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

Eberle Andrey Rambo

VERIFICAÇÃO DE CONSISTÊNCIA DE MEMÓRIA PARA SISTEMAS INTEGRADOS MULTIPROCESSADOS

Florianópolis (SC)

2011

Eberle Andrey Rambo

VERIFICAÇÃO DE CONSISTÊNCIA DE MEMÓRIA PARA SISTEMAS INTEGRADOS MULTIPROCESSADOS

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

Florianópolis (SC)

2011

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

R167v Rambo, Eberle Andrey
Verificação de consistência de memória para sistemas integrados multiprocessados [dissertação] / Eberle Andrey Rambo ; orientador, Luiz Cláudio Villar dos Santos. – Florianópolis, SC, 2011.
95 p.: il., grafs., tabs.

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

Inclui referências

1. Informática. 2. Ciência da computação. 3. Memória compartilhada distribuída. 4. Grafos de ligação. I. Santos, Luiz Cláudio Villar dos. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 681

Eberle Andrey Rambo

VERIFICAÇÃO DE CONSISTÊNCIA DE MEMÓRIA PARA SISTEMAS INTEGRADOS MULTIPROCESSADOS

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

Florianópolis (SC), 21 de Novembro de 2011.

Ronaldo dos Santos Mello, Prof. Dr.
Coordenador

Banca Examinadora:

Luiz Cláudio Villar dos Santos, Prof. Dr.
Orientador

Flávio Rech Wagner, Prof. Dr.

Djones Vinicius Lettnin, Prof. Dr.

Olinto José Varela Furtado, Prof. Dr.

José Luís Almada Güntzel, Prof. Dr.

Ao bondoso Criador e à minha família, por todo o cuidado e amor.

AGRADECIMENTOS

Aos meus pais e irmãos, Mário, Muriel, Estéfani, Gabriel e Juliano, e aos meus avós, Soeli e Marlene, por todo o apoio e compreensão em diversas formas em todas as etapas até agora.

À minha noiva, Joanna, pela paciência, amor e dedicação.

Ao meu orientador, Luiz Cláudio, pelo ótimo trabalho de orientação desempenhado.

Aos colegas de pesquisa do ECL e NIME, pela colaboração e compreensão. Em particular a Olav P. Henschel, pela colaboração técnica e a Gabriel Marcílio, que cedeu a implementação da ferramenta de E-matching.

À CAPES, no âmbito do Programa de Fomento à Pós-Graduação (PROF), pelo fomento à execução deste trabalho.

As atividades de pesquisa aqui descritas foram executadas no Laboratório de Computação Embarcada (ECL), Núcleo Interdepartamental de Microeletrônica (NIME), no âmbito do INCT NAMITEC, o qual custeou parcialmente este trabalho.

Insanity: doing the same thing over and over again and expecting different results.

Albert Einstein

RESUMO

O multiprocessamento em *chip* (CMP) mudou o panorama arquitetural dos servidores e computadores pessoais e agora está mudando o modo como os dispositivos pessoais móveis são projetados. CMP requer acesso a variáveis compartilhadas em hierarquias multiníveis sofisticadas onde *caches* privadas e compartilhadas coexistem. Ele se baseia no suporte em hardware para implicitamente gerenciar o relaxamento da ordem de programa e a atomicidade de escrita de modo a fornecer, na interface software-hardware, uma semântica de memória compartilhada bem definida, que é capturada pelos axiomas de um modelo de consistência de memória (MCM). Este trabalho aborda o problema de verificar se uma representação executável do subsistema de memória implementa um MCM especificado. Técnicas convencionais de verificação codificam os axiomas como arestas de um único grafo orientado, inferem arestas extras a partir de *traces* de memória e indicam um erro quando um ciclo é detectado. Usando uma abordagem diferente, esta dissertação propõe uma nova técnica que decompõe o problema de verificação em múltiplas instâncias de um problema (estendido) de emparelhamento de vértices em grafos bipartidos. Como a decomposição foi judiciosamente projetada para induzir instâncias independentes, o problema-alvo pode ser resolvido por um algoritmo paralelo de verificação. Também é proposto um gerador de sequências de instruções aleatórias distribuídas em múltiplas *threads* para estimular o sistema de memória sob verificação. Por ser independente do MCM sob verificação, o gerador proposto pode ser utilizado pela maioria dos verificadores. A técnica proposta, que é comprovadamente completa para diversos MCMs, superou um verificador convencional para um conjunto de 2400 casos de uso gerados aleatoriamente. Em média, o verificador proposto encontrou um maior percentual de faltas (90%) comparado ao convencional (69%) e foi, em média, 272 vezes mais rápido.

Palavras-chave: *Multicores*. Memória compartilhada. Consistência. Coerência. *Multithreading*. Grafos bipartidos.

ABSTRACT

Chip multiprocessing (CMP) changed the architectural landscape of servers and personal computers and is now changing the way personal mobile devices are designed. CMP requires access to shared variables in sophisticated multilevel hierarchies where private and shared caches coexist. It relies on hardware support to implicitly manage relaxed program order and write atomicity so as to provide, at the hardware-software interface, a well-defined shared-memory semantics, which is captured by the axioms of a memory consistency model (MCM). This dissertation addresses the problem of checking if an executable representation of the memory system complies with a specified consistency model. Conventional verification techniques encode the axioms as edges of a single directed graph, infer extra edges from memory traces, and indicate an error when a cycle is detected. Unlike them, this dissertation proposes a novel technique that decomposes the verification problem into multiple instances of an extended bipartite graph matching problem. Since the decomposition was judiciously designed to induce independent instances, the target problem can be solved by a parallel verification algorithm. To stimulate the memory system under verification, the dissertation also proposes a generator of multi-threading random-instruction sequences. It complies with an arbitrary MCM and can be used by most checkers. Our technique, which is proven to be complete for several MCMs, outperformed a conventional checker for a suite of 2400 randomly-generated use cases. On average, it found a higher percentage of faults (90%) as compared to that checker (69%) and did it, on average, 272 times faster.

