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

Leandro da Silva Freitas

ACELERADORES E MULTIPROCESSADORES EM *CHIP*: O IMPACTO DA EXECUÇÃO FORA DE ORDEM NA VERIFICAÇÃO DE FUNCIONALIDADE E DE CONSISTÊNCIA

Florianópolis

2012

Leandro da Silva Freitas

ACELERADORES E MULTIPROCESSADORES EM *CHIP*: O IMPACTO DA EXECUÇÃO FORA DE ORDEM NA VERIFICAÇÃO DE FUNCIONALIDADE E DE CONSISTÊNCIA

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Luiz Cláudio Villar dos Santos, Prof. Dr.

Florianópolis

2012

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

Freitas, Leandro da Silva
Aceleradores e multiprocessadores em chip [dissertação] : o impacto da execução fora de ordem na verificação de funcionalidade e de consistência / Leandro da Silva Freitas ; orientador, Luiz Cláudio Villar dos Santos - Florianópolis, SC, 2012.
83 p. ; 21cm

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

Inclui referências

1. Ciência da Computação. 2. Verificação funcional. 3. Modelo de consistência de memória. 4. Scoreboard. 5. Multiprocessadores em chip. I. Santos, Luiz Cláudio Villar dos. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Leandro da Silva Freitas

ACELERADORES E MULTIPROCESSADORES EM *CHIP*: O IMPACTO DA EXECUÇÃO FORA DE ORDEM NA VERIFICAÇÃO DE FUNCIONALIDADE E DE CONSISTÊNCIA

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pela Universidade Federal de Santa Catarina.

Florianópolis, 21 de setembro 2012.

Ronaldo dos Santos Mello, Prof. Dr.
Coordenador

Banca Examinadora:

Luiz Cláudio Villar dos Santos, Prof. Dr.
Orientador

Sandro Rigo, Prof. Dr.

Djones Vinicius Lettnin, Prof. Dr.

Olinto José Varela Furtado, Prof. Dr.

À minha família.

AGRADECIMENTOS

Aos meus pais, Edson e Ana Rita, pela educação, formação e, mesmo estando longe geograficamente, por sempre me acompanharem de perto. Também à minha irmã, Isabela, pela companhia e pelo apoio.

À Lilian, minha namorada, pelo carinho, incentivo e, principalmente, pela compreensão nos momentos difíceis.

Ao meu orientador, Professor Dr. Luiz Cláudio Villar dos Santos, pelas contribuições técnicas, pelos conselhos profissionais e pela amizade.

Aos membros da banca, Professor Dr. Sandro Rigo, Professor Dr. Djones Vinicius Lettnin e Professor Dr. Olinto José Varela Furtado, por aceitarem o convite para avaliar este trabalho e pelas contribuições.

Aos colegas de LAPS/ECL e NIME pelas críticas, sugestões e pelos momentos de descontração. Em particular aos amigos Eberle Rambo, Gabriel Andrade e Gabriel Marcílio pela contribuição com a infraestrutura utilizada neste trabalho e pela colaboração mais direta.

Ao CNPq, no âmbito do Programa Nacional de Microeletrônica (PNM: 556757/2009-2), do Programa de Pesquisa, Desenvolvimento e Inovação em Áreas Estratégicas (PDI: 380778/2012-2) e do Instituto Nacional de Ciência e Tecnologia de Sistemas Micro e Nanoeletrônicos (NAMITEC: 573738/2008-4) pelo fomento e custeio parciais deste trabalho.

The cure for boredom is curiosity.
There is no cure for curiosity.

Dorothy Parker

RESUMO

Este trabalho aborda duas classes de problemas enfrentados na verificação de projetos que exibem comportamentos fora de ordem, especificamente a verificação funcional de aceleradores em *hardware* e a verificação de consistência em sistemas de memória compartilhada. Comportamentos fora de ordem surgem quando relaxam-se restrições de precedência para aumentar a taxa de uso de componentes de *hardware* concorrentes e, portanto, aumentar o desempenho. Entretanto, o projeto de um sistema que apresenta comportamentos fora de ordem é suscetível a erros pelo fato de o relaxamento de ordem requerer controle sofisticado. Este trabalho compara as garantias de verificação de três classes de *checkers* dinâmicos para módulos com suporte a eventos fora de ordem. Comprovadamente, *scoreboards* relaxados podem ser construídos com plenas garantias de verificação contanto que utilizem regras de atualização baseadas na remoção de dominadores. Resultados experimentais mostram que um *scoreboard* relaxado assim projetado requer aproximadamente 1/2 do esforço exigido por um *scoreboard* convencional. Verificar a conformidade do *hardware* com um modelo de consistência é um problema relevante cuja complexidade depende da observabilidade dos eventos de memória. Este trabalho também descreve uma nova técnica de verificação de consistência de memória *on-the-fly* a partir de uma representação executável de um sistema *multi-core*. Para aumentar a eficiência sem afetar as garantias de verificação, são monitorados três pontos por núcleo, ao invés de um ou dois, como proposto em trabalhos correlatos anteriores. Os três pontos foram selecionados para serem altamente independentes da microarquitetura do *core*. A técnica usa *scoreboards* relaxados concorrentes para detectar violações em cada *core*. Para detectar violações globais, utiliza-se a ordem linear de eventos induzida por um caso de teste. Comprovadamente, a técnica não induz falsos positivos nem falsos negativos quando o caso de teste expõe um erro que afeta as sequências monitoradas, tornando-se o primeiro *checker on-the-fly* com plenas garantias de verificação. Resultados experimentais mostram que ele requer aproximadamente 1/4 a 3/4 do esforço global exigido por um *checker post-mortem* que monitora duas sequências por processador. A técnica é pelo menos 100 vezes mais rápida do que um *checker* que monitora uma única sequência por processador.

Palavras-chave: Verificação funcional. Modelo de consistência de memória. *Scoreboard*. Multiprocessadores em *chip*.

ABSTRACT

This work addresses two classes of problems faced when verifying designs exhibiting out-of-order behaviors, namely the functional verification of hardware accelerators and the verification of consistency in shared-memory systems. Out-of-order behaviors result from relaxing precedence constraints to increase the usage rate of concurrent hardware components and, therefore, lead to a performance improvement. However, the design of a system handling out-of-order behaviors is error prone, since order relaxation asks for sophisticated control. This work compares the verification guarantees of three classes of dynamic checkers for modules handling out-of-order behaviors. Provenly, relaxed scoreboards can be built with full verification guarantees, as far as they employ an update rule based on the removal of dominators. Experimental results show that such a relaxed scoreboard needs approximately 1/2 of the effort required by a conventional one. Verifying the hardware compliance with a consistency model is a relevant problem, whose complexity depends on the observability of memory events. This work also describes a novel on-the-fly technique for verifying memory consistency from an executable representation of a multi-core system. To increase efficiency without hampering verification guarantees, three points are monitored per core, instead of one or two, as proposed in previous related works. The points were selected to be largely independent from the core's microarchitecture. The technique relies on concurrent relaxed scoreboards to check for consistency violations in each core. To check for global violations, it employs a linear order of events induced by a given test case. Provenly, the technique neither indicates false negatives nor false positives when the test case exposes an error that affects the sampled sequences, making it the first on-the-fly checker with full guarantees. Experimental results show that it needs approximately 1/4 to 3/4 of the overall verification effort required by a post-mortem checker sampling two sequences per processor. The technique is at least 100 times faster than a checker sampling a single sequence per processor.

Keywords: Functional verification. Memory consistency model. Scoreboard. Chip multiprocessors.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE SÍMBOLOS

SUMÁRIO

1 INTRODUÇÃO

A história da computação é marcada pela democratização dos dispositivos: começou com uns poucos e caros *mainframes*, passando por minicomputadores, estações de trabalho, computadores pessoais, até os recentes *smartphones* e *tablets* [Reed, Gannon e Larus 2012]. Durante a evolução, ao mesmo tempo em que o tamanho dos dispositivos diminuiu, melhorou seu desempenho e aumentaram suas funcionalidades. Por muitos anos, essa evolução foi sustentada basicamente pela redução do tamanho dos transistores; entretanto, devido à barreira de potência, apenas o redimensionamento dos transistores deixou de ser suficiente [Patterson e Hennessy 2005, Hennessy, Patterson e Arpaci-Dusseau 2007, Reed, Gannon e Larus 2012].

Quando comparado a sistemas cujas funcionalidades estão repartidas em múltiplos *chips*, um sistema em um único *chip* (SoC, *system-on-a-chip*) resulta em menores dimensões, maior desempenho e menor consumo. Por isso, os dispositivos móveis pessoais (PMD, *personal mobile devices*) são baseados em SoCs. Atualmente, os PMDs são utilizados predominantemente para acesso à Internet e Computação em Nuvem, o que tende a aumentar a demanda por crescentes taxas de transferência de dados. Para suportar essa demanda, várias alternativas arquiteturais têm sido utilizadas, como: *pipelines*, emissão múltipla, execução fora de ordem, aceleradores em *hardware* e multiprocessamento. Devido à barreira de potência, o multiprocessamento em *chip* (MPSoC, *multiprocessor system-on-a-chip*) tornou-se uma solução mais energeticamente eficiente para sustentar altas taxas de dados do que, por exemplo, a emissão múltipla.

Por outro lado, a flexibilidade de se implementar algoritmos em *software* requer a busca e decodificação de instruções, etapas que contribuem para diminuir o desempenho e aumentar o consumo. Por isso, a implementação de um algoritmo em *hardware* é uma alternativa de projeto comum.

Por representarem soluções arquiteturais com alta eficiência energética, são frequentemente utilizados MPSoCs e aceleradores em *hardware*. Por exemplo, a Figura 1 mostra o diagrama de blocos do SoC OMAP4470 [Texas Instruments], que conta com dois núcleos multiprocessados Cortex-A9 [ARM Holdings] além de aceleradores para multimídia e criptografia.

1.1 RELEVÂNCIA DO RELAXAMENTO DA ORDEM EM MPSOCS

Para acompanhar o aumento da demanda por desempenho, as microarquiteturas frequentemente exploram comportamentos fora de ordem em di-

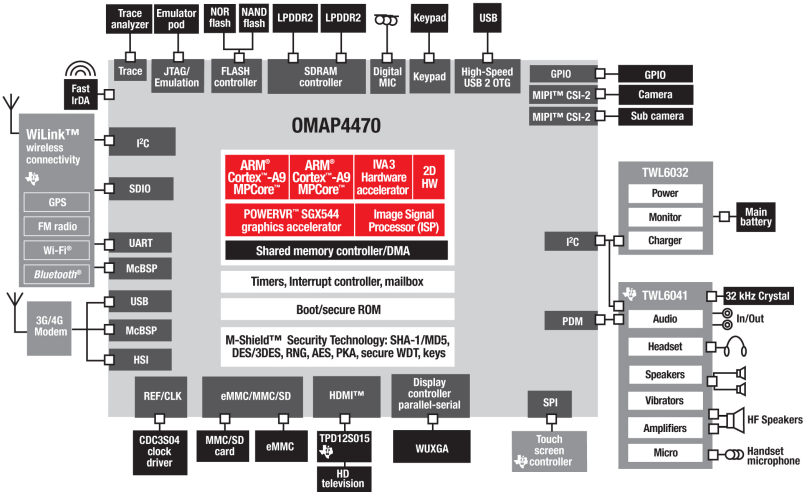


Figura 1: Diagrama de blocos de um SoC (Fonte: [Texas Instruments])

ferentes subsistemas, tais como hierarquia de memória, *pipeline* e interconexões. Tais comportamentos podem surgir, por exemplo, do *relaxamento* de consistência sequencial em memória compartilhada [Adve e Gharachorloo 1996], ordem de programa em *pipelines* [Jacobi 2002] e ordem de transações em barramentos [ARM Holdings].

Muitas aplicações de PMDs adotam o uso de múltiplas *threads* como modelo de computação. A comunicação entre duas *threads*, quando feita via memória compartilhada, pode levar a uma condição de corrida (*data race*) — quando ambas acessam a mesma posição de memória e pelo menos um dos acessos é uma escrita [Hennessy, Patterson e Arpaci-Dusseau 2007] [Hennessy, Patterson e Arpaci-Dusseau 2007]. Disso resulta uma ambiguidade no valor retornado por uma leitura. Essa ambiguidade é resolvida pelo modelo de consistência de memória (MCM, *memory consistency model*) adotado pelo multiprocessador.

Um MCM especifica, através de axiomas, quais valores podem ser retornados por uma operação de leitura. Intuitivamente, em um sistema com um único processador, o valor a ser retornado é o "último" valor escrito na mesma posição de memória. No entanto, o mesmo raciocínio não pode ser empregado em sistemas multiprocessados, pois duas execuções de um mesmo programa podem resultar em ordens diferentes de operações. Quanto mais restrito for o MCM, mais simples é a sua implementação em *hardware* e mais intuitivo é o comportamento do sistema; porém, menores são as possibilida-

des de otimizações para o *hardware* e para o compilador [Adve e Gharachorloo 1996]. Por outro lado, um MCM relaxado aumenta o desempenho do sistema, ficando a cargo do programador lançar mão de mecanismos de sincronização (como monitores e semáforos) para garantir que o seu programa execute da maneira desejada. No entanto, esse aumento de desempenho vem às custas de um *hardware* mais complexo e mais suscetível a erros, o que aumenta a importância da verificação.

Apesar da diminuição do número de aceleradores em *hardware* nos sistemas modernos, eles ainda são bastante utilizados e implementam, principalmente, algoritmos nas áreas de criptografia e multimídia. Os meios de conexão do sistema devem ser eficientes e flexíveis para que não representem gargalos que limitem o *throughput* potencial de um acelerador em *hardware*. As operações realizadas pelos aceleradores em *hardware* — como a codificação de um *frame* de um vídeo ou uma operação de acesso à memória — possuem latências distintas. Para que operações complexas e demoradas não bloqueiem as operações mais simples e rápidas, alguns protocolos de comunicação de alto desempenho permitem que transações sejam atendidas em uma ordem diferente daquela em que foram emitidas — desde que obedecendo as restrições estabelecidas pelo mestre do meio de conexão (por exemplo, através de rótulos ou IDs [OCP-IP Association 2009, ARM Holdings]).

Tanto em aceleradores em *hardware* quanto em modelos de consistência de memória, quanto maior a liberdade em se inverter a ordem, ou seja, quanto mais relaxado for o sistema, mais operações serão concluídas em menor intervalo de tempo, aumentando o *throughput* do sistema.

1.2 ESCOPO

Este trabalho aborda duas classes de problemas de verificação onde ocorre inversão na ordem de execução de eventos.

- **Aceleradores em *hardware*:** dada uma especificação executável prévalidada de um acelerador em *hardware* com suporte à execução fora de ordem e uma representação executável (derivada da primeira) que se queira verificar, determinar se, quando submetidos à mesma sequência de estímulos, ambas retornam sequências de resposta equivalentes. Isto é, todo elemento observado em uma sequência é também observado na outra e a ordem desses elementos preserva uma ordem parcial pré-especificada.

- **Consistência de memória:** dado um sistema multiprocessado cuja semântica de memória é definida por um determinado modelo de consis-

tência, para uma dada execução de um programa, determinar se existe uma ordem global das operações de acesso à memória realizadas (por todos os processadores) que satisfaça as restrições do modelo de consistência.

Para ambas as classes de problemas, adotou-se a mesma abordagem: a verificação dinâmica de uma representação executável do (sub) sistema. O uso de verificação dinâmica justifica-se pelo fato de os métodos formais tornarem-se impraticáveis com o crescimento do número de estados do projeto. Em geral, eles são utilizados apenas para verificar algumas partes críticas dos projetos. A disponibilidade de representações executáveis (exigência do método *response checking*) justifica-se diante da popularização da metodologia de projeto em nível de sistema (*electronic system level*, ESL) [Bailey, Martin e Piziali 2007, Rigo, Azevedo e Santos 2011].

