua detecção não ter sido trivializada como o erro E2. A vantagem que CHAIN+ exibe sobre PLAIN+ é explicada pelo uso das cadeias: elas facilitam a ocorrência da sequência de operações necessárias para que os erros sejam

detectados pelo *checker*. Já o erro E4 depende de conflitos inter-processador. Por fim, o erro E5 não depende especificamente de conflitos, mas sim de interações entre diferentes processadores, algo que o *chaining* procura aumentar. Em poucos casos CHAIN+ apresentou esforço e eficácia inferiores aos de PLAIN+, o que indica que o uso inteligente dessas técnicas pode levar a melhores resultados do que simplesmente aplicar a geração de testes pseudoaleatórios com restrições convencionais.

7. Conclusões

Este trabalho contribuiu para a avaliação de quatro geradores de testes para expor erros de projeto no subsistema de memória compartilhada. A principal contribuição foi a proposta de uma métrica de cobertura funcional, sua implementação e avaliação experimental para diferentes geradores sob diferentes parâmetros de geração. Uma contribuição secundária, mas crucial para a avaliação da eficácia e do esforço de verificação foi a elaboração de erros artificiais de projeto para desafiar os geradores. Essa experiência deixou clara a necessidade de que propriedades específicas do subsistema de memória (tais como princípios gerais de coerência e regras de consistência) sejam exploradas para melhorar a qualidade dos testes, norteando o desenvolvimento de novos geradores, ao invés do simples uso de meta-heurísticas de propósitos gerais.

A correlação entre os resultados de cobertura, esforço e eficácia tornam mais sólidas as conclusões em relação às propriedades dos programas de testes. Além disso, ao contrário de trabalhos [Wagner and Bertacco 2008, Qin and Mishra 2012] que realizam verificação orientada por protocolo, onde é necessária uma análise do protocolo de coerência utilizado para quebrá-lo em subespaços [Qin and Mishra 2012] ou criar uma modelagem de máquina de estados dicotômica [Wagner and Bertacco 2008], a adoção da cobertura de transições não prejudica o reuso dessa métrica para variantes do mesmo protocolo ou diferentes protocolos de coerência. Sendo assim, a utilização da métrica proposta para dirigir um novo gerador não limita a reusabilidade do *checker*, que é consequência de se usar um modelo de memória baseado em axiomas independentes do protocolo e da implementação específica.

Como tópico de investigação científica futura pretende-se propor um gerador de testes dirigidos que use as técnicas avaliadas dentro de um laço de realimentação através do qual, dependendo da cobertura medida, parâmetros do programa (n, s) e restrições de endereços sejam gerados automaticamente. Os resultados de cobertura obtidos deixam claro que CHAIN+ deve ser escolhido para um módulo *gerador* e que um módulo *diretor* deve ser desenvolvido levando em conta o aprendizado obtido neste trabalho.

Referências

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.

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

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.

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.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition.

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.

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

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.

Rambo, E., Henschel, O., and dos Santos, L. (2011). Automatic generation of memory consistency tests for chip multiprocessing. In *IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS)*, pages 542–545.

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.

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

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