Keywords: Multicores. Shared memory. Consistency. Coherency. Multithreading. Bipartite graph.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS

ARO	Alpha Relaxed Order
BG	Bipartite Graph
BVG	Bipartite Verification Graph
CG	Constraint Graph
CMP	Chip Multiprocessing
DAG	Directed Acyclic Graph
ESL	Electronic System Level
ILP	Instruction Level Parallelism
MCM	Memory Consistency Model
MPSoC	Multiprocessor System-on-Chip
PC	Processor Consistency
PSO	Partial Store Order
RC	Release Consistency
RIT	Random Instruction Test
RMO	SPARC Relaxed Memory Order
SC	Sequential Consistency
SMP	Symmetric Multiprocessing
SoC	System-on-Chip
TCC	Transaction Coherence and Consistency
TSO	Total Store Order
VMCM	Verify Memory Consistency Model
VSC	Verify Sequential Consistency
VTSO	Verify Total Store Order
WO	Weak Ordering

LISTA DE SÍMBOLOS

Parâmetros do programa concorrente e da arquitetura:

n	Número de instruções no programa concorrente
s	Número de variáveis compartilhadas no programa concorrente
p	Número de processadores da arquitetura
π_t	Fração de operações do tipo t no programa concorrente

Operações de memória:

Op_a^i	Uma operação de memória (L_a^i, S_a^i ou M)
L_a^i	Uma operação de leitura (*load*) em um endereço a por um processador i
S_a^i	Uma operação de escrita (*store*) em um endereço a por um processador i
X_a^i	Um operação atômica de leitura e escrita (*exchange* ou *swap*) em um endereço a por um processador i
M	Uma operação de barreira de memória

Atributos de operações:

$Val[L_a^i]$	O valor lido por L_a^i
$Val[S_a^i]$	O valor escrito por S_a^i

Conjuntos:

$\mathscr{O}$	O conjunto de operações de memória emitidas por todos os processadores
$\mathscr{O}^i$	O subconjunto de operações de memória emitidas pelo processador i
$\mathscr{S}$	O conjunto de operações de escrita emitidas por todos os processadores
$\mathscr{S}^i$	O subconjunto de operações de escrita emitidas pelo processador i

Relações:

$<_+$	Ordem local induzida pelo programa
$<_-$	Ordem local induzida pela execução
$<_t$	Ordem linear induzida por *timestamping*
$<_a$	Ordem total de escritas para o endereço a
$\leq$	Ordem global especificada pelo modelo de consistência
$\ll$	Ordem linear arbitrária
R_i	Projeção local da ordem global $\leq$ no i-ésimo processador

Funções:

$Max_{<_+}[\{S_a^i\}]$	Última operação de escrita para o endereço a observada em um dado processador i em ordem de programa
$Max_{\leq}[\{S_a^i\}]$	Última operação de escrita para o endereço a observada globalmente
$t(\tau_i)$	A marca de tempo (*timestamp*) de um evento τ_i
ρ	O mapeamento entre operações de leitura e escrita

Pontos observáveis e meios de observação:

i^+	Ponto de monitoramento de $<_+$ no i-ésimo processador
i^-	Ponto de monitoramento de $<_-$ no i-ésimo processador
m_i	Um monitor inserido no processador i
m_i^+	Uma instância do monitor m_i no ponto i^+
m_i^-	Uma instância do monitor m_i no ponto i^-

Resultados de observação:

τ	Um evento associado a uma operação de memória
T	Um *trace* de eventos de memória
T_i	Um *trace* observado no processador i
T_i^+	*Trace* observado pelo monitor m_i^+
T_i^-	*Trace* observado pelo monitor m_i^-
T^G	Um *trace* global

SUMÁRIO

1 INTRODUÇÃO

Na computação de propósitos gerais, fabricantes de computadores pessoais e servidores adotaram multiprocessamento em *chip* (CMP: *Chip Multiprocessing*) para continuar aumentando o desempenho ao se depararem com o limite prático de dissipação de potência denominado de *Power Wall* (HENNESSY; PATTERSON; ARPACI-DUSSEAU, 2007; PATTERSON; HENNESSY, 2009). Na computação móvel e embarcada, o multiprocessamento em *chip* foi a chave para viabilizar sistemas integrados multiprocessados (MPSoCs: *Multiprocessor System-on-Chip*) energeticamente eficientes, como os suportados pelas famílias ARM Cortex-A (ARM HOLDINGS, 2011) e MIPS 1074K (MIPS TECHNOLOGIES, 2011).