1.3 CONTRIBUIÇÕES

1.3.1 Modelo de *scoreboard* relaxado para verificação de aceleradores em *hardware* com plenas garantias

Ao contrário de um *scoreboard* convencional [Wile, Goss e Roesner 2005], o método aqui proposto para a verificação de aceleradores em *hardware* não requer a sincronização dos domínios temporais do dispositivo sob verificação e de sua especificação executável. O método proposto, assim como outros métodos correlatos [Marcilio et al. 2009, Shacham et al. 2008], admite valores de resposta em diferentes ordens (desde que obedecendo à ordem parcial pré-especificada). Por isso, diversas implementações podem ser consideradas corretas para uma mesma especificação, ao contrário do que ocorre quando se utiliza um *scoreboard* convencional. O método proposto utiliza uma abordagem de verificação *black-box* e, ao contrário de [Marcilio et al. 2009], a verificação é realizada *on-the-fly* (durante a simulação). Obteve-se um *checker* completo[1] — ou seja, que não induz falsos positivos nem falsos negativos — com desempenho semelhante a um *scoreboard* convencional.

[1] Entre os autores que adotam esta terminologia pode-se citar [Manovit e Hangal 2006].

1.3.2 *Nova* técnica de verificação de consistência de memória

A maioria das técnicas de verificação de consistência de memória reportadas na literatura utilizam um único ponto de observação *por processador*, pois foram projetadas para serem usadas para o teste de protótipos e reusadas em tempo de verificação. Um trabalho recente [Rambo, Henschel e Santos 2012] reporta uma técnica desenvolvida para operar exclusivamente sobre representações executáveis, através da monitoração de *dois pontos* por processador. Em contraste com os trabalhos correlatos, este trabalho demonstra que, monitorando-se *três pontos*, é possível garantir a detecção de todos os erros (desde que estimulados durante a simulação) sem a necessidade de *backtracking*[2]. A técnica proposta é a primeira a fornecer plenas garantias de verificação[3] com uso de uma abordagem *on-the-fly*. Além disso, a utilização de múltiplos *scoreboards* para verificação de consistência (um para cada processador) é uma abordagem original.

1.4 ORGANIZAÇÃO DESTA DISSERTAÇÃO

Assim está organizada esta dissertação: o Capítulo 2 faz uma revisão sobre verificação de *hardware*, dando ênfase aos métodos dinâmicos. O Capítulo 3 propõe um modelo para a construção de um *checker on-the-fly* capaz de tratar eventos fora de ordem com plenas garantias de verificação. O Capítulo 4 propõe um algoritmo, baseado no modelo proposto no capítulo anterior, para a verificação de consistência de memória. Por fim, o Capítulo 5 apresenta as conclusões e perspectivas de trabalhos futuros.

1.5 REPERCUSSÃO E IMPACTO ESPERADO

Os resultados preliminares obtidos com a técnica do Capítulo 3 já tiveram seu mérito reconhecido pelo comitê de programa da *IEEE International Conference on Computer Design* (ICCD 2012), resultando na publicação de trabalho intitulado "*Efficient Verification of Out-of-Order Behaviors with Relaxed Scoreboards*" nos anais daquele evento [Freitas, Andrade e Santos 2012].

[2] *Backtracking* é o ato de reconsiderar, durante a verificação, decisões anteriormente tomadas que se mostraram equivocadas posteriormente.

[3] Considera-se que um *checker* dinâmico fornece plenas garantias de verificação quando ele determina corretamente a equivalência entre um dispositivo sob verificação e uma especificação executável pré-validada *para um determinado conjunto de estímulos*.

Os resultados experimentais completos e os resultados teóricos da técnica proposta no Capítulo 3 tiveram seu mérito reconhecido pelo Comitê de Programa da *IEEE International Conference on Electronics, Circuits, and Systems* (ICECS 2012), resultando na publicação de trabalho intitulado "*A Template for the Construction of Efficient Checkers with Full Verification Guarantees*" nos anais daquele evento [Freitas, Andrade e Santos 2012].

Os resultados teóricos e experimentais da técnica proposta no Capítulo 4, que não haviam sido apreciados pela comunidade científica até a data da publicação desta dissertação, foram submetidos ao evento *Design, Automation & Test in Europe* (DATE 2013). Dada a importância da verificação em sistemas *multi-core*, espera-se causar significativo impacto com essa técnica.

2 ABORDAGENS PARA VERIFICAÇÃO DE *HARDWARE*

A solução para o problema de verificação é, geralmente, uma combinação de métodos estáticos e dinâmicos. Um método de verificação *estático* baseia-se em análise formal para demonstrar que *uma* determinada funcionalidade foi corretamente implementada, qualquer que seja o estado atingido pelo dispositivo sob verificação (DUV, *device under verification*) (100% de cobertura). Um método de verificação *dinâmico* requer a simulação de uma representação do DUV, a qual é submetida a um conjunto de estímulos para os quais se conhece a resposta esperada, verificando *múltiplas* funcionalidades, mas com um índice de cobertura limitado pelo tempo disponível para a verificação [Bailey, Martin e Piziali 2007].

Este capítulo faz uma revisão dos métodos de verificação mais utilizados, dando maior ênfase aos métodos dinâmicos, e introduz alguns conceitos que serão utilizados ao longo desta dissertação.

2.1 MÉTODOS ESTÁTICOS

Existem duas classes fundamentais de métodos formais: verificação de equivalência e verificação de propriedades. De um ponto de vista externo, ambas comparam duas representações de um sistema. Enquanto a primeira, amparando-se em diagramas de decisão binária[1] (BDDs, *Binary Decision Diagram*) e técnicas de análise de "satisfatibilidade"[2] (*SAT*, *satisfiability*), verifica a equivalência entre o DUV e uma representação em um nível de abstração mais alto, a segunda compara o DUV a um conjunto de comportamentos esperados, na forma de propriedades [Bailey, Martin e Piziali 2007].

Outras técnicas bem conhecidas são *Model Checking* e *Theorem Proving*. A primeira consiste em construir um modelo abstrato do sistema (através de uma máquina de estados finitos, por exemplo) e estabelecer se as propriedades especificadas são válidas naquele modelo [Clarke, Emerson e Sistla 1986]. A segunda consiste em descrever tanto a implementação quanto a especificação através de lógica formal e provar matematicamente que a implementação esteja de acordo com a especificação [Seger 1992]. Enquanto *Theorem Proving* é uma técnica que consegue tratar problemas maiores, po-

[1] Os métodos de verificação que utilizam BDDs consistem, basicamente, em representar uma função booleana utilizando uma árvore binária (ordenada e reduzida). Como existe uma única representação para cada função booleana [Bryant 1986], a obtenção de representações idênticas caracteriza a equivalência entre funções distintas.

[2] Consiste em determinar se uma função booleana pode ser satisfeita; isto é, se as variáveis que compõem a função podem ser atribuídas de forma que o resultado da função seja igual a 1.

rem demandando grande esforço manual, *Model Checking* consegue tratar problemas complexos com menor esforço, mas rapidamente se torna impraticável (em termos de consumo de memória) conforme cresce o tamanho do problema. Tomando essas duas técnicas como exemplo, existem iniciativas que combinam duas ou mais técnicas distintas para abordar problemas que dificilmente seriam resolvidos utilizando apenas uma delas [Jacobi 2002].

Os métodos formais dificilmente serão aplicados a sistemas completos, pois, como eles exercitam todos os estados do DUV, rapidamente a capacidade da memória da estação de trabalho empregada para verificação se tornaria insuficiente. Além disso, o fato de a verificação formal obter cobertura total não é garantia de um DUV sem erros ao final do processo. Embora ela exercite todos os estados do DUV, sua eficácia depende de quão adequado é o conjunto de propriedades para se estabelecer se o DUV contém ou não erros [Bailey, Martin e Piziali 2007].

2.2 MÉTODOS DINÂMICOS

A verificação dinâmica consiste na aplicação de estímulos em um DUV simulado para expor erros [Bailey, Martin e Piziali 2007]. A não ser em projetos menores, dificilmente os métodos dinâmicos conseguem atingir máxima cobertura. Em outras palavras, quando comparados aos estáticos, para um mesmo tempo disponível para a verificação, *checkers* dinâmicos resultam em menor cobertura, mas são capazes de verificar múltiplas funcionalidades. De acordo com [Wile, Goss e Roesner 2005], a verificação baseada em simulação pode ser classificada em: *on-the-fly*, quando a verificação ocorre durante a simulação; *end-of-test case*, quando a verificação ocorre após a simulação, mas no mesmo ambiente que a simulação; ou *post-mortem*, quando a verificação ocorre após a simulação, em um ambiente específico para a verificação.

A verificação baseada em simulação compreende três aspectos: geração de estímulos, *response checking* e análise de cobertura [Bailey, Martin e Piziali 2007]. O índice de cobertura é uma consequência da qualidade do processo de geração de estímulos. Uma vez que um erro tenha sido estimulado durante a simulação, cabe ao *checker* sinalizá-lo. As próximas seções exploram um pouco mais esses aspectos.

2.2.1 Análise de cobertura e geração de estímulos

A cobertura é qualquer medida que indique o quão exaustivamente o DUV foi estimulado, sendo que quanto maior a cobertura, menor a possibilidade de um erro ter passado despercebido pela verificação (ignorando-se a possibilidade de o *checker* não detectar um erro que tenha sido estimulado durante a simulação) [Bailey, Martin e Piziali 2007].

Para aumentar o índice de cobertura deve-se melhorar a geração de estímulos, esta que pode ser feita de maneira determinística ou aleatória [Wile, Goss e Roesner 2005]. Os estímulos determinísticos são gerados antes da simulação e são utilizados para exercitar cenários específicos que o engenheiro de verificação deseja observar. A vantagem de se gerar estímulos aleatoriamente é que, com a mesma quantidade de linhas de código, pode-se exercitar um maior número de cenários. Na realidade, dificilmente se faz a geração de estímulos completamente aleatória, mas sim de uma forma restrita para evitar combinações de sinais não permitidas. Existem trabalhos que ajustam os parâmetros do gerador de estímulos dinamicamente para "conduzir" a simulação em direção aos casos extremos (que dificilmente seriam exercitados através da geração de estímulos puramente aleatória), aumentando o índice de cobertura.

2.2.2 *Response checking*

O elemento do ambiente de verificação responsável por sinalizar os erros do DUV é o *checker*, às vezes contando com o auxílio de um *scoreboard*. Originalmente, o termo "*scoreboard*" era utilizado para se referir a uma estrutura de dados que armazena estímulos de entrada ou resultados esperados. No entanto, em um trabalho recente [Shacham et al. 2008], utiliza-se o termo para se referir a uma estrutura que, além de armazenar valores, acumula também a função de *checker*. Este trabalho adota esse conceito de *scoreboard*.

Para poder dizer se um determinado resultado emitido pelo DUV está correto, o *checker* precisa de uma sequência de resultados esperados para servir como referência, a qual pode ser obtida de duas formas:

- **Vetor de referência:** uma sequência de valores gerados, seja manualmente, seja automaticamente, e armazenados em arquivo antes da simulação. Durante a simulação, o *scoreboard* carrega o vetor de referência e retorna os seus valores à medida em que vão sendo solicitados pelo *checker*.

- **Modelo de referência (RGM, *reference golden model*):** uma representação de projeto, escrita em linguagem de alto nível (como C/C++). O RGM e o DUV são submetidos à mesma sequência de estímulos de entrada e os seus resultados são comparados pelo *checker*. Assume-se que o RGM tenha sido previamente validado. Sendo assim, uma incompatibilidade detectada entre os resultados (observados e esperados) indica a presença de erro(s) no DUV.

De posse das sequências de valores esperados (referência) e observados (emitidos pelo DUV), o *checker* deve ser capaz de estabelecer a equivalência entre elas. Essa tarefa pode ser trivial quando ambas as sequências estão ordenadas. No entanto, nem sempre essa premissa é verdadeira.

2.2.3 *Checkers* temporais e de dados

Os *checkers* temporais e os *checkers* de dados podem ser inseridos em diversas partes do DUV para verificar determinadas propriedades do sistema. Este tipo de verificação complementa o *response checking*. Podendo ser implementados utilizando tanto código procedural quanto asserções, esses checkers podem ser inseridos bem próximos à fonte de uma anomalia, reduzindo assim o tempo de depuração [Bailey, Martin e Piziali 2007]. Esta classe de checkers está fora do escopo desta dissertação.

3 VERIFICAÇÃO EFICIENTE DE ACELERADORES EM HW

Como o relaxamento de ordem requer um controle sofisticado, o projeto de um módulo com capacidade de tratar comportamentos fora de ordem é suscetível a erro. A verificação funcional desse tipo de sistema é desafiadora, pois o módulo não preserva em sua saída a ordem correspondente aos estímulos de entrada [Bailey, Martin e Piziali 2007], violando uma suposição básica dos *scoreboards* convencionais. Portanto, um *checker* adequado deve *garantir* a distinção entre ordenamentos válidos e inválidos.

Estritamente falando, um *checker* fornece **plenas garantias de verificação** quando ele prova a não-equivalência entre duas representações de um projeto para ao menos um estímulo ou prova suas equivalências para todos os possíveis estímulos, o que é impraticável para verificação dinâmica. No entanto, um *checker* dinâmico pode ser projetado para prover plenas garantias de verificação *sob a perspectiva de um caso de teste*[1] quando ele emprega uma **condição necessária e suficiente** para estabelecer que um DUV é funcionalmente equivalente a uma representação do projeto pré-validada para um *dado* estímulo de entrada. Tal *checker* sempre encontra um erro exposto por um caso de teste[2] e nunca sinaliza erros quando verifica um DUV correto independentemente do caso de teste[3]. Essa noção tem a vantagem de desacoplar cobertura e *response checking*.

Checkers on-the-fly [Bailey, Martin e Piziali 2007,Shacham et al. 2008] utilizam *scoreboards* para emparelhar sucessivamente o valor observado em um DUV com um valor esperado, sinalizando um erro assim que uma incompatibilidade for encontrada. Eles contrastam com *checkers post-mortem* [Manovit e Hangal 2006, Marcilio et al. 2009, Hu et al. 2012], que se baseiam no conjunto de todos os comportamentos induzidos por um caso de teste para tomar decisões globais. Um *scoreboard* é um *checker* de *valor* que oferece garantias plenas para DUVs que preservam a ordem dos dados. Quando este não é o caso, os domínios temporais do *scoreboard* e do DUV devem ser sincronizados [Bailey, Martin e Piziali 2007] para meramente contornar a inversão de ordem de forma a permitir a verificação dos valores esperados. No entanto, esse expediente negligencia a verificação da ordem. Por esse motivo, um *scoreboard* convencional não pode verificar se os comportamentos estão em uma *ordem válida* a menos que ele faça tomadas de decisões baseadas em heurísticas, o que pode afetar a plenitude de suas garantias. Este capítulo in-

[1] Frequentemente essa frase será omitida quando isso estiver claro pelo contexto.
[2] Ou seja, nunca emite falso negativo.
[3] Ou seja, nunca emite falso positivo.

vestiga até que ponto *scoreboards* relaxados podem oferecer plenas garantias de verificação enquanto preservam a eficiência.

O fato de as garantias de verificação de *checkers on-the-fly* serem negligenciadas pela literatura parece indicar que muitos *checkers* assumem garantias plenas ou simplesmente ignoram o problema. Para preencher essa lacuna, esta dissertação compara garantias teóricas e evidências experimentais de três classes de *checkers* dinâmicos. Não é de conhecimento do autor de que uma comparação similar tenha sido reportada até o momento. Além disso, amparando-se nas noções de *regra de atualização* [Shacham et al. 2008] e *emparelhamento próprio* [Marcilio et al. 2009] e utilizando garantias de verificação já estabelecidas para *checkers post-mortem* [Marcilio et al. 2009], esta dissertação oferece provas matemáticas de que *scoreboards* relaxados podem ser construídos com plenas garantias de verificação, contanto que suas regras de atualização induzam um emparelhamento próprio. Resultados experimentais mostram que o esforço de verificação pode ser reduzido em 50% quando se substitui um *scoreboard* convencional por um relaxado.

Este capítulo está organizado como segue. A Seção 3.1 informalmente introduz noções importantes por meio de um exemplo. A Seção 3.2 formula o problema alvo. A Seção 3.3 discute as limitações de *checkers* convencionais em face dos comportamentos fora de ordem. A Seção 3.4 compara formalmente as garantias de verificação de três classes de *checkers*. A Seção 3.5 apresenta os resultados experimentais que levam às conclusões na Seção 3.6.

3.1 EXEMPLO ILUSTRATIVO

Seja S um módulo acoplado a um meio de interconexão cujo protocolo possua regras especificando que transações fora de ordem com o mesmo identificador devem ser concluídas na mesma ordem em que foram emitidas [ARM Holdings]. Assuma que S é um acelerador em *hardware* que implementa as operações $+$, $-$ e $\times$ sobre números complexos. A Tabela 1 mostra as operações iniciadas por transações sucessivas e a sequência de respostas esperada (obtida de um RGM[4]). Devido às diferentes latências das transações, a sequência de respostas observada pode exibir valores fora de ordem.

As Figuras 2, 3 e 4 mostram três *checkers* em dois cenários distintos para o caso de teste da Tabela 1. Cada *checker* deve retornar verdadeiro, para um DUV correto, e falso, em caso contrário. O exemplo assume que o erro no DUV incorreto não afeta os valores esperados, mas induz ordenamentos inváv-

[4]Embora assuma-se um RGM, a técnica proposta também pode ser aplicada com base em um vetor de referência (veja Seção 2.2.2).

Tabela 1: Um caso de teste que induz comportamentos fora de ordem

Estímulos		**RGM**
ID	Operação	Resultados esperados
0	$(1+1i)\times(2+2i)$	$(0+4i)$
0	$(2+2i)+(2+2i)$	$(4+4i)$
1	$(1+2i)\times(3+4i)$	$(-5+10i)$
2	$(0+2i)+(0+2i)$	$(0+4i)$

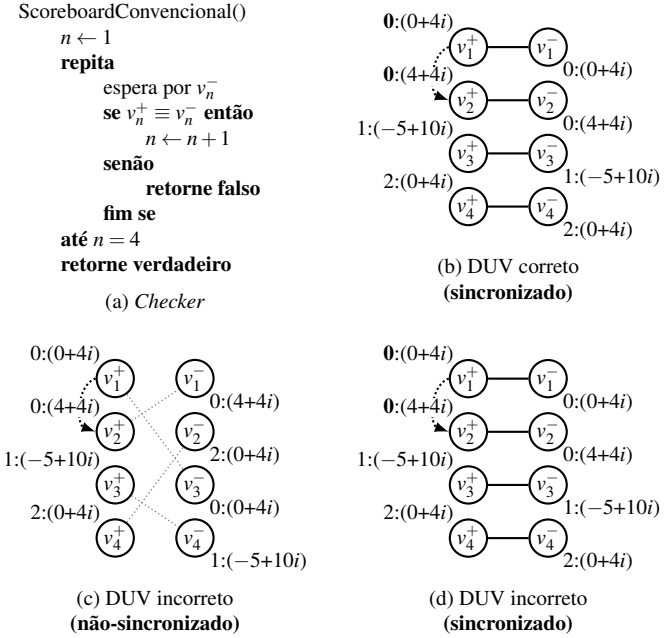
lidos. Os dois primeiros *checkers* baseiam-se na noção de *scoreboards* [Bailey, Martin e Piziali 2007], que mapeiam estímulos para resultados esperados. Para uma dada sequência de estímulos, um *scoreboard convencional* (Figura 2a) aceita apenas uma sequência de respostas, enquanto um *scoreboard relaxado* (Figura 3a) aceita sequências de respostas distintas[5]. O terceiro é um *checker post-mortem* (Figura 4a), que se baseia diretamente no RGM e não em um *scoreboard*.

Suponha que S possua duas representações: S^+ é um RGM atemporal e S^- é um DUV com temporização aproximada [Bailey, Martin e Piziali 2007, Black e Donovan 2004]. Quer-se verificar se S^+ e S^- são funcionalmente equivalentes para um dado caso de teste. Suponha que V^+ e V^- sejam conjuntos de valores de respostas *esperadas* e *observadas*, respectivamente. Assuma que o protocolo especifique uma ordem parcial $\leq$ para valores esperados; isto é, $v_i^+ \leq v_j^+$ denota que v_i^+ deve preceder v_j^+ (devido a transações com o mesmo ID). Nas Figuras 2, 3 e 4, os conjuntos V^+ e V^- são representadas como partições do grafo onde vértices adjacentes denotam valores equivalentes. Mantendo os vértices de cada partição na ordem em que os valores foram amostrados, o grafo bipartido também captura (embora implicitamente) as duas *sequências* de eventos $(v_1^+, v_2^+, v_3^+, v_4^+)$ e $(v_1^-, v_2^-, v_3^-, v_4^-)$. A ordem especificada $\leq$ é representada esquematicamente por flechas pontilhadas (que não são arestas do grafo bipartido).

Note que um cruzamento de arestas representa um comportamento fora de ordem. Se os valores envolvidos estão ordenados pela relação $\leq$, esse cruzamento indica um erro; em caso contrário, ele denota um comportamento válido. Diz-se que o primeiro é um cruzamento *impróprio* e o último *próprio*. Um emparelhamento incompleto significa que um comportamento especificado para S^+ não foi implementado em S^- e indica um DUV incorreto. No entanto, um emparelhamento completo somente indica um um DUV correto se ele não induz cruzamentos impróprios. Esse conceito, que foi de-

[5]Isto é, múltiplos comportamentos distintos com a mesma funcionalidade, mas diferentes ordens válidas.

nominado *emparelhamento próprio*, é emprestado de [Marcilio et al. 2009] para a análise seguinte.



(a) *Checker*

(b) DUV correto **(sincronizado)**

(c) DUV incorreto **(não-sincronizado)**

(d) DUV incorreto **(sincronizado)**

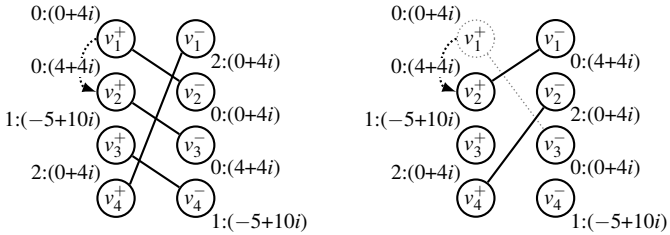
Figura 2: Como um *scoreboard* convencional lida com valor e ordem

Um *scoreboard* convencional (Figura 2a) compara valores esperados (v_n^+) e observados (v_n^-) na ordem induzida pela sequência de estímulos e retorna falso tão logo uma inconsistência seja detectada. Como é incapaz de tratar inversões de ordem, esse *checker* exige que a sincronização dos domínios temporais[6] [Bailey, Martin e Piziali 2007] do *scoreboard* e do DUV (representado pelo comando `wait`). Enquanto os valores esperados são observados no DUV sincronizado, o *scoreboard* convencional retorna verdadeiro, mesmo se a sincronização em tempo de verificação impedir a detecção de um erro (na representação de projeto) que será observado (no *hardware*) em tempo de execução (Figura 2c), caracterizando assim um falso negativo. Por isso o grafo da Figura 2b é idêntico ao da Figura 2d, significando que o ambiente de verificação induz um comportamento onde não ocorre inversão de ordem.

[6]Isto é, um novo estímulo só é enviado depois que o resultado do estímulo anterior esteja disponível à saída do DUV.

```
ScoreboardRelaxado()
    n ← 1
    repita
        se ∃v_j^+ ∈ V^+ | v_j^+ ≡ v_n^- então
            V^+ ← V^+ − {v_j^+} − {v_k^+ | v_k^+ ≤ v_j^+}
            n ← n + 1
        senão
            retorne falso
        fim se
    até n = 4
    retorne verdadeiro
```

(a) *Checker*



(b) DUV correto (c) DUV incorreto

Figura 3: Como um *scoreboard* relaxado lida com valor e ordem

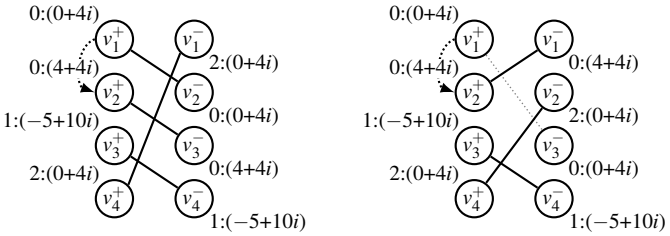
Um *scoreboard* relaxado (Figura 3a) consulta sua estrutura de dados em busca de um valor esperado (v_j^+) equivalente ao valor observado atual (v_n^-). Se existir mais de um valor equivalente, deve-se utilizar algum critério de desempate (o primeiro valor equivalente, por exemplo). O valor emparelhado é então removido do *scoreboard* junto com todo valor (v_k^+) que o domina[7] em $\leq$. Note que, na ausência de cruzamentos impróprios (Figura 3b), o conjunto de dominadores é sempre vazio. Como exatamente um valor é removido por vez, o *scoreboard* relaxado retorna verdadeiro para um DUV correto. Contudo, quando existe um cruzamento impróprio (Figura 3c), a remoção de um dominador (vértice cinza) impede o emparelhamento de um valor posterior. Como ele falha no emparelhamento daquele valor, o *checker* retorna falso para um DUV incorreto. Em resumo, o *scoreboard* relaxado assim construído fornece plenas garantias de verificação.

Um *checker post-mortem* exige a disponibilidade de todos os valores observados antes de iniciar a verificação. Isso permite a tomada de deci-

[7] Utiliza-se aqui uma noção de dominância similar à utilizada em grafos orientados acíclicos. Suponha que $v_i^+, v_2^+, ..., v_n^+$ sejam vértices de um grafo orientado acíclico cujas arestas representam a relação de precedência $\leq$. Diz-se que v_k^+ domina v_i^+ se todo caminho que atinge v_k^+ passa por v_i^+.

```
CheckerPostmortem()
    V ← V⁺ ∪ V⁻
    E ← {(v⁺, v⁻) ∈ V⁺ × V⁻ | v⁺ ≡ v⁻}
    M ← {e ∈ E | ∀e′ ≠ e : ¬χ(e, e′)}
    se |M| ← |V⁺| então
        retorne verdadeiro
    senão
        retorne falso
    fim se
```

(a) *Checker*



(b) DUV correto (c) DUV incorreto

Figura 4: Como um *checker post-mortem* lida com valor e ordem

sões globais. Por exemplo, o *checker* descrito na Figura 4a tenta encontrar um emparelhamento próprio ($\mathscr{M}$) utilizando a função $\chi(e,e')$, que retorna verdadeiro quando as arestas e e e' formam um cruzamento impróprio. Na Figura 4b, como os cruzamentos entre a aresta (v_4^+, v_1^-) e as arestas (v_1^+, v_2^-), (v_2^+, v_3^-), e (v_3^+, v_4^-) são todos cruzamentos próprios, um emparelhamento próprio é encontrado e o *checker* indica um DUV correto. Em contraste, a Figura 4c mostra que, após a aresta (v_2^+, v_1^-) ser incluída no emparelhamento $\mathscr{M}$, a aresta (v_1^+, v_3^-) não pode ser incluída em $\mathscr{M}$, pois isso levaria a um cruzamento impróprio (pois $v_1^+ \leq v_2^+$). Como nem todos os valores observados puderam ser emparelhados apropriadamente, o *checker* indica um erro no DUV incorreto.

Este exemplo mostra que o impacto de decisões locais nas garantias de verificação deve ser considerado ao projetar *checkers on-the-fly*. Se um *checker* utiliza uma condição suficiente para correção, ele pode induzir falsos positivos, o que leva à perda de tempo procurando erros inexistentes. Se ele emprega uma condição necessária, ele pode emitir falsos negativos, o que requer mais casos de teste para manter índices de cobertura aceitáveis. Por esse motivo, *checkers on-the-fly* devem empregar uma condição *necessária e suficiente* para resolver o problema de verificação, que é formalmente definido na próxima seção.

3.2 PROBLEMA ALVO

Dada uma ordem de eventos (pré-especificada) a ser preservada, duas representações de projeto distintas são consideradas equivalentes, para a mesma sequência de entrada, quando suas sequências de respostas são tais que todo valor observado em uma delas é também observado na outra, como formalizado a seguir.

Definição 1. *Dada uma ordem parcial $\leq$ sobre o conjunto $V^+ = \{v_1^+, v_2^+, \cdots, v_N^+\}$, diz-se que as sequências $(v_1^+, v_2^+, \cdots, v_N^+)$ e $(v_1^-, v_2^-, \cdots, v_N^-)$ são propriamente mapeadas se e somente se existe um mapeamento $\mu : \{1,2,\cdots,N\} \mapsto \{1,2,\cdots,N\}$ que satisfaça as seguintes condições [Marcilio et al. 2009]:*

- *μ é bijetora;*
- *$\forall v_j^+ \in V^+ : (\mu(j) = n) \Rightarrow (v_j^+ \equiv v_n^-)$;*
- *$\forall v_i^+, v_j^+ \in V^+ : (v_i^+ \leq v_j^+) \wedge (\mu(i) = t) \wedge (\mu(j) = m) \Rightarrow (t < m)$.*

Definição 2. *Dadas duas representações de projeto S^+ e S^- de um módulo S e a sequência de estímulos de entrada, S^+ e S^- são funcionalmente equivalentes com respeito àquela sequência se e somente se suas sequências de saída são propriamente mapeadas.*

Problema 1. *Dado um caso de teste induzindo uma sequência de N estímulos, uma ordem parcial $\leq$ e as sequências de saída $(v_1^+, v_2^+, \cdots, v_N^+)$ e $(v_1^-, v_2^-, \cdots, v_N^-)$ de duas representações de projeto S^+ e S^-, verificar se S^+ e S^- são funcionalmente equivalentes com respeito àquela sequência de entrada.*

3.3 TRABALHOS RELACIONADOS

Apesar da abrangente cobertura literária geral sobre verificação funcional (por exemplo [Bailey, Martin e Piziali 2007, Wile, Goss e Roesner 2005]), *checkers* capazes de tratar eventos fora de ordem normalmente são abordados na literatura de domínios especializados.

A verificação de *pipelines* dinâmicos é um domínio bem conhecido onde eventos fora de ordem precisam ser tratados. Por exemplo, Jacobi 2002 propõe a combinação de *model checking* e prova de teoremas. Skakkebaek, Jones e Dill 1998 propõem uma abordagem similar, que primeiro deriva uma abstração sequencial da implementação fora de ordem e então estabelece suas

equivalências através de provas de teoremas. Apesar dos seus resultados, dificilmente a verificação formal de *pipelines* fora de ordem [Jacobi 2002, Skakkebaek, Jones e Dill 1998] substituiria totalmente a simulação. Abordagens semiformais costumam ser utilizadas na prática (por balancear garantias de verificação e tempo de execução)).

Eventos fora de ordem também ocorrem em multiprocessadores com memória compartilhada, onde a ordem de programa de leituras e escritas pode ser relaxada de acordo com as regras de um modelo de consistência de memória [Adve e Gharachorloo 1996]. Tipicamente, a verificação de consistência de memória utiliza *checkers post-mortem* (por exemplo [Manovit e Hangal 2006, Hu et al. 2012]), embora um checker on-the-fly eficiente tenha sido proposto [Shacham et al. 2008].

Vamos focar agora em abordagens mais gerais para construir *checkers* dinâmicos visando o Problema 1. A abordagem mais simples é *response-checking* [Wile, Goss e Roesner 2005], que assume que os valores esperados de um *scoreboard* (ou RGM) podem ser comparados aos valores observados em um DUV *na mesma ordem*. Quando o DUV não preserva aquela ordem, o ambiente de verificação é alterado de forma a sincronizar o DUV com o *scoreboard* (ou RGM). Apesar de tal expediente preservar a verificação adequada de valores, nada pode ser concluído sobre a ordem.

Portanto, um *scoreboard convencional* resolve uma variante do Problema 1 que assume sequências de eventos mantendo a mesma ordem induzida pela sequência de entrada. Por esse motivo, ele é incapaz de detectar violações de uma ordem pré-especificada para os eventos.