A comunicação entre *threads* usualmente é feita através de memória compartilhada. Dois acessos concorrentes à memória provenientes de diferentes *threads* referenciando o mesmo endereço levam a uma condição de corrida (*data race*) se pelo menos um deles escreve naquele endereço. Como *data races* podem induzir comportamentos distintos para um mesmo programa concorrente, eles devem ser eliminados através de um mecanismo de sincronização (por exemplo, um semáforo). Entretanto, a ordem dos acessos à memória influencia tanto a sincronização quanto o desempenho. Por um lado, se a ordem dos acessos à memória a diferentes endereços é relaxada, a sincronização pode não funcionar como esperado (ADVE; GHARACHORLOO, 1996). Por outro lado, se a ordem de programa é mantida, o desempenho é limitado (já que isto restringe o escalonamento estático em tempo de compilação e limita os benefícios do escalonamento dinâmico em microarquiteturas com execução fora-de-ordem). Portanto, o subsistema de memória deve fornecer suporte adequado em hardware ao relaxamento de ordem.

Para que os programas concorrentes possam explorar o relaxamento sem prejudicar a sincronização, é crucial definir a noção de semântica de memória compartilhada na interface hardware-software. Tal noção será ilustrada por meio de um exemplo (adaptado de (HILL, 1998)). A Figura 1 mostra dois segmentos de código a serem executados em processadores distintos (P1 e P2). Os processadores comunicam-se através de variáveis compartilhadas: `data` transmite os dados e `flag` é usado para sincronização. Uma vez que as atribuições `data = new` e `data_copy = data` são de diferentes *threads* e referenciam a mesma posição de memória, elas levam a uma condição de corrida (*data race*), a menos que estejam propriamente sincronizadas. Dependendo de como os eventos venham a ocorrer, dois comportamentos distintos poderiam ser induzidos: tanto `data_copy == new` quanto `data_copy == old`.

```
Valores iniciais: flag == UNSET; data == old;
  P1:             P2:
  data = new;
  flag = SET;
                  while (flag != SET) {}
                  data_copy = data;
```

Figura 1 – Ordem de acesso à memória e seu impacto na sincronização

Observe que o código assume que a variável `flag` é atribuída após `data`. Se esta ordem não for obedecida, a sincronização não funciona como esperado. Todavia, como entre `data` e `flag` não existe dependência (de controle ou de dados), elas poderão vir a ser reordenadas, tanto estaticamente por uma otimização de compilador ou dinamicamente pelo hardware de uma microarquitetura que suporte execução fora-de-ordem. Se tal reordenamento para o subsistema de memória é permitido, como ilustrado na Figura 2, isto pode induzir o comportamento anômalo `data_copy == old`.

```
Valores iniciais: flag == UNSET; data == old;
  P1:                 P2:
  flag = SET
  ----------------    ----------------
                      flag != SET
  ----------------    ----------------
                      data_copy = data
  ----------------    ----------------
  data = new
```

Figura 2 – Ordem global do ponto de vista do subsistema de memória

Portanto, para um comportamento adequado, é necessário garantir globalmente a ordem entre operações de acesso à memória compartilhada (`data` e `flag`), levando à noção de consistência de memória (*memory consistency*), que define **quando** as escritas de um processador são observadas por outro processador, i.e. **em que ordem** um processador deve ver as escritas de outro processador. Diversos modelos de consistência de memória (MCM: *Memory Consistency Model*) são relatados na literatura (ADVE; GHARACHORLOO, 1996), tais como *Sequential Consistency* (SC), *Processor Consistency* (PC), *Release Consistency* (RC), *Total Store Order* (TSO), *Partial Store Order* (PSO), *Alpha Relaxed Order* (ARO), *SPARC Relaxed Memory Order* (RMO), *Transaction Coherence and Consistency* (HAMMOND et al.,

2004), etc.

Um MCM é formalmente definido através de axiomas. Essencialmente, existem dois tipos de axiomas. Um **axioma de ordem** especifica até que ponto a ordem global vista por todos os processadores deve concordar com a ordem local especificada pelo programa. Um **axioma de valor** especifica o valor retornado por uma leitura para uma dada ordem de acessos de escrita. Como um MCM define parte da semântica da interface hardware-software, o problema de verificar se o hardware do subsistema de memória compartilhada implementa um MCM é crucial para a programação paralela.

No contexto da computação de propósitos gerais, diversas técnicas pós-silício foram propostas para resolver este problema (HANGAL et al., 2004; MANOVIT; HANGAL, 2005, 2006; CHEN et al., 2009). Elas se baseiam na observação de ordens locais e verificam se existe uma ordem global que satisfaz os axiomas de ordem e de valor de um dado MCM. Devido à observabilidade limitada do hardware, tal abordagem de verificação (*black-box*) é um tanto complexa (GIBBONS; KORACH, 1997).

Esta dissertação propõe uma nova técnica que opera no nível de sistema (ESL: *Electronic System Level*) (BAILEY; MARTIN; PIZIALI, 2007). Ela explora a observabilidade estendida de uma representação executável do subsistema de memória para reduzir o esforço computacional de se verificar, em fases inciais do fluxo de projeto, se aquele sistema realmente implementa um dado MCM. A ideia-chave é a decomposição em dois subproblemas: 1) a verificação do comportamento local observado na interface entre a memória privada de cada processador e a memória compartilhada; 2) a verificação do comportamento global resultante da interação entre todos os processadores através da memória compartilhada.

Para os MCMs que não requerem a ordenação total de escritas (como ARO, RMO, RC, WO, etc.), prova-se que a técnica proposta é **completa**[1] ao analisar o comportamento induzido por um dado caso de teste: não deixa de detectar erros existentes nem sinaliza erros aparentes (i.e. não apresenta falsos negativos ou falsos positivos). A técnica foi experimentalmente validada com 2400 casos de uso e foi comparada com um verificador convencional. Em média, a técnica proposta encontrou um maior percentual de falhas (90%) se comparada ao verificador convencional (69%) e foi, em média, 272 vezes mais rápida.