Como eventos fora de ordem resultam de uma especificação relaxada, diferentes sequências de resposta são aceitáveis para uma mesma sequência de estímulos. Isso leva à noção de *scoreboard relaxado*. Várias técnicas de domínio específico baseiam-se nesse conceito. Por exemplo, essa noção foi explorada pela primeira vez por [Saha et al. 1995] para verificar consistência de memória compartilhada. Mais tarde, o conceito foi estendido por Shacham et al. 2008 e modelado como um *framework*, que é construído incrementalmente de acordo com as propriedades do modelo de consistência de memória. Inicialmente, ele admite uma grande quantidade de comportamentos mas, à medida em que novas propriedades vão sendo codificadas no *framework*, o conjunto de comportamentos aceitáveis diminui progressivamente. Como reportado pelos autores, a técnica pode sinalizar falsos negativos [Shacham et al. 2008] ao resolver um problema de verificação mais complexo que o Problema 1.

Checkers post-mortem podem lidar com comportamentos fora-de-ordem com plenas garantias contanto que se evite a utilização de heurísticas. Várias técnicas de domínio específico foram propostas, especialmente para a

verificação de consistência de memória (exemplo [Manovit e Hangal 2006,Hu et al. 2012]). Por outro lado, Marcilio et al. 2009 propõe uma técnica de propósitos gerais para verificação *white-box*, a qual é baseada em *emparelhamento de grafos bipartidos* (*E-matching*). Os vértices de partições distintas, V^+ e V^-, representam valores amostrados em pontos equivalentes do RGM e do DUV, enquanto as arestas denotam a compatibilidade de valores esperados e observados. Ao contrário do emparelhamento convencional, um *E-matching* deve satisfazer uma restrição de ordem entre vértices pré-especificada. Devido à abordagem *white-box*, artefatos de implementação [Bailey, Martin e Piziali 2007] são modelados, levando a partições de cardinalidades distintas ($|V^+| \leq |V^-|$). Um erro é indicado se e somente se um emparelhamento próprio não puder ser encontrado.

Em suma, *scoreboards* convencionais oferecem garantias limitadas mas são muito eficientes, enquanto *checkers post-mortem* podem oferecer garantias plenas embora sejam bastante ineficientes. Isso torna *scoreboards* relaxados os candidatos naturais para tratar comportamentos fora de ordem. Por isso investigou-se *como* eles podem ser projetados de modo a preservar tanto a eficiência (tempo de verificação) quanto a eficácia (garantias de verificação). A primeira é abordada na Seção 3.5 e a última na próxima seção.

3.4 PROPOSTA DE UM MODELO COM PLENAS GARANTIAS

Primeiramente, vamos estabelecer que as garantias de verificação do E-matching servem como um limite superior para as garantias de verificação do Problema 1.

Teorema 1. *O algoritmo* `proper-matching`*, descrito em [Marcilio et al. 2009], nunca indica falsos negativos nem falsos positivos ao resolver o Problema 1.*

O Teorema 1, cuja prova encontra-se no Apêndice A, permite que uma noção crucial, desenvolvida matematicamente em trabalho correlato, possa ser reusada para a análise pretendida de *checkers on-the-fly*. Para garantir que esta dissertação seja autossuficiente, reproduz-se abaixo parte do formalismo original [Marcilio et al. 2009], tentando preservar ao máximo sua notação.

Definição 3. *Dada uma relação de precedência $\leq$, definida no conjunto V^+, diz-se que existe um* `cruzamento impróprio` *entre as arestas (v_j^+, v_n^-) e (v_k^+, v_m^-), com $j \neq k$, se e somente se $((v_j^+ \leq v_k^+) \wedge (n > m)) \vee ((v_k^+ \leq v_j^+) \wedge (m > n))$.*

Definição 4. *A função de cruzamento* `impróprio` *é definida como:*

$$\chi((v_j^+, v_n^-), (v_k^+, v_m^-)) = \begin{cases} \textit{Verdadeiro, se vale a Definição 3} \\ \textit{Falso, em caso contrário} \end{cases}$$

Outro trabalho correlato [Shacham et al. 2008], estabelece que um *scoreboard* relaxado deve tomar duas decisões locais para cada valor observado v_n^-:

- *Regra de emparelhamento*: entre múltiplos valores equivalentes armazenados em um *scoreboard*, selecionar um deles como o correspondente a v_n^-;
- *Regra de atualização*: após o emparelhamento de v_n^-, selecionar o(s) valor(es) a ser(em) removido(s) do *scoreboard*.

No âmbito desta dissertação, desenvolveu-se um modelo para o projeto de *scoreboards* relaxados onde a regra de emparelhamento foi tornada a mais simples possível (para máxima eficiência) e onde a regra de atualização remove os dominadores de um valor emparelhado (para plenas garantias).

O Algoritmo 1[8] mostra as regras adotadas nas linhas 8 (primeiro emparelhamento) e 9 (remoção de dominadores). Elas levam a uma condição necessária e suficiente para correção, de acordo com o Teorema 2, cuja prova encontra-se no Apêndice A.

Algoritmo 1 modelo-scoreboard-relaxado()

```
 1: n ← 0
 2: N ← |V+|
 3: repita
 4:     n ← n + 1
 5:     Q ← {v_i+ ∈ V+ : v_i+ ≡ v_n-}
 6:     se Q = ∅ então
 7:         retorne falso
 8:     fim se
 9:     j ← min{1 ≤ i ≤ N : v_i+ ∈ Q}
10:     V+ ← V+ − {v_j+} − {v_k+ ∈ V+ : v_k+ ≤ v_j+}
11: até n = N
12: retorne verdadeiro
```

Teorema 2. *Sob a perspectiva de um dado caso de teste, o Algoritmo 1 resolve o Problema 1 com plenas garantias de verificação.*

[8] Por simplicidade, este algoritmo pressupõe que V^+ e V^- têm escopo global.

3.5 RESULTADOS EXPERIMENTAIS

Esta seção compara três *checkers*: um *scoreboard* convencional (CS) [Bailey, Martin e Piziali 2007], um *scoreboard* relaxado (RS) projetado de acordo com o modelo descrito pelo Algoritmo 1 e um *checker* post-mortem (PM) implementando o algoritmo `proper-matching` descrito em [Marcilio et al. 2009]. Foram selecionados dois módulos exibindo comportamentos fora de ordem e modelados em SystemC. Posteriormente, foram construídas duas representações executáveis de cada módulo: um RGM atemporal e um DUV com temporização aproximada. A Tabela 2 resume os módulos e os tipos de erros injetados nos DUVs.

O módulo CAR foi verificado sob 14 cenários distintos (e1 to e11, e1+e4, e1+e5 e e4+e5) e o MCO sob 6 cenários (e12 a e16, e12+e13). Para cada *checker*, foi medida a sua *eficácia* como a proporção dos casos de teste para os quais um erro foi encontrado[9] e o seu *esforço* como a soma dos tempos de simulação médios e dos tempos de verificação médios. Os tempos de simulação/verificação são expressos em ms e foram medidos em uma estação de trabalho HP xw8600 (processador *quad-core* 2.66 GHz Intel® Xeon® e memória principal de 4GB).

Primeiramente, foram gerados aleatoriamente 100 casos de teste, cada um induzindo 300 estímulos de entrada. Eles foram então aplicados aos módulos CAR e MCO para cada cenário de erro adotado (resultando em 1400 e 600 casos de uso, respectivamente). As Figuras 5 e 6 comparam a eficácia e o esforço de cada *checker*. Como esperado, RS e PM possuem a mesma eficácia, que é aproximadamente 2 vezes maior do que a de CS. CS e RS requerem esforços similares, que são de 3 a 4 vezes menor que a de PM. Claramente, RS é a melhor escolha para casos de teste de tamanho fixo. Todos os erros puderam ser encontrados com RS e PM, exceto para 11% dos casos de teste, que não os expuseram. Por outro lado, os erros e1–e4 e e9 nunca foram encontrados por CS para quaisquer dos casos de teste utilizados.

Pôde-se observar que, além dos erros que afetam a ordem, alguns erros que afetam valor não foram expostos pelo CS, pois a exigida sincronização de domínio temporal altera a concorrência das operações em tempo de verificação. Por exemplo, o erro e9 só pode ser detectado se duas operações pendentes (uma adição e uma multiplicação) estão em execução simultaneamente. Entretanto, o erro não é exposto quando a adição completa antes da multiplicação começar. Efeitos similares explicam a baixa eficácia de CS em módulos que executam múltiplas operações simultaneamente.

[9] Isto é, "eficácia" é igual ao número de erros encontrados dividido pelo número de casos de teste.

Tabela 2: Caracterização dos erros injetados na representação executável do DUV

Módulo	**Erro**			
	Nome	**Localização**	**Descrição**	**Impacto**
Aritmética Complexa (**CAR**) [Özgür]: realiza (+, -, *, ÷) em números complexos com latências (3ns, 3ns, 10ns, 15ns)	e1	Unidade de despacho	ID não verificado	ordem
	e2	Unidade de despacho	Dependência de operações não verificada para divisões	ordem
	e3	Registrador de interface	Código de operação lido como ID	ordem
	e4	Unidade de despacho	Tipo de operação não verificado	valor
	e5	Subtrator	Subtratores transformados em somadores	valor
	e6	Controlador da ULA	Registrador de resultado de divisão com bit preso em 0	valor
	e7	Controlador da ULA	Bit de *status* preso em 0	valor
	e8	Controlador da ULA	Operadores da soma roteados para multiplicador	valor
	e9	Controlador da ULA	Multiplicador escreve no registrador de soma	valor
	e10	Registrador de interface	Endereço de leitura/escrita com bit preso em 0	valor
	e11	Registrador de interface	Endereço de leitura com bit preso em 0	valor
Controlador de Memória (**MCO**): gerencia portas independentes de leitura e escrita com latências 10ns e 15ns	e12	Unidade de despacho	ID não verificado	ordem
	e13	Unidade de despacho	Endereço não verificado	ordem
	e14	Unidade de despacho	Transbordo no contador de dependências	ordem
	e15	Interface da memória	Bit de endereço preso em 0	valor
	e16	Interface da memória	Bit de dados preso em 0	valor

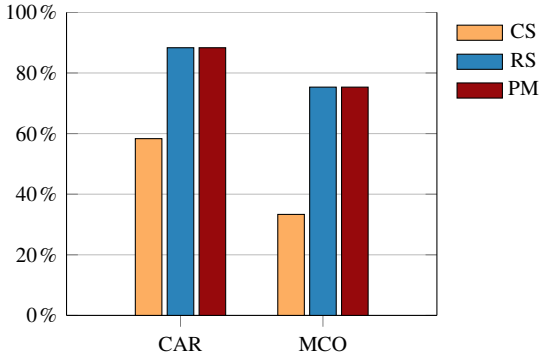


Figura 5: Eficácia para casos de teste aleatórios de tamanho fixo

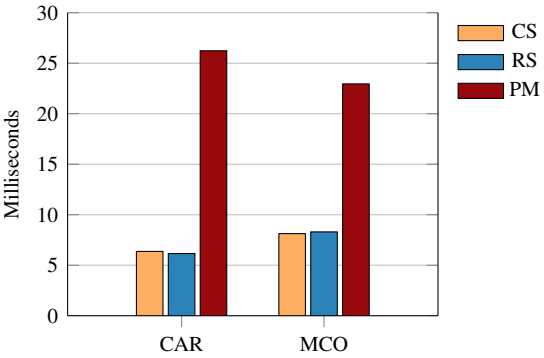


Figura 6: Esforço para casos de teste aleatórios de tamanho fixo

Em um segundo experimento, cada cenário foi submetido a 100 casos de teste de tamanhos crescentes. As Figuras 7 e 8 mostram os resultados. A Figura 7 mostra que a eficácia de CS satura em 58% e 33% quando os casos de teste de tamanhos 260 e 140 são aplicados aos módulos CAR e MCO, respectivamente. Observe que a eficácia de RS atinge níveis similares para casos de teste de tamanho 20, conforme indicado pelos pontos destacados naquela figura. Comparando os respectivos tempos de execução (indicados pelos pontos em destaque) na Figura 8, conclui-se que um RS apropriadamente projetado pode reduzir o esforço de verificação de 58% a 63% ($^{3,5}/_{5,58}$). Isso significa que um esforço de verificação aproximadamente duas vezes maior pode ser o preço a ser pago por negligenciar as garantias de verificação quando se projeta *checkers on-the-fly*.

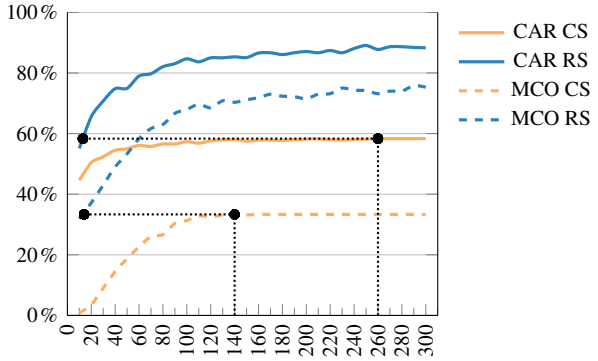


Figura 7: Eficácia para casos de teste aleatórios de tamanho crescente

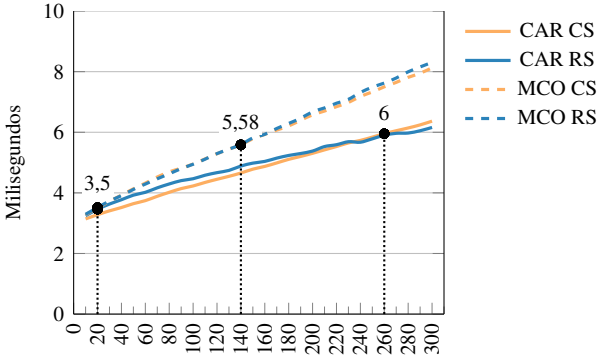


Figura 8: Esforço para casos de teste aleatórios de tamanho crescente

3.6 CONCLUSÕES

Como esta dissertação oferece provas matemáticas de que *scoreboards* bem projetados oferecem exatamente as mesmas garantias de verificação de *checkers post-mortem* menos eficientes e mostra experimentalmente que o esforço de verificação pode ser reduzido pela metade quando um *scoreboard* relaxado substitui um convencional, pode-se concluir que a maneira mais eficiente e eficaz para verificar módulos exibindo comportamentos fora de ordem sob uma abordagem *black-box* é a utilização de *scoreboard* relaxados projetados com uma política de primeiro emparelhamento e uma regra de atualização baseada na remoção de dominadores.

4 VERIFICAÇÃO DE CONSISTÊNCIA DE MEMÓRIA

Talvez a pergunta mais primitiva que possa ser feita sobre o comportamento de um sistema de memória é: *qual* é o valor retornado por uma instrução de leitura? A resposta não é trivial para multiprocessadores, pois a "última" instrução de escrita para o mesmo endereço não é precisamente especificada pela ordem de programa. Portanto, a programação paralela com memória compartilhada requer a definição de semântica de memória por meio de um MCM [Adve e Gharachorloo 1996], o qual, essencialmente, especifica *quando* um valor armazenado deve ser visto por uma instrução de leitura. A forma mais simples de se definir precisamente a semântica de memória é impor uma ordem sequencial para todas as operações, dando origem a um MCM denominado *Sequential Consistency* [Adve e Gharachorloo 1996]. Esse modelo, entretanto, restringe excessivamente o uso de otimizações no programa (em tempo de compilação) e no *hardware* (em tempo de execução). Por esse motivo, MCMs relaxados são necessários, tais como *Weak Ordering* (empregado pelo ARMv7), *Total Store Ordering* [Hangal et al. 2004] (utilizado pelo SPARC), e os modelos ainda mais relaxados adotados pelos processadores Alpha [Sites e Witek 1995] e PowerPC [May et al. 1994].

Um MCM não somente afeta o modo como os programas paralelos são escritos, mas também como o *hardware* paralelo é projetado. O uso de vários recursos arquiteturais para a melhoria de desempenho é afetado pelo MCM adotado. Por exemplo, múltiplas unidades de leitura/escrita, filas de escrita com adiantamento de leituras (*read bypassing*), múltiplos módulos de memória, *caches* não-bloqueantes, protocolos de coerência de *cache* etc. Como muitos aspectos do projeto de um *hardware* paralelo são afetados, a verificação de sua conformidade com um MCM é um problema relevante, cuja complexidade depende da observabilidade dos eventos de memória.

Esta dissertação explora a observabilidade estendida de uma representação executável para acelerar a verificação de consistência, ao invés de reusar – em tempo de projeto – métodos originalmente desenvolvidos para testes pós-fabricação. A maior observabilidade é explorada através da sobreamostragem de eventos de memória. Ao invés de amostrar uma única sequência de eventos por processador [Shacham et al. 2008, Hangal et al. 2004, Manovit e Hangal 2005, Roy et al. 2006, Manovit e Hangal 2006, Chen et al. 2009, Hu et al. 2012] ou duas sequências [Rambo, Henschel e Santos 2012], foram monitorados três pontos por processador. Ao invés de tentar inferir a ordem das relações entre operações de memória [Hangal et al. 2004, Manovit e Hangal 2005, Roy et al. 2006, Manovit e Hangal 2006, Chen et al. 2009, Hu et al. 2012] ou recorrer ao emparelhamento de grafos bipartidos [Rambo, Henschel

e Santos 2012], a técnica aqui proposta verifica a equivalência de sequências de valores esperados e observados por meio de *múltiplos scoreboards* relaxados concorrentes, em vez de um único *scoreboard* relaxado, como o proposto em [Shacham et al. 2008].

O restante deste capítulo está organizado como segue. A Seção 4.1 formula o problema alvo. A Seção 4.2 apresenta uma revisão dos trabalhos correlatos. A Seção 4.3 formaliza os algoritmos que suportam a técnica proposta. A Seção 4.4 fornece as garantias teóricas. A Seção 4.5 relata os resultados experimentais, comparando a técnica proposta com dois *checkers post-mortem* em termos de eficácia e eficiência. Por fim, na Seção 4.6, os resultados teóricos e experimentais são postos em perspectiva.

4.1 PROBLEMA ALVO

Para estabelecer claramente as ligações com as técnicas correlatas mais próximas da proposta, foram emprestadas algumas notações utilizadas em [Hangal et al. 2004] e [Rambo, Henschel e Santos 2012] com algumas adaptações.

Um MCM é especificado por dois tipos de axiomas. **Axiomas de ordem** definem o grau de relaxamento em relação à ordem de programa. Um **axioma de valor** restringe os valores que podem ser retornados por uma leitura. Descrições formais de tais axiomas podem ser encontradas na literatura (por exemplo, [Hangal et al. 2004] e [Rambo, Henschel e Santos 2012]) para MCMs distintos.

A verificação de consistência requer a observação de sequências de eventos de memória, denominados *traces*. Há diferentes tipos de operações de memória. Um *Load* e um *Store* representam uma leitura e uma escrita de um valor, respectivamente. Um *Swap* é uma única operação que realiza a leitura de um valor que estava armazenado na memória e, "ao mesmo tempo", a escrita de um novo valor naquela mesma posição de memória. Por fim, uma *Membar* representa uma barreira de memória que impõe uma restrição de precedência: as instruções (de acesso a memória) que a precedem na ordem de programa devem ser concluídas, durante a execução, antes que as suas sucessoras sejam executadas.

Definição 5. *Um trace é uma sequência* $(\tau_1, \tau_2, \cdots \tau_j, \cdots \tau_m)$*, onde* $\tau_j = (op, a, v)$ *é um evento de memória tal que:*

- $op \in \{Load, Store, Swap, Membar\}$*, a é um endereço e v é um valor;*
- *v é o valor lido ou escrito em memória em um endereço a, exceto para* $op = Membar$*, quando* $v = a = NIL$*.*

Seja n o *número de operações de memória* de um programa paralelo e p o *número de processadores*. A verificação de consistência de memória pode ser formalizada como segue:

Problema 2. *Dada uma coleção de traces $T_1, T_2, \cdots T_p$, existe um trace global*[1] *T que satisfaça todos os axiomas de um MCM?*

4.2 TRABALHOS RELACIONADOS

O modo como o Problema 2 é abordado por vários autores é resumido na Tabela 3. Para cada *checker*, a tabela mostra se ele pode ser utilizado para verificação em tempo de projeto (pré-fabricação) ou teste de protótipo (pós-fabricação), se a análise é feita durante a simulação (*on-the-fly*) ou após (*post-mortem*), e se ele oferece ou não plenas garantias de encontrar um erro exposto por um dado caso de teste. Por fim, ela mostra a observabilidade exigida e o seu impacto na complexidade de pior caso. A última linha da tabela contrasta o *checker* proposto na próxima seção com os trabalhos relacionados a seguir analisados.

A maioria dos *checkers* requer a disponibilização de todos os *traces* antes de iniciar a verificação e utilizam grafos orientados acíclicos (DAG, *directed acyclic graph*) para codificar relações de ordem inferidas dos *traces*. A detecção de um ciclo em um grafo orientado é uma prova de inconsistência de memória. No entanto, o fato de nenhum ciclo ser detectado não prova a consistência, pois pode não ter sido possível inferir alguma relação entre operações. Para excluir eventuais falsos negativos que podem resultar das limitações do mecanismo de inferência, alguns *checkers* recorrem a *backtracking* [Manovit e Hangal 2006, Chen et al. 2009, Hu et al. 2012], o que resulta em longos tempos de execução, especialmente quando o número de processadores aumenta. Um método pré-fabricação [Rambo, Henschel e Santos 2012] oferece garantias similares sem a necessidade de *backtracking*. Ele monitora duas sequências de eventos por processador. Apesar de sua alta complexidade de pior caso, evidências experimentais mostram que o esforço computacional médio daquela técnica é bem menor quando utilizados testes com instruções geradas aleatoriamente. Os autores reutilizaram sem adaptações um algoritmo de emparelhamento [Marcilio et al. 2009] projetado para resolver um problema mais geral do que o Problema 2. Devido a essa falta de adaptação, o método proposto por [Rambo, Henschel e Santos 2012] (para resolver um problema mais simples) herdou, desnecessariamente, a maior

[1] Um trace global é um trace que contém todas as operações de memória associadas a qualquer processador.

Tabela 3: Comparação qualitativa entre classes distintas de *checkers* de consistência

(legenda: G = garantias plenas, A = número de pontos de amostragem por processador)

Técnica	Uso	Tipo	Ideia chave	G	A	Complexidade
[Hangal et al. 2004]	(Pré-) Pós-fabricação	*Post-mortem*	Inferência baseada em DAG		1	$O(n^5)$
[Manovit e Hangal 2005]	(Pré-) Pós-fabricação	*Post-mortem*	Inferência baseada em DAG		1	$O(pn^3)$
[Manovit e Hangal 2006]	(Pré-) Pós-fabricação	*Post-mortem*	Inferência baseada em DAG	✓	1	$O((n/p)^p pn^3)$
[Roy et al. 2006]	(Pré-) Pós-fabricação	*Post-mortem*	Inferência baseada em DAG		1	$O(n^4)$
[Chen et al. 2009]	(Pré-) Pós-fabricação	*Post-mortem*	Inferência baseada em DAG	 ✓	1	$O(p^3n)$ $O(p^2n^2)$
[Hu et al. 2012]	(Pré-) Pós-fabricação	*Post-mortem*	Inferência baseada em DAG	✓	1	$O(p^3n)$
[Shacham et al. 2008]	Pré-fabricação	*On-the-fly*	*Scoreboard* relaxado único		1	$O(p^2n^2)$
[Rambo, Henschel e Santos 2012]	Pré-fabricação	*Post-mortem*	Emparelhamento de grafo bipartido	✓	2	$O(n^6/p^5)$
Este trabalho	Pré-fabricação	*On-the-fly*	Múltiplos *scoreboards* relaxados concorrentes	✓	3	$O(n^2/p)$

complexidade do método proposto por [Marcilio et al. 2009] para resolver um problema mais complexo.

Um método recente [Hu et al. 2012] afirma que a verificação de MCM pode ser realizada em tempo linear. Embora isso seja válido para complexidade de pior caso (veja Tabela 3), para um dado tamanho de caso de teste, o tempo de verificação cresce exponencialmente com o número de processadores na proporção de C^p, onde C é uma constante. Além disso, a verificação em tempo linear só é possível quando o multiprocessador possui um relógio global[2].

Como mostra a Tabela 3, um *checker on-the-fly* é raramente utilizado para verificar um MCM pelos seguintes motivos. Um *scoreboard* convencional, que é um *checker* muito eficiente, não é capaz de tratar diretamente um subsistema que não preserva em sua saída a ordem correspondente aos estímulos de entrada, como já explicado no Capítulo 2. Para permitir a verificação dos *valores*, os domínios temporais do *scoreboard* e do DUV devem ser sincronizados. Entretanto, a sincronização tem dois efeitos colaterais: 1) limita a capacidade do *scoreboard* de verificar se a *ordem* dos valores obedece as restrições de ordem pré-especificadas (neste caso, pelos MCMs) e 2) ela serializa, em tempo de verificação, eventos de memória, que são concorrentes em tempo de execução (como acontece no *hardware* paralelo).

É por isso que o uso de um *scoreboard relaxado* foi proposto para verificação de MCMs [Shacham et al. 2008]. Ao invés de armazenar um único valor esperado para cada estímulo de entrada (como faz um *scoreboard* convencional), o *scoreboard* relaxado mantém múltiplos valores esperados enquanto um único valor não puder ser deterministicamente identificado. Ele utiliza uma regra de atualização que armazena um novo valor após cada escrita e, dinamicamente, remove do *scoreboard* os valores que se tornam inválidos após cada leitura. Como resultado, o número de valores esperados para cada estímulo diminui progressivamente. Apesar de representar uma solução muito eficiente para substituir *checkers* baseados em inferência [Hangal et al. 2004, Manovit e Hangal 2005, Roy et al. 2006, Manovit e Hangal 2006, Chen et al. 2009, Hu et al. 2012], os autores reconhecem que ele pode induzir falsos negativos [Shacham et al. 2008], pois o *scoreboard* relaxado nunca reconsidera uma decisão já tomada.

Como as garantias de verificação de um *scoreboard* relaxado *único* é limitada, esta dissertação propõe um método que utiliza *múltiplos scoreboards* relaxados, os quais verificam — concorrentemente — a consistência sob

[2]Pois pode-se associar instantes de tempo ao início e ao fim de cada instrução, obtendo assim uma ordenação parcial global, que é explorada para reduzir o problema de verificação a uma instância de menor complexidade.

a perspectiva de cada processador individualmente, como descrito na próxima Seção.

4.3 O *CHECKER ON-THE-FLY* PROPOSTO

Como os *scoreboards* convencional e relaxado podem induzir falsos negativos e as garantias plenas de um *checker post-mortem* pré-fabricação só se viabilizam às custas de alto esforço de verificação (frequentemente devido a *backtracking*), o *checker* proposto é construído sobre os seguintes fundamentos:

1. *Amostragem em ordem de programa*: operações de memória são monitoradas na saída da unidade de consolidação de resultados[3] de cada processador;

2. *Amostragem em ordem de execução*: operações de memória são monitoradas na interface de cada processador com a sua *cache* privada;

3. *Verificação de operações não-consolidadas*: O *buffer* de reordenamento é monitorado para identificar operações não-consolidadas.

Os dois primeiros monitores evitam que a observabilidade limitada de eventos de memória possa conduzir a falsos negativos. O terceiro monitor impede que artefatos de implementação (tais como operações especulativas) possam levar a falsos positivos.

Note que os pontos a serem monitorados foram escolhidos judiciosamente para serem amplamente independentes das especificidades de uma microarquitetura com suporte a execução fora de ordem. De agora em diante, denotam-se por i^+, i^- e i^* os monitores posicionados na unidade de consolidação do processador i, na interface entre o processador i e sua *cache* privada e no seu *buffer* de reordenamento, respectivamente.

A solução do Problema 2 foi decomposta em dois subproblemas, que abordados por dois tipos complementares de *checkers*: um individualmente é usado para verificar a consistência do ponto de vista de cada processador, o outro é usado para verificar a consistência sob uma perspectiva global. Deve-se notar que ambos tratam apenas operações visíveis por todos os processadores. Operações que nunca atingem a interface de memória (exemplo:

[3] Do inglês, *commit unit* [Patterson e Hennessy 2005]; ou seja, a unidade que armazena num *buffer* de reordenamento os valores gerados pela execução especulativa de leituras e desvios até que seja resolvida a incerteza sobre a hipótese de especulação, quando os resultados são consolidados em memória ou registrador (se a hipótese for verdadeira ou são *descartados* em caso contrário).

read-on-write-early [Adve e Gharachorloo 1996]) podem ser tratadas separadamente[4].

Foram empregados *p checkers* locais *independentes*, cada um contando com seu *scoreboard* relaxado, e um único *checker* global. Tão logo o *scoreboard* detecte uma inconsistência, um erro é sinalizado. Se as sequências monitoradas são localmente consistentes para todos os processadores, o *checker* global é então invocado para verificar se o valor retornado por cada leitura é único sob uma perspectiva global e indicar um erro em caso contrário.

Para construir o *checker* global, foi adotado o algoritmo global-behavior-ok proposto em [Rambo, Henschel e Santos 2012], pois ao empregar uma ordem linear induzida pela execução de um *dado* caso de teste, ele evita a ineficiência que resultaria do uso de *backtracking* para inferir ordens válidas [Manovit e Hangal 2006, Chen et al. 2009, Roy et al. 2006, Hu et al. 2012], um mecanismo que se torna impraticável com o aumento no número de processadores.

Para construir os *checkers* locais, foi projetada uma *nova* técnica que instancia um *scoreboard* relaxado por processador. Cada instância de *scoreboard* aguarda por eventos observados pelos monitores i^+, i^- e i^* de cada processador i para atualizar, continuamente, os *conjuntos de eventos monitorados* V_i^+, V_i^- e V_i^*.

Seja $\leq$ a ordem parcial especificada pelos axiomas (de ordem) de um dado MCM, incluindo o axioma de *Membar* [Hangal et al. 2004]. Apesar de as *Membars* serem monitoradas por i^+, elas não são observadas por i^- (pois não chegam no sistema de memória). Como os efeitos decorrentes das barreiras de memória (*Membars*) são capturados pela ordem $\leq$, a verificação de consistência pode ser reduzida[5], sem prejuízo das garantias de verificação, à análise de sequências livres de *Membars*. Portanto, os conjuntos V_i^+, V_i^- e V_i^* contém apenas leituras, escritas e *swaps*.

Dado um processador i, seu *checker* local verifica se as sequências monitoradas nos pontos i^+ e i^- são consistentes. Duas sequências são consideradas consistentes quando três condições são simultaneamente satisfeitas: 1) todo evento consolidado é equivalente a um evento observado na interface de memória, 2) um evento observado é consolidado apenas uma vez e 3) os eventos observados que foram consolidados satisfazem à ordem parcial $\leq$, especificada pelos axiomas de um dado MCM. Essa noção é formalizada a seguir.

[4]Como mostrado em [Rambo 2011].

[5]Esta redução consiste em substituir vértices de V^+ que representariam *Membars* pela inclusão de novos pares ordenados em $\leq$, representando o efeito das *Membars* como restrições de precedência.

Definição 6. *Dada uma ordem parcial $\leq$ sobre o conjunto $V^+ = \{v_1^+, v_2^+, \cdots, v_N^+\}$ e uma relação de equivalência $R = \{(j,m) \in \{1,\cdots,N\} \times \{1,\cdots,M\} : v_j^+ \equiv v_m^-\}$, dizemos que as sequências $(v_1^+, v_2^+, \cdots, v_N^+)$ e $(v_1^-, v_2^-, \cdots, v_M^-)$ são consistentes se e somente se existir um mapeamento $\mu : \{1,\cdots,N\} \mapsto \{1,\cdots,M\}$ tal que:*

1. *μ é uma função tal que $\mu \subseteq R$;*
2. *μ é injetora;*
3. $\forall k, j \in \{1,\cdots,N\} : (v_k^+ \leq v_j^+) \wedge (\mu(k) = t) \wedge (\mu(j) = m) \Rightarrow (t < m)$.

Um evento observado em i^- é o resultado de uma dentre duas condições mutuamente exclusivas: 1) um evento consolidado e observado em i^+, 2) um evento não-consolidado e observado em i^*; ou seja, $|V_i^-| = |V_i^+| + |V_i^*|$. Essa noção, crucial para as provas do Apêndice B, é formalizada a seguir.