A técnica proposta nesta dissertação e sua avaliação experimental foram reportadas em trabalho a ser publicado nos anais do evento DATE 2012: *Design, Automation & Test in Europe Conference* (RAMBO; HENSCHEL; SANTOS, 2012).

A avaliação experimental do gerador de estímulos proposto está repor-

[1] A noção de completude de verificadores dinâmicos é discutida na página 44.

tado em trabalho publicado nos anais do evento ICECS 2011: *IEEE International Conference on Electronics, Circuits, and Systems* (RAMBO; HENSCHEL; SANTOS, 2011).

1.1 EXEMPLOS DE MODELOS DE CONSISTÊNCIA

As noções mais importantes sobre modelos de consistência de memória serão resumidas informalmente a seguir. Primeiramente, são revisados os principais requisitos do modelo SC e como otimizações comuns de hardware precisam ser proibidas ou seguramente restringidas de modo a cumprir tais requisitos. Em seguida, mostra-se como os requisitos dos modelos TSO e ARO habilitam algumas importantes otimizações.

1.1.1 Requisitos para SC e suas consequências

Uma arquitetura implementa o modelo *Sequential Consistency* (SC) se ela satisfaz, simultaneamente, os seguintes requisitos:

- **Preservação de ordem de programa por processador:** dada uma *thread* de um programa concorrente, a ordem de programa entre operações leitura (L) e escrita (S) para aquela *thread* deve ser preservada em quaisquer execuções do programa concorrente.
- **Unicidade da ordem global por execução:** uma única ordem entre todas as operações deve ser observada do ponto de vista de cada processador em uma dada execução do programa concorrente, embora uma ordem distinta possa ser observada para diferentes execuções do mesmo programa.

Em uma arquitetura sem caches, o segundo requisito seria respeitado por construção, já que todos os processadores referenciam uma variável compartilhada no mesmo componente de memória. Assim, não é possível observar ordens distintas em uma mesma execução. Nessas arquiteturas, a chave para suportar SC é preservar a ordem de programa. Todavia, o uso de atalho entre fila de escrita e fila de leitura sem passar pela interface com a memória deve ser vedado (para impedir violações $S \rightarrow L$ na ordem global) e o uso de rede genérica conectando múltiplos módulos de memória deve ser restringido (para evitar violações $S \rightarrow S$ e $L \rightarrow S$) (ADVE; GHARACHORLOO, 1996; MOSBERGER, 1993).

Por outro lado, em arquiteturas com caches, o uso combinado de caches não bloqueantes e escalonamento dinâmico devem ser restringidos (para

evitar violações $L \rightarrow L$) para respeitar o primeiro requisito. Ainda mais importante é o fato de que, já que processadores diferentes podem observar diferentes cópias de uma variável compartilhada, o segundo requisito só é satisfeito se uma escrita para uma variável compartilhada é vista por todos os processadores como uma operação indivisível (atômica). Todavia, a propagação das mudanças para múltiplas cópias da mesma variável residindo em caches diferentes (por meio de um protocolo de coerência) é essencialmente uma operação não-atômica. Como uma consequência, para cumprir o segundo requisito, é necessário emular escritas globais e atômicas. Mas, se não forem restringidas, a maioria das arquiteturas falhariam em preservar a atomicidade de escrita por causa de redes de interconexão que não garantem a ordem de entrega das mensagens em caminhos diferentes ou por causa de atraso em tornar coerentes as réplicas de uma variável compartilhada. Para superar estas dificuldades, deve haver um protocolo de comunicação (*handshaking*) entre o processador e o subsistema de memória para detectar a conclusão global de uma escrita antes que outro acesso à memória possa ser emitido.

1.1.2 Requisitos para TSO e suas consequências

A proibição de algumas otimizações, as restrições impostas a outras e o tempo gasto em *handshaking* para manter a atomicidade de escrita contribuem para a redução de desempenho. Por isso, o relaxamento de restrições, como no modelo *Total Store Order* (TSO), permite reduzir a degradação no desempenho.

Uma arquitetura implementa o modelo TSO se ela satisfaz, simultaneamente, os seguintes requisitos:

- **Preservação da ordem de programa por processador, exceto $S \rightarrow L$:** dada uma *thread* de um programa concorrente, a ordem de programa entre operações de leitura (L) e de escrita (S) para aquela *thread* deve ser preservada em quaisquer execuções do programa concorrente, exceto que operações de leitura (L) podem ser executadas antes de escritas (S) que a precedem na ordem de programa, desde que acessem endereços distintos.
- **Adiantamento de leituras para o mesmo endereço:** uma leitura (L) pode ler o valor de uma escrita (S) emitida anteriormente pelo mesmo processador para o mesmo endereço, antes que esta escrita esteja visível para os demais processadores.

O relaxamento da ordem de programa no modelo TSO pode melhorar substancialmente o desempenho do hardware ao reduzir o impacto de latên-

cia das operações de escrita (ADVE; GHARACHORLOO, 1996; MOSBERGER, 1993).