Propriedade 1. *Seja $\sigma : V_i^- \mapsto V_i^*$ uma função injetora tal que $\sigma \subseteq \{(v_m^-, v_s^*) \in V_i^- \times V_i^* : v_m^- \equiv v_s^*\}$. Para todo $v_j^- \in V_i^-$, somente uma das seguintes condições é válida:*

1. $(\exists m \in \{1,\cdots,M\} : \mu(j) = m) \wedge (\nexists v_s^* \in V_i^* : \sigma(v_m^-) = v_s^*)$
2. $(\nexists m \in \{1,\cdots,M\} : \mu(j) = m) \wedge (\exists v_s^* \in V_i^* : \sigma(v_m^-) = v_s^*)$

A técnica proposta baseia-se em três algoritmos desenvolvidos no âmbito desta dissertação, apresentados a seguir, e um algoritmo reusado de trabalho correlato (cuja descrição pode ser encontrada em [Rambo, Henschel e Santos 2012]). O Algoritmo 2[6] monitora eventos, continuamente, até o fim da simulação. Ele retorna falso assim que uma inconsistência é detectada. Após a simulação, ele retorna verdadeiro se todos os eventos consolidados foram emparelhados e todos os eventos observados na interface de memória obedeceram à Propriedade 1. Ele retorna falso se algum evento consolidado não foi emparelhado ou se a Propriedade 1 foi violada. Na linha 5, ele invoca a função match(i, v_m^-), onde está implementado o novo mecanismo de verificação descrito pelo Algoritmo 3.

Essencialmente, o Algoritmo 3 retorna falso se nenhum evento consolidado for equivalente ao evento v_m^- observado ou se o seu emparelhamento a um valor equivalente violaria a ordem de eventos preespecificada ($\leq$). Primeiro, ele encontra os eventos consolidados que são equivalentes a um dado v_m^- (linha 2). Se algum for encontrado (linha 3), o emparelhamento se dá com o primeiro v_j^+ (linha 4). Então, encontra-se o conjunto de dominadores de v_j^+ em relação à ordem $\leq$ (linha 5).

[6] Este e os demais algoritmos desta seção assumem, por simplicidade, a disponibilidade global de V^+, V^- e $\leq$.

Algoritmo 2 LOCAL-BEHAVIOR-OK(i)

1: $m \leftarrow 1$
2: **seja** T_i^- uma sequência vazia de eventos
3: **repita**
4: $v_m^- \leftarrow$monitor(i^-)
5: **se** $\neg$ match(i, v_m^-) **então**
6: **retorne falso**
7: **fim se**
8: $m \leftarrow m+1$
9: concatene v_m^- a T_i^-
10: **até** fim-da-simulação
11: **se** $(V_i^+ = \emptyset) \wedge (V_i^- = \emptyset)$ **então**
12: **retorne verdadeiro**
13: **senão**
14: **retorne falso**
15: **fim se**

Se nenhum evento equivalente for encontrado ($Q = \emptyset$), isso significa que um evento consolidado não foi observado na interface de memória ou v_m^- não foi consolidado (ele foi descartado). De forma similar, se, dentre os eventos consolidados ainda não emparelhados, existir algum dominador para v_j^+ ($D \neq \emptyset$), o emparelhamento de v_j^+ com v_m^- violaria a ordem $\leq$, a menos que v_m^- fosse um evento descartado. Por esse motivo, em ambos os cenários (linha 7), o algoritmo primeiramente busca por eventos descartados equivalentes a v_m^- (linha 8). Como nestas condições, a inexistência de evento descartado (linha 9) é uma prova de inconsistência, o algoritmo retorna falso. Caso contrário, ele seleciona para v_m^- um v_x^* arbitrário[7] entre os eventos descartados (linha 12) e o remove do *scoreboard* (linha 13). Nesse caso, v_j^+ é mantido no *scoreboard* (pois ainda pode ser emparelhado com um evento posterior).

Do contrário, quando um evento v_j^+ equivalente for encontrado e não possuir dominador, v_j^+ é considerado o emparelhamento adequado para v_m^- e é removido do *scoreboard* (linha 15). Finalmente, independentemente de v_m^- ter sido emparelhado com um evento consolidado ou não-consolidado, ele é removido do *scoreboard* (linha 17).

O Algoritmo 4 integra as verificações — local e global — de consistência de memória. A inovação do *checker* proposto está exatamente no uso de *scoreboards* relaxados para avaliar *concorrentemente* LOCAL-BEHAVIOR-OK(i) para cada i (linha 2). Após a verificação de consistência local para todos os p processadores, uma verificação global é necessária (linha 6). Como

[7] Pois não há restrição de ordenamento entre os eventos descartados.

Algoritmo 3 match(i, v_m^-)

1: $D \leftarrow \emptyset$
2: $Q \leftarrow \{v_q^+ \in V_i^+ : v_q^+ \equiv v_m^-\}$
3: **se** $Q \neq \emptyset$ **então**
4: $\quad j \leftarrow \min\{1 \leq q \leq |V_i^+| : v_q^+ \in Q\}$
5: $\quad D \leftarrow \{v_k^+ \in V_i^+ : v_k^+ \leq v_j^+\}$
6: **fim se**
7: **se** $Q = \emptyset \vee D \neq \emptyset$ **então**
8: $\quad S \leftarrow \{v_s^* \in V_i^* : v_s^* \equiv v_m^-\}$
9: $\quad$ **se** $S = \emptyset$ **então**
10: $\quad\quad$ **retorne falso**
11: $\quad$ **fim se**
12: $\quad x \leftarrow$ elemento arbitrário de $\{1 \leq s \leq |V_i^*| : v_s^* \in S\}$
13: $\quad V_i^* \leftarrow V_i^* - \{v_x^*\}$
14: **senão**
15: $\quad V_i^+ \leftarrow V_i^+ - \{v_j^+\}$
16: **fim se**
17: $V_i^- \leftarrow V_i^- - \{v_m^-\}$
18: **retorne verdadeiro**

Algoritmo 4 behavior-ok()

1: **for** $i \leftarrow 1$ to p **do in parallel**
2: $\quad$ **se** ¬LOCAL-BEHAVIOR-OK(i) **então**
3: $\quad\quad$ **retorne falso**
4: $\quad$ **fim se**
5: **fim for**
6: **retorne** global-behavior-ok$(T_1^-, T_2^-, \cdots, T_p^-)$

já justificado, foi reutilizado o algoritmo global-behavior-ok, que é formalmente descrito em [Rambo, Henschel e Santos 2012]. Essencialmente, o algoritmo constrói um *trace* global a partir de *traces* locais, de acordo com um ordenamento linear induzido por um dado caso de teste, e verifica a consistência no consumo de valores.

A complexidade temporal do Algoritmo 3 é dominada pelo cálculo de S (linha 8). Vamos supor que um caso de teste é um programa paralelo perfeitamente balanceado com n operações; isto é, cada processador executa exatamente n/p operações. O cenário de pior caso corresponde a uma sequência de n/p operações tal que a execução de cada uma operação cause o descarte de todo o conteúdo do *buffer* de reordenamento. Dessa forma, considerando constante o tamanho do *buffer* de reordenamento, a complexidade do Algoritmo 3 é $O(n/p)$. Como o Algoritmo 2 invoca o Algoritmo 3 no máximo

$n/p + Cn/p$ vezes (onde C é o tamanho constante do *buffer* de reordenamento), sua complexidade é $O(n^2/p^2)$. Considerando que global-behavior-ok leva $O(n \log p)$ [Rambo, Henschel e Santos 2012], a complexidade do Algoritmo 4 é $O(n^2/p)$ sob a hipótese pessimista que o seu laço paralelo (Linha 1) executa todas as p iterações sequencialmente.

4.4 GARANTIAS TEÓRICAS

Para estabelecer as garantias de verificação para a técnica proposta, foram consideradas as provas reportadas em trabalhos correlatos [Marcilio et al. 2009, Rambo, Henschel e Santos 2012] e provas complementares foram obtidas. Como o *checker* global da técnica *post-mortem* proposta em [Rambo, Henschel e Santos 2012] foi deliberadamente reutilizado, mas se substituiu cada *checker* local com uma versão *on-the-fly* (LOCAL-BEHAVIOR-OK) do algoritmo *post-mortem* original (local-behavior-ok), é possível herdar as garantias fornecidas por aquele trabalho [Rambo, Henschel e Santos 2012] se for provado que o primeiro (baseado em *scoreboard* relaxado) é equivalente ao último (baseado em emparelhamento estendido de grafos bipartidos).

Os seguintes lemas e teoremas, cujas provas formais encontram-se no Apêndice B, formam as garantias teóricas de que a técnica não induz falsos positivos nem falsos negativos.

Lema 1. *Cada invocação do Algoritmo 3 tal que $Q \neq \emptyset \wedge D = \emptyset$ remove exatamente um elemento de cada conjunto V_i^+ e V_i^- e retorna verdadeiro.*

Lema 2. *Se as sequências $(v_1^+, v_2^+, \cdots, v_N^+)$ e $(v_1^-, v_2^-, \cdots, v_M^-)$ são consistentes, então toda invocação do Algoritmo 3 tal que $Q = \emptyset$ remove exatamente um elemento de V_i^-, nenhum elemento de V_i^+ e retorna verdadeiro.*

Os raciocínios desenvolvidos nos Lemas 1 e 2 são utilizados na prova do seguinte teorema:

Teorema 3. *O Algoritmo 2 retorna verdadeiro se e somente se as sequências $(v_1^+, v_2^+, \cdots, v_N^+)$ e $(v_1^-, v_2^-, \cdots, v_M^-)$ são consistentes.*

Além disso, é possível provar o seguinte lema:

Lema 3. *O Algoritmo local-behavior-ok, descrito em [Rambo, Henschel e Santos 2012], retorna verdadeiro se e somente se as sequências $(v_1^+, v_2^+, \cdots, v_N^+)$ e $(v_1^-, v_2^-, \cdots, v_M^-)$ são consistentes.*

Ora, como o Teorema 3 e o Lema 3 garantem a equivalência do *checker* local aqui proposto com o reportado em [Rambo, Henschel e Santos

Tabela 4: Caracterização dos erros inseridos nas plataformas

ID	**Descrição**	**Localização**
e1	Escrita pendente ignorada por leitura no mesmo endereço	Fila de escritas
e2	Escritas para o mesmo endereço submetidas fora da ordem de programa	*Buffer* de reordenamento
e3	Valor incorreto de escrita pendente adiantado para requisição de leitura	Fila de escritas
e4	Valor incorreto da cache enviado para requisição de leitura	Interface da *cache* de dados
e5	Violação de barreira de memória	Controle da unidade de execução
e6	Leitura de bloco obsoleto devido a falha no mecanismo de invalidação	Interface do barramento da *cache* de dados
e7	Leitura de bloco corrompido devido a falha no mecanismo de invalidação	Interface do barramento da *cache* de dados
e8	Impedimento de redefinição do bit de validade	Lógica de controle da *cache*
e9	Valor incorreto lido por instrução de *swap*	Interface da *cache* de dados
e10	Leitura de valor obsoleto pela não-definição do *dirty bit*	Lógica de controle da *cache*

2012], este trabalho herda as mesmas garantias formais daquele trabalho. Ou seja, pode-se provar o seguinte teorema:

Teorema 4. *Para qualquer sistema de memória com coerência de caches e para qualquer MCM não requerendo ordenação total de escritas*[8]*, o Algoritmo 4 retorna verdadeiro se todos os axiomas do MCM forem válidos para os traces induzidos por um dado caso de teste.*

Informalmente, o Teorema 4 diz que, para modelos amplamente relaxados, quando se analisa o comportamento induzido por um dado caso de teste, a técnica aqui proposta nunca ignora um erro existente e nem sinaliza equivocadamente um erro inexistente.

Após estabelecer as garantias teóricas de verificação, o *checker* proposto foi comparado experimentalmente com dois *checkers post-mortem*, como reporta a próxima seção.

4.5 RESULTADOS EXPERIMENTAIS

Utilizou-se o *framework* GEM5 [Binkert et al. 2006] para construir diferentes instâncias de uma plataforma que implementa o MCM da arquitetura Alpha [Sites e Witek 1995]. Elas foram construídas com números de processadores distintos ($p \in \{2,4,8\}$) para uma configuração onde as *caches* de instruções/dados (L1) são privadas, a *cache* (L2) unificada é compartilhada e utiliza-se *snooping* para coerência. Foram gerados 240 casos de testes com instruções geradas aleatoriamente combinando diferentes números de operações ($n \in \{2K,4K,8K,16K\}$), de endereços compartilhados (2, 4, 8, 16, 32) e de *mixes* de instruções (4).

Posteriormente, foram modelados dez erros distintos, que estão descritos na Tabela 4. A partir de uma instância pré-validada como correta, foram derivadas dez instâncias defeituosas, cada uma contendo um erro distinto. Cada caso de teste foi executado em todas as plataformas defeituosas, dando origem a 2400 cenários de casos de uso. Cada cenário foi submetido ao *checker* proposto e a dois *checkers post-mortem*: um convencional *baseado em inferência* (INF) similar ao descrito em [Hangal et al. 2004] e um *checker baseado em emparelhamento estendido* (EXM) [Marcilio et al. 2009] similar ao descrito em [Rambo, Henschel e Santos 2012].

Como estimativa de *cobertura*, foi medida a proporção dos casos de uso para os quais um erro foi detectado[9]. Como estimativa para o *tempo de*

[8] Exemplos: *Relaxed Order* do Alpha e do PowerPC, *Weak Ordering* etc.

[9] Isto é, "cobertura" é igual ao número de erros detectados dividido pelo número de casos de teste.

verificação, foi medido o tempo de execução médio do caso de teste. Como o *checker* proposto é *on-the-fly*, esse tempo já captura todo o esforço de verificação. Contudo, para os *checkers post-mortem*, o *esforço* de verificação contabiliza, além do tempo de verificação, o tempo de geração dos traces.

A Figura 9 mostra o impacto do tamanho do caso de teste para um sistema com quatro processadores. Em média, tanto o *checker* proposto quanto EXM encontraram 92% dos erros enquanto INF encontrou 77%. EXM é duas ordens de magnitude mais rápido que INF. O *checker* proposto atinge a mesma cobertura que a do EXM mas requer aproximadamente 1/4 do tempo de verificação médio de EXM. Mesmo comparando apenas os casos de teste que encontraram um erro (como se pudéssemos otimisticamente assumir 100% de cobertura na prática), EXM ainda é mais rápido que INF por uma ordem de magnitude e o *checker* proposto é 3,5 vezes mais rápido que EXM.

A Figura 10 mostra o impacto do aumento no número de processadores para casos de teste de tamanho fixo ($n = 16K$). Note que, para os casos observados, a eficácia do *checker* proposto, assim como a do EXM, é maior que 90%. Observe que, como a sua complexidade diminui com o aumento de p para casos de teste com o mesmo tamanho n (Tabela 3), o tempo de verificação do EXM diminui acompanhando a redução do número de processadores. Obviamente, espera-se que, quanto maior o número de processadores, maiores sejam os casos de teste necessários para manter uma cobertura aceitável, os quais requerem tempos de verificação mais longos. Apesar de a complexidade do *checker* proposto também diminuir a medida em que se reduz p, um ligeiro aumento no tempo de verificação é observado. Isso ocorre porque, para um dado tamanho de caso de teste, a simulação leva mais tempo para um número de processadores maior, por ser necessário inicializar um maior número de *threads*. Além disso, a simulação da representação executável do *hardware* também demora mais. Observou-se que o tempo de simulação aumentou em aproximadamente 10% para cada processador incluído na plataforma.

Também foi avaliado quão eficientes são os *checkers* para atingir o *mesmo nível de cobertura*. INF foi excluído dessa avaliação, pois o seu esforço é algumas ordens de magnitude maior que EXM e que o *checker* proposto. Para plataformas com diferentes números de processadores, determinou-se o esforço de verificação para atingir níveis de cobertura de 70%, 80% e 90%. A Figura 11 mostra o esforço de verificação necessário em função da cobertura e do número de processadores (p). Conforme p aumenta de 2 até 8, o *checker* proposto requer aproximadamente de 1/4 a 3/4 do esforço de verificação global exigido por EXM para atingir o mesmo nível de cobertura. Observe que, para um dado p, o esforço de verificação do *checker* proposto é menos sensível ao nível de cobertura que o de EXM. Note que o esforço de

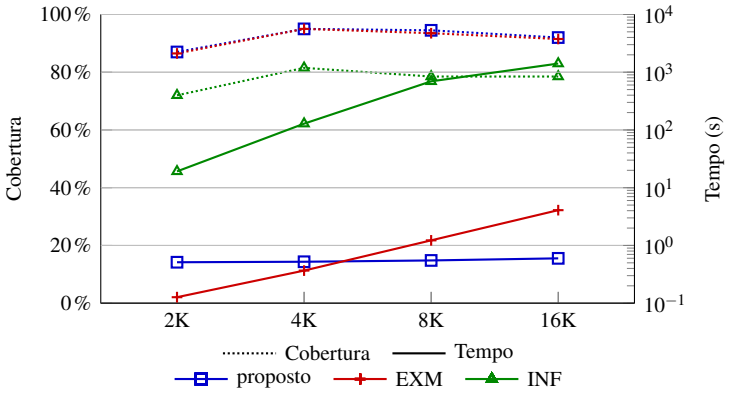


Figura 9: Impacto do tamanho crescente dos casos de teste ($p = 4$)

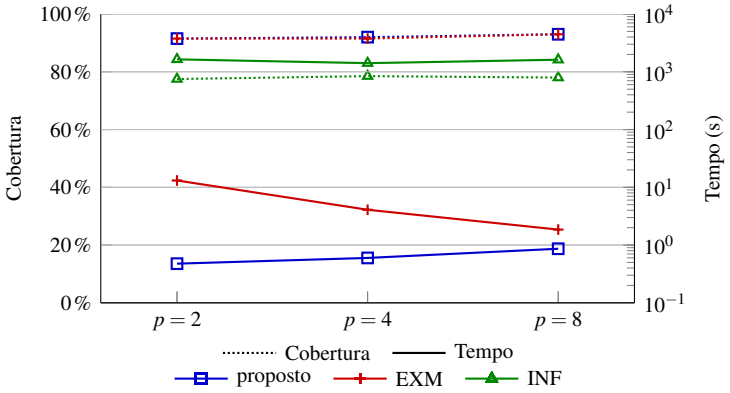


Figura 10: Impacto do aumento do número de processadores ($n = 16K$)

verificação de ambas as técnicas aumenta conforme cresce o número de processadores por dois motivos: 1) para um tamanho fixo de caso de teste, leva-se mais tempo para inicializar um maior número de *threads* e mais tempo para simular o *hardware*; 2) *casos de teste maiores* são necessários para atingir o mesmo nível de cobertura. O último explica porque o aumento do esforço é maior na Figura 11 do que é na Figura 10 (onde o tamanho do caso de teste é fixo).

Ao se combinar os resultados experimentais com as garantias teóricas, importantes conclusões podem ser tiradas sobre o impacto do *checker* proposto, como mostra a próxima seção.

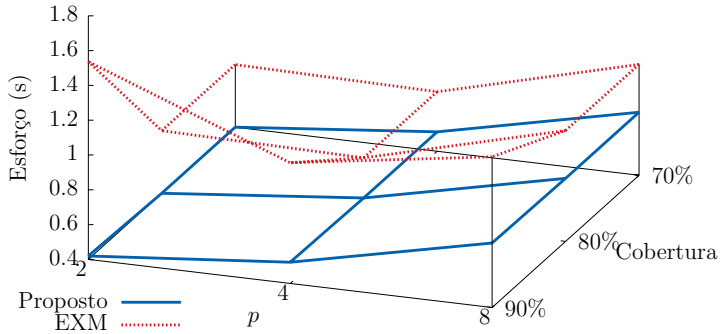


Figura 11: Impacto da cobertura e escalabilidade no esforço de verificação

4.6 CONCLUSÕES

Apesar de os *checkers* baseados em inferência convencionais serem cruciais para o teste *pós-fabricação* (devido à observabilidade limitada), os experimentos mostraram que o seu reuso é inadequado para verificação *pré-fabricação* de MCMs relaxados. Mostrou-se que vale à pena projetar *checkers* específicos para a verificação pré-fabricação, pois isso acelera o processo em 1 a 2 ordens de magnitude, mesmo quando comparados a *checkers* convencionais sem *backtracking* [Hangal et al. 2004]. Também pode-se concluir que *backtracking* não vale à pena para verificação *pré-fabricação*, pois levaria a tempos de execução ainda maiores para se obter, essencialmente, as mesmas garantias de verificação fornecidas pelos *checkers* propostos em [Rambo, Henschel e Santos 2012] e neste trabalho. Além disso, o uso de *backtracking* limita a escalabilidade[10] de longo prazo de *checkers* baseados em inferência para tratar MCMs altamente relaxados. A técnica proposta requer aproximadamente de 1/4 a 3/4 do esforço de verificação global exigido pelo *checker post-mortem* com o mesmo potencial de escalabilidade.

[10]Ou seja, a viabilidade de uso prático da técnica diante do aumento do número de processadores.

5 CONCLUSÕES E PERSPECTIVAS

5.1 CONSEQUÊNCIAS DAS TÉCNICAS PROPOSTAS

Primeiramente, esta dissertação provou que *scoreboards* projetados com uma política de primeiro emparelhamento e uma regra de atualização baseada na remoção de dominadores são capazes de determinar a equivalência entre duas representações de projeto de um acelerador em *hardware* com as mesmas garantias de verificação de um *checker post-mortem* menos eficiente. Os experimentos mostraram que o uso de um *scoreboard* convencional para verificar módulos exibindo comportamentos fora de ordem, através da sincronização dos domínios temporais de RGM e DUV, pode fazer com que alguns erros não sejam sequer estimulados, mesmo com o aumento de tamanho e de quantidade de casos de teste, como os que dependem da interação entre unidades aritméticas durante o processamento de operações em paralelo, levando a uma redução drástica da eficácia do processo de verificação. O limitado número de aceleradores utilizados nos experimentos é uma consequência da não disponibilidade em domínio público de descrições executáveis de aceleradores em hardware reais por razões de sigilo industrial. A existência de barramentos (e.g. Amba AXI [ARM Holdings]) que permitem transações fora de ordem é uma evidência de que têm sido projetados aceleradores em hardware que admitem inversão de ordem.

Este trabalho também propôs uma técnica *on-the-fly* de verificação de consistência de memória que, por ser dedicada à verificação pré-fabricação, aproveita a maior observabilidade para dispensar o uso de *backtracking*, sem com isso induzir falsos positivos e negativos. Ao decompor o problema em dois subproblemas, a verificação local e a global, conforme recomendado em [Rambo, Henschel e Santos 2012] e ao propor um novo algoritmo para verificação local, herdou-se parte das garantias de verificação daquele trabalho, mas obteve-se maior eficiência. Os pontos de monitoramento em cada processador foram escolhidos para que a técnica seja largamente independente de microarquitetura. Deve-se ressaltar que a configuração experimental utilizada para validar o *checker* proposto é próxima de configurações reais. Os casos de teste foram gerados de foram similar ao adotado por empresas como Sun [Hangal et al. 2004] e Intel [Roy et al. 2006]. Ademais, a configuração das representações executáveis utilizadas (2, 4 e 8 processadores) é compatível com as aplicações reais. Por exemplo, na data de escrita desta dissertação, a maioria dos *smartphones* usa 2 cores, os últimos lançamentos contam com 4 cores e nenhum com 8 cores.

5.2 LIMITAÇÕES E TRABALHOS FUTUROS

5.2.1 Verificação de aceleradores

Expansão do número de aceleradores:
Avaliou-se a técnica de verificação de aceleradores em *hardware* para dispositivos relativamente simples, embora tenham sido projetados para refletir a estrutura e o comportamento de dispositivos reais. Estão sendo conduzidos experimentos com casos de uso ainda mais realistas para avaliar o comportamento da técnica em termos de desempenho e eficácia e compará-la com um *scoreboard* convencional.

Abordagem *white-box*:
Uma limitação inerente da técnica desenvolvida é a impossibilidade de comparar duas sequências de eventos de tamanhos diferentes, um cenário que pode ser induzido por artefatos de implementação [Bailey, Martin e Piziali 2007]. Ao se realizar a verificação sob uma abordagem *black-box*, pode-se utilizar os elementos da interface do dispositivo (sinais que implementam o protocolo de comunicação) para descartar o efeito daqueles artefatos. Lançando mão da ferramenta desenvolvida por [Albertini et al. 2007], pretende-se estabelecer até que ponto a técnica permite a verificação de aceleradores em *hardware* sob uma abordagem *white-box* onde os efeitos dos artefatos de implementação se manifestam. Pretende-se utilizar a técnica [Marcilio et al. 2009] como referência para os resultados experimentais.

5.2.2 Verificação de consistência

Suporte a programas reais:
Para fins de **verificação** pré-fabricação, usam-se de programas paralelos artificiais, gerados aleatoriamente, como casos de teste de forma a assegurar um nível adequado de cobertura [Hangal et al. 2004, Roy et al. 2006]. Por outro lado, para a **depuração** de uma representação executável defeituosa (ou seja, para encontrar a fonte do erro reportado pela ferramenta de verificação), é desejável utilizar programas reais. Pretende-se realizar testes para determinar até que ponto a configuração atual da técnica suporta programas reais e, em caso contrário, adicionar o suporte reposicionando os monitores atuais ou, eventualmente, adicionando um novo ponto de monitoramento.

Comparação com *scoreboard* relaxado único:

Realizou-se a comparação com técnicas de verificação de consistência de memória baseadas em grafos bipartidos [Marcilio et al. 2009] e em DAGs [Hangal et al. 2004]. Está sendo implementada a infraestrutura para que seja possível realizar a comparação com a técnica baseada em um *scoreboard* único proposto em [Shacham et al. 2008].

REFERÊNCIAS BIBLIOGRÁFICAS

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

ALBERTINI, B. et al. A computational reflection mechanism to support platform debugging in systemc. In: *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*. Salzburg, Austria: ACM, 2007. p. 81–86. ISBN 978-1-59593-824-4.

ARM HOLDINGS. *AMBA Open Specifications*. Disponível em: <http://www.arm.com/products/system-ip/amba/amba-open-specifications.php>. Acesso em: jul. 2011.

ARM HOLDINGS. *The ARM Cortex-A9 Processors*. Disponível em: <http://www.arm.com/products/processors/cortex-a/cortex-a9.php>. Acesso em: set. 2009. White paper.

BAILEY, B.; MARTIN, G.; PIZIALI, A. *ESL design and verification: a prescription for electronic system level methodology*. São Francisco, EUA: Morgan Kaufmann Publishers, 2007.

BINKERT, N. et al. The M5 simulator: Modeling networked systems. *IEEE Micro*, v. 26, n. 4, p. 52–60, jul. 2006. ISSN 0272-1732.

BLACK, D. C.; DONOVAN, J. *SystemC: From the Ground Up*. [S.l.]: Kluwer Academic Publishers, 2004.

BRYANT, R. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35, n. 8, p. 677–691, ago. 1986. ISSN 0018-9340.

CHEN, Y. et al. Fast complete memory consistency verification. In: *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*. [S.l.: s.n.], 2009. p. 381–392. ISSN 1530-0897.

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Nova Iorque, EUA, v. 8, n. 2, p. 244–263, abr. 1986. ISSN 0164-0925.

FREITAS, L. S.; ANDRADE, G. G.; SANTOS, L. C. V. dos. Efficient verification of out-of-order behaviors with relaxed scoreboards. In: *The 30th IEEE International Conference on Computer Design (ICCD 2012)*. Montreal, Canada: [s.n.], 2012.

FREITAS, L. S.; ANDRADE, G. G.; SANTOS, L. C. V. dos. A template for the construction of efficient checkers with full verification guarantees. In: *The IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*. Sevilha, Espanha: [s.n.], 2012.

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 13)*. [S.l.: s.n.], 2013. **(trabalho a ser submetido)**.

HANGAL, S. et al. TSOtool: a program for verifying memory systems using the memory consistency model. In: *31st ACM/IEEE International Symposium on Computer Architecture (ISCA)*. [S.l.: s.n.], 2004. p. 114–123. ISBN 0-7695-2143-6.

HENNESSY, J.; PATTERSON, D.; ARPACI-DUSSEAU, A. *Computer Architecture: A Quantitative Approach*. 4. ed. [S.l.]: Morgan Kaufmann, 2007. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780123704900.

HU, W. et al. Linear time memory consistency verification. *IEEE Transactions on Computers*, v. 61, n. 4, p. 502–516, abr. 2012. ISSN 0018-9340.

JACOBI, C. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In: BRINKSMA, E.; LARSEN, K. (Ed.). *Computer Aided Verification*. [S.l.]: Springer, 2002, (Lecture Notes in Computer Science, v. 2404). p. 211–226.

MANOVIT, C.; HANGAL, S. Efficient algorithms for verifying memory consistency. In: *17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. [S.l.: s.n.], 2005. p. 245–252. ISBN 1-58113-986-1.

MANOVIT, C.; HANGAL, S. Completely verifying memory consistency of test program executions. In: *The Twelfth International Symposium on High-Performance Computer Architecture*. [S.l.: s.n.], 2006. p. 166–175. ISSN 1530-0897.

MARCILIO, G. et al. A novel verification technique to uncover out-of-order duv behaviors. In: *46th ACM/IEEE Design Automation Conference (DAC)*. São Francisco, California: ACM, 2009. p. 448–453. ISSN 0738-100X.

MAY, C. et al. (Ed.). *The PowerPC architecture: a specification for a new family of RISC processors*. São Francisco, EUA: Morgan Kaufmann Publishers Inc., 1994. ISBN 1-55860-316-6.

OCP-IP ASSOCIATION. *Open Core Protocol Specification 3.0*. 2009.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. 3. ed. [S.l.]: Elsevier, 2005. ISBN 1-55860-604-1.

RAMBO, E. A. *Verificação de consistência de memória para sistemas integrados multiprocessados*. Dissertação (mestrado) — Universidade Federal de Santa Catarina, 2011.

RAMBO, E. A.; HENSCHEL, O. P.; SANTOS, L. C. V. On ESL verification of memory consistency for system-on-chip multiprocessing. In: *ACM/IEEE Design, Automation & Test in Europe Conference (DATE), 2012*. [S.l.: s.n.], 2012. p. 9–14. ISSN 1530-1591.

REED, D.; GANNON, D.; LARUS, J. Imagining the future: Thoughts on computing. *IEEE Computer Magazine*, v. 45, n. 1, p. 25–30, jan. 2012. ISSN 0018-9162.

RIGO, S.; AZEVEDO, R.; SANTOS, L. *Electronic System Level Design: An Open-Source Approach*. [S.l.]: Springer, 2011.

ROY, A. et al. Fast and generalized polynomial time memory consistency verification. In: *Computer Aided Verification (CAV)*. [S.l.]: Springer, 2006, (Lecture Notes in Computer Science, v. 4144). p. 503–516. ISBN 978-3-540-37406-0.

SAHA, A. et al. A simulation-based approach to architectural verification of multiprocessor systems. In: *Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on Computer Communications (INFOCOM)*. [S.l.: s.n.], 1995. p. 34 –37.

SEGER, C. *An introduction to formal hardware verification*. [S.l.]: University of British Columbia, Department of Computer Science, 1992.

SHACHAM, O. et al. Verification of chip multiprocessor memory systems using a relaxed scoreboard. In: *41st ACM/IEEE International Symposium on Microarchitecture (MICRO)*. [S.l.: s.n.], 2008. p. 294–305. ISSN 1072-4451.

SITES, R. L.; WITEK, R. T. *Alpha AXP architecture reference manual (2nd ed.)*. Newton, MA, EUA: Digital Press, 1995.

SKAKKEBAEK, J.; JONES, R.; DILL, D. Formal verification of out-of-order execution using incremental flushing. In: HU, A.; VARDI, M. (Ed.). *Computer Aided Verification*. [S.l.]: Springer Berlin / Heidelberg, 1998, (Lecture Notes in Computer Science, v. 1427). p. 98–109. ISBN 978-3-540-64608-2. 10.1007/BFb0028737.

TEXAS INSTRUMENTS. *OMAP4470 mobile applications processor*. Disponível em: <http://www.ti.com/hdr_p_omap>. Acesso em: jun. 2012. White paper.

WILE, B.; GOSS, J. C.; ROESNER, W. *Comprehensive Functional Verification: The Complete Industry Cycle*. San Diego, EUA: Elsevier, 2005.