1.1.3 Requisitos para ARO e suas consequências

Uma arquitetura implementa o modelo *Alpha Relaxed Order* (ARO) se ela satisfaz, simultaneamente, os seguintes requisitos:

- **Relaxamento da ordem de programa por processador:** dada uma *thread* de um programa concorrente, a ordem de programa entre operações de leitura (L) e de escrita (S) para aquela *thread* não precisa ser mantida, desde que acessem endereços distintos. A ordem de programa entre operações (L/S) que acessam o mesmo endereço deve ser mantida.
- **Adiantamento de leituras para o mesmo endereço:** uma leitura (L) pode ler o valor de uma escrita (S) emitida anteriormente pelo mesmo processador para o mesmo endereço, antes que esta escrita esteja visível para os demais processadores.

O maior relaxamento deste modelo quando comparado aos anteriores tem o potencial de induzir mais oportunidades de otimização, que podem ser aproveitadas pelo compilador (escalonamento estático) e pelo hardware (escalonamento dinâmico, emissão múltipla), reduzindo o impacto da latência das operações de escrita e leitura (MOSBERGER, 1993). Entretanto, a complexidade da implementação aumenta e o modelo torna-se menos intuitivo para o programador (ADVE; GHARACHORLOO, 1996).

A Tabela 1 sintetiza os requisitos de SC, TSO e ARO, comparando seus graus de relaxamento.

Tabela 1 – Comparação entre modelos de consistência

Modelos	**Relaxamento**			
	Ordem de progama (endereços diferentes)			**Atomicidade de escrita (mesmo endereço)**
	$S \rightarrow L$	$S \rightarrow S$	$L \rightarrow L/S$	Read own write early
SC				✓
TSO	✓			✓
ARO	✓	✓	✓	✓

A Figura 3 ilustra como a ordem de operações de acesso à memória de um mesmo programa pode ou não ser relaxada dependendo do modelo de

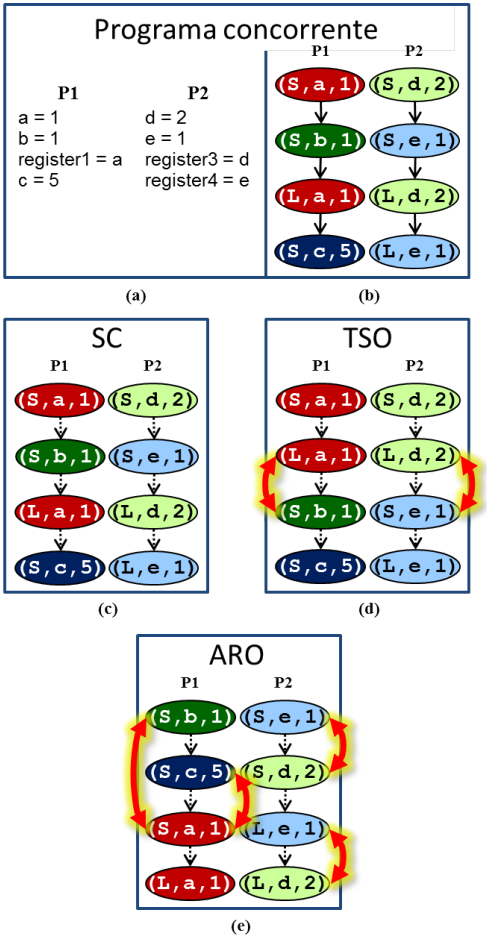

Figura 3 – Exemplo ilustrativo: relaxamentos de ordem em relação a SC

consistência adotado (SC, TSO ou ARO). O programa concorrente (Figura 3a) foi extraído e adaptado de um exemplo ilustrado em (ADVE; GHARACHORLOO, 1996). O programa consiste de duas *threads* que serão alocadas em processadores distintos (P1 e P2). *a*, *b*, *c*, *d* e *e* são variáveis compartilhadas alocadas em memória e register1, register3 e register4 representam variáveis locais alocadas em registradores.

A Figura 3b mostra a sequência de operações induzida pela *ordem de programa* em cada *thread* (a ordem é representada na vertical, de cima para baixo). Cada operação de acesso à memória é representada pela tripla (op, a, v), onde op identifica o tipo de operação (L: leitura ou S: escrita), a representa o endereço referenciado e v representa o valor lido ou escrito.

As Figuras 3c a 3e mostram ordens de execução que satisfazem os respectivos modelos de consistência. Inversões em relação à ordem de programa são nelas representadas por setas bidirecionais. No modelo SC não há relaxamento algum da ordem de programa (Figura 3c). O modelo TSO (Figura 3d) relaxa a ordem de operações que referenciam endereços distintos: as leituras podem ser realizadas antes das escritas. O modelo ARO (Figura 3e) é o mais relaxado dos três: as leituras e escritas para endereços distintos podem ser executadas em qualquer ordem.

Para manter a simplicidade do exemplo, a ordem global das operações não foi incluída na Figura 3. A ordem global é o resultado do entrelaçamento das operações de memória na ordem de execução dos processadores (P1 e P2). As Figuras 4a a 4c ilustram ordens globais válidas para as execuções das Figuras 3c a 3e, respectivamente.

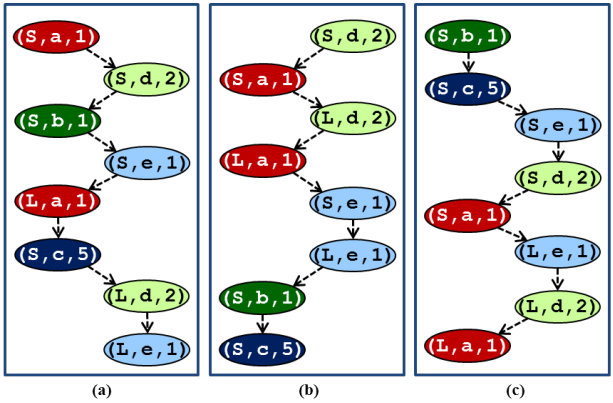

Figura 4 – Exemplo ilustrativo: ordens globais

As noções ilustradas no exemplo da Figura 3 serão formalizadas no Capítulo 2.

1.2 MODELO DE ARQUITETURA DE REFERÊNCIA

A chave para verificar a execução de um programa concorrente contra um MCM é observar a ordem dos eventos em alguns pontos do sistema. Ainda que a técnica proposta não esteja amarrada a uma arquitetura específica, a fim de identificar claramente os pontos de observação relevantes, faz-se necessário descrever uma família de arquiteturas na forma de um modelo genérico de plataforma, mostrado na Figura 5. Este é um modelo adequado para MPSoCs (LOGHI; PONCINO; BENINI, 2006). ARM Cortex A-15 e MIPS32 1074K são exemplos de arquiteturas que podem ser acomodadas nele (ARM HOLDINGS, 2011; MIPS TECHNOLOGIES, 2011).

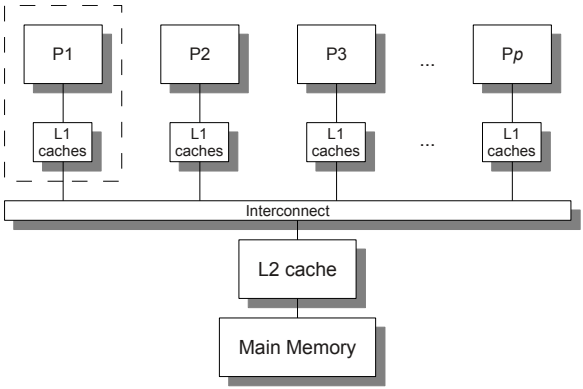

Figura 5 – Modelo genérico de uma arquitetura CMP

A microarquitetura de um elemento processante (delimitado pela linha tracejada na Figura 5) é mostrada em detalhes na Figura 6, adaptada de (HENNESSY; PATTERSON; ARPACI-DUSSEAU, 2007).

Na Figura 6, as instruções que realizam operações de memória são despachadas para uma fila de leitura (*load queue*) ou para uma fila de escrita (*store queue*), que não está mostrada explicitamente na figura, pois faz parte do buffer de reordenamento (*reorder buffer*). As instruções que não acessam a memória são despachadas para múltiplas unidades de execução (*execution units*). O buffer de reordenamento, interno à unidade de consolidação (*commit unit*), garante a escrita em registradores ou em memória na mesma ordem da fila de instruções.

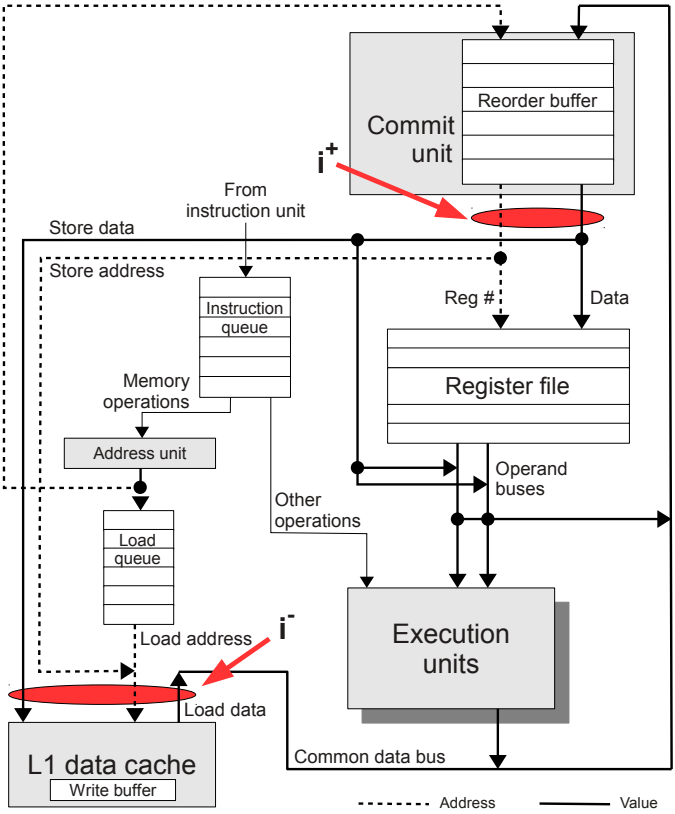

Figura 6 – Microarquitetura genérica de um elemento processante

Para suportar execução especulativa e exceções precisas (HENNESSY; PATTERSON; ARPACI-DUSSEAU, 2007), a maioria dos processadores consolidam as instruções na mesma ordem que elas foram emitidas (*in-order commit*). Por isso assume-se que, dada uma representação de uma arquitetura CMP, a ordem de programa pode ser observada ao se monitorar os eventos de memória na saída da unidade de consolidação de resultados (ponto i^+ na Figura 6). Quando são observados no ponto i^-, os eventos são monitorados na ordem em que foram efetivamente realizados na memória compartilhada. Em sistemas que fornecem um atalho para ler valores pendentes diretamente na fila de escritas (*store queue bypass*), a ocorrência de uma leitura antecipada pode ser observada monitorando-se as filas de leitura e escrita (*load and store queues*).