ÖZGüR, T. *Complex arithmetic operations*. [S.l.]: OpenCores. Disponível em: <http://opencores.org/project,complexarithmetic>. Acesso em: jul. 2011.

APÊNDICE A – Provas dos Lemas e Teoremas do Capítulo 3

Embora não tenham sido obtidas pelo autor desta dissertação, as provas formais aqui apresentadas são incluídas para tornar a dissertação autocontida. As provas foram elaboradas pelo Prof. Luiz Cláudio V. dos Santos como contrapartida de um trabalho cooperativo [Freitas, Andrade e Santos 2012] submetido à *IEEE International Conference on Electronics, Circuits, and Systems* (ICECS 2012).

Os Teoremas 1 e 2 são reapresentados a seguir juntamente com suas respectivas provas.

Teorema 1: *O algoritmo* `proper-matching`*, descrito em [Marcilio et al. 2009], nunca indica falsos negativos nem falsos positivos ao resolver o Problema 1.*

Prova. Esse algoritmo resolve um problema de emparelhamento sobre um grafo $G = (V^+ \cup V^-, E)$, com $V^+ \cap V^- = \emptyset$, tal que $|V^+| \leq |V^-|$ e $E = \{(v^+, v^-) \in V^+ \times V^- | v^+ \sim v^-\}$, onde $\sim$ denota uma relação (não necessariamente transitiva) de compatibilidade entre valores. Em [Marcilio et al. 2009] dois teoremas são provados: o primeiro estabelece que o algoritmo nunca induz falsos negativos; o segundo estabelece que ele nunca induz falsos positivos se e somente se $E = \{(v^+, v^-) \in V^+ \times V^- | v^+ \equiv v^-\}$ onde $\equiv$ denota uma relação de equivalência entre valores. Como o Problema 1 é uma instância do problema E-matching tal que $|V^+| = |V^-|$ (Definição 1, primeira cláusula) e $v^+ \equiv v^-$ para cada $(v^+, v^-) \in E$ (Definição 1, segunda cláusula), `proper-matching` nunca induz resultados falsos ao resolver o Problema 1. □

Teorema 2: *Sob a perspectiva de um dado caso de teste, o Algoritmo 1 resolve o Problema 1 com plenas garantias de verificação.*

Prova. Suposição 1 - DUV correto: Em um DUV correto, para todo vértice $v_n^- \in V^-$ existe um vértice $v_j^+ \in V^+$ tal que $v_j^+ \equiv v_n^-$ e vice-versa (Definição 1, primeira e segunda cláusulas), isto é $|V^+| = |V^-|$. Vamos provar por contradição que $\{v_k^+ \in V^+ | v_k^+ \leq v_j^+\} = \emptyset$ para uma iteração arbitrária do laço. Supondo provisoriamente que $\{v_k^+ \in V^+ | v_k^+ \leq v_j^+\} \neq \emptyset$, deve existir pelo menos um v_k^+ para o qual um valor equivalente, digamos v_m^-, ainda não foi encontrado na iteração n. Em outras palavras, como o DUV é correto, deve existir um v_m^- tal que $v_k^+ \equiv v_m^-$ com $m > n$. Na representação de um grafo bipartido, isso pode ser descrito como $(v_j^+, v_n^-), (v_k^+, v_m^-) \in E$ com $m > n$. Como $v_k^+ \leq v_j^+$, conclui-se da Definição 4 que $\chi((v_j^+, v_n^-), (v_k^+, v_m^-))$ = verdadeiro. Como um DUV correto não pode induzir um cruzamento impróprio (pois ele violaria a terceira cláusula da Definição 1, contradizendo assim, via Definição 2, a suposta equivalência funcional a um RGM pré-validado), conclui-se que $\{v_k^+ \in V^+ | v_k^+ \leq v_j^+\} = \emptyset$ para todo $v_n^- \in V^-$. Como $|V^+| = |V^-|$, o laço

executa exatamente N vezes e o Algoritmo 1 retorna verdadeiro.
Suposição 2 - DUV incorreto: Pelo menos um dos seguintes cenários é válido para um DUV incorreto.

Suposição 2.1 - Incompatibilidade de valor: Neste cenário, existe pelo menos um $v_n^- \in V^-$ para o qual nenhum $v_j^+ \in V^+$ correspondente pôde ser encontrado, isto é $Q = \emptyset$ (na linha 4) para pelo menos uma iteração do laço. Assim, o Algoritmo 1 retorna falso.

Suposição 2.2 - Violação de ordem: Neste cenário, existe pelo menos um cruzamento impróprio de arestas, isto é, existe pelo menos um $v_n^- \in V^-$ cujo equivalente $v_j^+ \in V^+$ é tal que existe uma aresta (v_k^+, v_m^-) para a qual $\chi((v_j^+, v_n^-), (v_k^+, v_m^-))$ = verdadeiro. Como resultado, conclui-se que $v_k^+ \leq v_j^+$ para $m > n$. Como v_k^+ é removido de V^+ (na linha 9) na iteração n, o Algoritmo 1 eventualmente retornará falso no máximo em alguma iteração, digamos y, tal que $v_k^+ \equiv v_y^-$, pois v_y^- resultará não emparelhado ($Q = \emptyset$ na linha 4).

Como ele sempre retorna verdadeiro para um DUV correto e falso para um DUV incorreto, o Algoritmo 1 não induz falsos negativos e nem falsos positivos para um dado caso de teste. □

APÊNDICE B – Provas dos Lemas e Teoremas do Capítulo 4

Embora não tenham sido obtidas pelo autor desta dissertação, as provas formais aqui apresentadas são incluídas para tornar a dissertação autocontida. As provas foram elaboradas pelo Prof. Luiz Cláudio V. dos Santos como contrapartida de um trabalho cooperativo [Freitas, Rambo e Santos 2013] que será submetido ao evento *Design, Automation and Test in Europe* (DATE 2013).

Os Lemas 1, 2 e 3 e os Teoremas 3 e 4 são reapresentados a seguir juntamente com suas respectivas provas.

Lema 1: *Cada invocação do Algoritmo 3 tal que* $Q \neq \emptyset \wedge D = \emptyset$ *remove exatamente um elemento de cada conjunto* V_i^+ *e* V_i^-*, e retorna verdadeiro.*

Prova. Seja match(i, v_m^-) uma invocação do Algoritmo 3 tal que $Q \neq \emptyset \wedge D = \emptyset$. Como $(Q \neq \emptyset) \wedge (D = \emptyset)$ é verdadeiro, um elemento (v_j^+) é removido de (V_i^+) (linha 15), um elemento (v_m^-) é removido de V_i^- (linha 17), e o algoritmo retorna verdadeiro. □

Lema 2: *Se as sequências* $(v_1^+, v_2^+, \cdots, v_N^+)$ *e* $(v_1^-, v_2^-, \cdots, v_M^-)$ *são consistentes, então toda invocação do Algoritmo 3 tal que* $Q = \emptyset$ *remove exatamente um elemento de* V_i^-*, nenhum elemento de* V_i^+ *e retorna verdadeiro.*

Prova. Por hipótese, todas as cláusulas da Definição 6 são verdadeiras. Seja match(i, v_m^-) uma invocação do Algoritmo 3 tal que $Q = \emptyset$. Como as Cláusulas 1 e 2 são verdadeiras e $Q = \emptyset$, conclui-se que $\forall j \in \{1, \cdots, N\} : \mu(j) \neq m \Rightarrow \forall v_j^+ \in V_i^+ : v_j^+ \not\equiv v_m^-$. Portanto, da Propriedade 1 (Condição 2), conclui-se que $\exists v_s^* \in V_i^* : v_m^- \equiv v_s^*$ para um dado v_m^-. Assim, como $S \neq \emptyset$, v_m^- é removido de V_i^-, v_j^+ não é removido de V_i^+, e o algoritmo retorna verdadeiro. □

Teorema 3: *O Algoritmo 2 retorna verdadeiro se e somente se as sequências* $(v_1^+, v_2^+, \cdots, v_N^+)$ *e* $(v_1^-, v_2^-, \cdots, v_M^-)$ *são consistentes.*

Prova. Suposição 1 – Sequências consistentes.

(1.1) $N = M$: Como o Algoritmo 2 invoca o Algoritmo 3 M vezes sob a condição $Q \neq \emptyset \wedge D = \emptyset$ e como $M = N$, conclui-se que N elementos são removidos de V_i^+ e M elementos de V_i^- (Lema 1). Portanto, à saída da M-ésima invocação, tem-se $(V_i^+ = \emptyset) \wedge (V_i^- = \emptyset)$. Assim, o Algoritmo 2 retorna verdadeiro.

(1.2) $N < M$: Seja $|V_i^*| = U$. o Algoritmo 2 invoca o Algoritmo 3 N vezes sob a condição $Q \neq \emptyset \wedge D = \emptyset$. Conclui-se que N elementos são removidos de V_i^+ e N elementos de V_i^- (Lema 1). Como ele invoca o Algoritmo 3 U vezes sob a condição $Q = \emptyset \vee D \neq \emptyset$, U elementos são removidos de V_i^- (Lema 2). Como $|V_i^-| = M$, $|V_i^+| = N$ e $M = N + U$, conclui-se que, após a M-ésima invocação, tem-se $(V_i^+ = \emptyset) \wedge (V_i^- = \emptyset)$. Assim, o Algoritmo 2 retorna

verdadeiro.

Suposição 2 – Sequências inconsistentes.

(2.1) μ não é uma função.

(2.1.1) Elemento do domínio não mapeado: Para algum $u \in \{1,2,\cdots,N\}$, a seguinte situação é válida: $\forall m \in \{1,2,\cdots,M\}$: $v_u^+ \not\equiv v_m^-$. Para qualquer invocação de match(i, v_m^-), tem-se $v_u^+ \notin Q$. Quando $(Q = \emptyset) \vee (D \neq \emptyset)$ é verdadeiro, v_u^+ não é removido de V_i^+. Quando $Q \neq \emptyset \wedge D = \emptyset$ é verdadeiro, v_j^+ é removido de V_i^+ mas $v_j^+ \neq v_u^+$. Como v_u^+ nunca é removido de V_i^+, o Algoritmo 2 retorna falso pois $V_i^+ \neq \emptyset$.

(2.1.2) Múltiplos mapeamentos partindo do mesmo elemento do domínio: Para algum $j \in \{1,2,\cdots,N\}$, a seguinte situação é válida: $\exists m,t,\cdots,z \in \{1,2,\cdots,M\} : v_j^+ \equiv v_m^-, v_j^+ \equiv v_t^-, \cdots, v_j^+ \equiv v_z^-$ e $\forall y \in \{j+1,\cdots,N\} : v_y^+ \not\equiv v_t^-, \cdots, v_y^+ \not\equiv v_z^-$. Portanto, como v_j^+ é removido de V_i^+, os eventos $v_t^-,\cdots,v_z^-$ nunca poderão ser removidos de V_i^- em invocações posteriores ao Algoritmo 3, levando a $V_i^- \neq \emptyset$. Assim, o Algoritmo 2 retorna falso (linha 14).

(2.2) μ é uma função não-injetora: Para algum $m \in \{1,2,\cdots, M\}$, a seguinte situação é válida: $\exists j,k,\cdots,x \in \{1,2,\cdots,N\} : \mu(j) = m \wedge \mu(k) = m \wedge \cdots \wedge \mu(x) = m$. Como μ é uma função e $\mu(k) = m$, conclui-se que $\forall y \neq m : \mu(k) \neq y, \cdots, \mu(x) \neq y$. Como o Algoritmo 3 remove v_m^- de V_i^- na m-ésima invocação, conclui-se que $v_k^+ \notin Q, \cdots, v_x^+ \notin Q$ para qualquer invocação arbitrária tal que $y \in \{m,\cdots,M\}$. Portanto, os eventos $v_k^+,\cdots,v_x^+$ nunca poderão ser removidos de V_i^+. Assim, tem-se $V_i^+ \neq \emptyset$ à saída da última invocação ao Algoritmo 3 e o Algoritmo 2 retorna falso (linha 14).

(2.3) A ordem parcial $\leq$ é violada: $\exists k,j \in \{1,\cdots,N\}$: $(v_k^+ \leq v_j^+) \wedge (\mu(j) = m) \wedge (\mu(k) = t) \wedge (t \geq m)$ e $\forall y \in \{t,\cdots,M\} : v_j^+ \not\equiv v_y^-$. Seja match$(i, v_y^-)$ uma invocação do Algoritmo 3. Como $\mu(j) = m$ é verdadeiro, v_j^+ não foi removido de V_i^+ para y tal que $1 \leq y < m$. Como $(v_k^+ \leq v_j^+) \wedge (\mu(k) = t) \wedge (t \geq m)$, conclui-se que $D \neq \emptyset$ para $m \leq y \leq t$. Portanto, v_j^+ não foi removido de V_i^+ para tais invocações. Como $\forall y \in \{t,\cdots,M\} : v_j^+ \not\equiv v_y^-$ é verdadeiro, conclui-se que $v_j^+ \notin Q$ para y tal que $t < y \leq M$. Portanto, v_j^+ não foi removido de V_i^+ para tais invocações. Como v_j^+ não é removido após M invocações do Algoritmo 3, tem-se $V_i^+ \neq \emptyset$ e o Algoritmo 2 retorna falso.

Assim, o Algoritmo 2 sempre retorna verdadeiro para sequências consistentes e falso para sequências inconsistentes. □

Lema 3: *O Algoritmo local-behavior-ok, descrito em [Rambo, Henschel e Santos 2012], retorna verdadeiro se e somente se as sequências $(v_1^+, v_2^+, \cdots, v_N^+)$ e $(v_1^-, v_2^-, \cdots, v_M^-)$ são consistentes.*

Prova. O Algoritmo local-behavior-ok encontra um *emparelhamento próprio* $\mathscr{M}$ [Marcilio et al. 2009] em um grafo bipartido (V,E) com $V = V_i^+ \cup V_i^-$, $V_i^+ \cap V_i^- = \emptyset$, e $E = \{(v^+,v^-) \in V_i^+ \times V_i^- : v^+ \equiv v^-\}$ sujeito a uma ordem parcial $\leq$ sobre o conjunto V_i^+ e tal que as seguintes condições são todas válidas: 1) $\mathscr{M} \subseteq E$ é um emparelhamento; 2) $|\mathscr{M}| = |V_i^+|$; 3) $\mathscr{M} = \{(v^+,v^-) \in E : v^+ \equiv v^-\}$; 4) $\forall (v_j^+, v_m^-), (v_k^+, v_t^-) \in \mathscr{M} : ((v_j^+ \leq v_k^+) \wedge (m < t)) \vee ((v_k^+ \leq v_j^+) \wedge (t < m))$. As Condições 2 e 3 garantem que $\mathscr{M}$ induz um emparelhamento $\mu : \{1,2,\cdots,N\} \mapsto \{1,2,\cdots,M\}$, que é uma função $\mu \subseteq R$; isto é, a Cláusula 1 da Definição 6 é verdadeira. A Condição 1 garante que μ é injetora; isto é, a Cláusula 2 da Definição 6 é verdadeira. A Condição 4 garante que $\forall v_k^+, v_j^+ \in V^+ : (v_k^+ \leq v_j^+) \wedge (\mu(k) = t) \wedge (\mu(j) = m) \Rightarrow (t < m)$; isto é, a Cláusula 3 da Definição 6 é verdadeira. □

Teorema 4: *Para qualquer sistema de memória com coerência de caches e para qualquer MCM não requerendo ordenação total de escritas*[1]*, o Algoritmo 4 retorna verdadeiro se todos os axiomas do MCM forem válidos para os traces induzidos por um dado caso de teste.*

Prova. Este teorema está provado em [Rambo, Henschel e Santos 2012] para um algoritmo semelhante ao Algoritmo 4, exceto que todas as invocações de local-behavior-ok foram substituídas por uma invocação de LOCAL-BEHAVIOR-OK. Como, pelo Teorema 3 e pelo Lema 3, esses algoritmos são indistinguíveis para as mesmas sequências, a prova fornecida em [Rambo, Henschel e Santos 2012] serve como prova para este teorema. □

[1]Exemplos: *Relaxed Order* do Alpha e do PowerPC, *Weak Ordering* etc.