1.3 ESTRUTURA DA DISSERTAÇÃO

O restante desta dissertação está organizado como segue. O Capítulo 2 formaliza o problema-alvo de verificação. Em seguida, o Capítulo 3 revisa as abordagens convencionais do problema de verificação e, ao final, identifica detalhadamente a contribuição deste trabalho. O Capítulo 4 descreve a técnica de verificação proposta, através de uma decomposição do problema-alvo que é independente da arquitetura do subsistema de memória sob verificação. A validação e a avaliação experimental da técnica proposta são apresentadas no Capítulo 5. As conclusões e trabalhos futuros estão descritos no Capítulo 6.

Para melhor legibilidade, as provas formais das garantias de verificação apresentadas no Capítulo 4 foram isoladas no Apêndice A.

2 O PROBLEMA DE VERIFICAÇÃO

Este capítulo revisa as ordens de operações de memória utilizadas para especificar MCMs e localiza precisamente onde elas podem ser observadas na arquitetura genérica descrita na Seção 1.2. Em seguida, formulam-se instâncias relevantes do problema-alvo de verificação.

Vamos adotar a seguinte notação[1]. $\mathscr{O}$ é o conjunto de operações de memória emitidas por todos os processadores e $\mathscr{O}^i$ denota o subconjunto de operações emitidas pelo processador i. $\mathscr{S}$ é o conjunto de operações de escrita emitidas por todos os processadores e $\mathscr{S}^i$ denota o subconjunto de escritas emitidas pelo processador i. Uma operação de memória arbitrária (leitura ou escrita) é denotada por Op_j. Para especificar que a operação foi emitida pelo processador i, escreve-se $(Op_j)^i$. Para especificar ainda que tal operação referencia o endereço a, escreve-se $(Op_j)^i_a$.

Substitui-se Op por L ou S para especificar que a operação é uma leitura ou escrita, respectivamente. Por exemplo, $(L_j)^i_a$ e $(S_j)^i_b$ representam operações de leitura e escrita que referenciam os endereços a e b, respectivamente, e são emitidos pelo mesmo processador i. O subscrito j é descartado em notação abreviada, quando não é relevante para se distinguir uma operação de outra.

2.1 ORDENS EM MODELOS DE CONSISTÊNCIA

Primeiramente serão abordadas as ordens de eventos de memória que são observáveis por um dado processador.

Definição 2.1. Ordem local induzida pelo programa: Diz-se que $(Op_1)^i$ precede $(Op_2)^i$ em ordem de programa, o que é denotado por $(Op_1)^i <_+ (Op_2)^i$, se e somente se $(Op_1)^i$ atinge o início da fila de instruções do processador i antes de $(Op_2)^i$.

Como explicado anteriormente, a ordem de programa $<_+$ pode ser observada ao se monitorar os eventos de memória na saída da unidade de validação (*commit unit*) (ponto i^+ na Figura 6).

Apesar da ordem de busca na fila de instruções, um processador i pode colocar operações fora de ordem nas suas filas de escrita e leitura. Quando um processador emite uma escrita para uma *cache* privada, o seu efeito pode não ser visto imediatamente pelo sistema de memória compartilhada, já que a operação pode ter sido colocada em um *buffer* de escrita e, caso se refira

[1] Inspirada na notação proposta em (HANGAL et al., 2004).

a uma variável compartilhada, o seu efeito será visível somente depois de invalidar e/ou atualizar o bloco contendo uma cópia daquela variável em outras *caches* privadas. Além disso, leituras podem ser reordenadas pelo uso de *caches* privadas não bloqueantes. Como resultado, uma ordem de eventos distinta da ordem de programa pode ser observada no subsistema de memória compartilhada, mudando assim a ordem em que eles são observados na interface entre o processador e o subsistema de memória compartilhada (ponto i^- na Figura 6). Esta noção é formalizada abaixo.

Definição 2.2. Ordem local induzida pela execução: Diz-se que $(Op_1)^i$ precede $(Op_2)^i$ na ordem de execução, o que é denotado por $(Op_1)^i <_- (Op_2)^i$, se e somente se a conclusão de $(Op_1)^i$ é sinalizada para o controlador da *cache* de dados privada do processador i antes que a de $(Op_2)^i$.

Uma ordem global resulta da interação entre processadores distintos, através do sistema de memória compartilhada.

Definição 2.3. Ordem global: Diz-se que $(Op_1)^i$ precede $(Op_2)^k$ na ordem global, o que é denotado por $(Op_1)^i \leq (Op_2)^k$, se e somente se $(Op_1)^i$ completa a sua escrita ou leitura no subsistema de memória compartilhada antes que $(Op_2)^k$.

Observe que, ao contrário das ordens locais *observáveis*, tal ordem global é uma abstração. Em geral, não se pode apontar um lugar no sistema de memória onde ela pode ser monitorada[2], porque os eventos envolvidos estão distribuídos pelo sistema de memória compartilhada. Ela pode somente ser *inferida* da correlação entre as ordens locais observáveis. No entanto, é a ordem global que é *especificada* pelas regras de um dado MCM, como é mostrado na próxima seção.

2.2 OS AXIOMAS DE ORDEM

A maioria dos MCMs, tais como ARO, PSO, SC, TSO e WO (ADVE; GHARACHORLOO, 1996), tem uma restrição em comum: a serialização de escritas para o mesmo endereço, como formalizado abaixo.

Propriedade 2.1. Restrição de coerência de cache:

$$\forall S_a^i, S_a^k \in \mathscr{S} : (S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i).$$

MCMs são caracterizados pelos diferentes graus de relaxamento de ordem e requisitos de atomicidade de escrita (ARVIND; MAESSEN, 2006).

[2]Exceto se não existe cache no sistema e a memória compartilhada tem uma única porta.

Sem perda de generalidade, são ilustrados os requisitos de três MCMs[3]: *Sequential Consistency* (SC) (DUBOIS; THAKKAR, 1992), *Total Store Order* (TSO) (HANGAL et al., 2004) e *Alpha Relaxed Order* (ARO) (SITES; WITEK, 1995). Os axiomas de outros MCMs podem ser encontrados na literatura.

Essencialmente, existem dois requisitos de ordem para um MCM (ARVIND; MAESSEN, 2006). O primeiro requisito especifica o grau de relaxamento da ordem de programa. O modelo SC define que a ordem das operações executadas em um dado processador deve ser a ordem de programa. TSO difere de SC em permitir leituras serem executadas antes de escritas, desde que referenciem endereços distintos. Diferente dos anteriores, o modelo ARO permite o relaxamento da ordem de todas as operações, exceto operações que referenciam o mesmo endereço (para preservar as dependências de dados do programa). Este requisito está formalizado abaixo:

Axioma 2.1. Relaxamento da ordem de programa:

$$(Op_1)^i <_+ (Op_2)^i \Rightarrow Op_1 \leq Op_2 \textbf{ [SC]}$$

$$(L_a^i <_+ Op_b^i \Rightarrow L_a^i \leq Op_b^i) \vee (S_a^i <_+ S_b^i \Rightarrow S_a^i \leq S_b^i) \textbf{ [TSO]}$$

$$(Op_1)_a^i <_+ (Op_2)_a^i \Rightarrow (Op_1)_a^i \leq (Op_2)_a^i \textbf{ [ARO]}$$

O segundo requisito, formalizado em seguida, especifica a atomicidade das operações. O modelo SC especifica que todos os processadores devem observar a mesma ordem linear de operações. TSO especifica que deve haver uma ordem linear apenas para escritas. ARO requer que seja mantida somente a ordem das escritas para o mesmo endereço.

Axioma 2.2. Restrição de atomicidade:

$$\forall Op_a^i, Op_b^k \in \mathscr{O} : (Op_a^i \leq Op_b^k) \vee (Op_b^k \leq Op_a^i) \textbf{ [SC]}$$

$$\forall S_a^i, S_b^k \in \mathscr{S} : (S_a^i \leq S_b^k) \vee (S_b^k \leq S_a^i) \textbf{ [TSO]}$$

$$\forall S_a^i, S_a^k \in \mathscr{S} : (S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i) \textbf{ [ARO]}$$

2.3 O AXIOMA DE VALOR

A maioria dos MCMs (por exemplo, SC, PC, TSO, PSO, WO, RC, ARO e RMO) incorporam o efeito de uma propriedade interna de um processador no subsistema de memória:

[3] A descrição informal dos modelos SC, TSO e ARO consta da Seção 1.1.

Propriedade 2.2. Leitura antecipada de escrita local: A fila de escritas de cada processador i pode antecipar o valor de uma escrita pendente S_a^i para uma leitura L_a^i, antes que S_a^i alcance a memória compartilhada e se torne globalmente visível.

O impacto da Propriedade 2.2 nas leituras será avaliado em seguida.

Definição 2.4. Produtor local: Dada uma leitura L_a^i, o seu produtor local, o que é denotado por $Max_{<_+}[\{S_a^i | S_a^i <_+ L_a^i\}]$, é a operação $S_a^i \in \{S_a^i | S_a^i <_+ L_a^i\}$ tal que $\neg(\exists (S')_a^i | S_a^i <_+ (S')_a^i <_+ L_a^i)$.

Todos os MCMs têm um requisito comum que define o impacto da ordem global no valor a ser retornado por uma leitura, como formulado abaixo:

Definição 2.5. Produtor global: Dada uma leitura L_a^i, o seu produtor global, o que é denotado por $Max_{\leq}[\{S_a^k | S_a^k \leq L_a^i\}]$, é a operação $S_a^k \in \{S_a^k | S_a^k \leq L_a^i\}$ tal que $\neg(\exists S_a^j | S_a^k \leq S_a^j \leq L_a^i)$.[4]

Seja $Val[L_a^i]$ o valor retornado por uma leitura e $Val[S_a^i]$ o valor armazenado por uma escrita. Seja A o conjunto de todos os endereços referenciados pelas operações de memória.

Para cada endereço, um MCM requer que, entre os produtores potenciais para uma leitura, tanto na ordem global quanto na local, o valor retornado por uma leitura é o valor armazenado pela última escrita (HANGAL et al., 2004), como formalizado abaixo:

Axioma 2.3. Unicidade do valor retornado:

$$\forall a \in A : Val[L_a^i] = \begin{cases} Val[Max_{<_+}[\{S_a^i | S_a^i <_+ L_a^i\}]] & \text{se Propriedade 2.2} \\ Val[Max_{\leq}[\{S_a^k | S_a^k \leq L_a^i\}]] & \text{em caso contrário} \end{cases}$$

Além dos requisitos capturados pelos axiomas de ordem e de valor, axiomas suplementares são formulados para especificar operações de memória compostas (como o *swap* atômico) e para codificar o efeito de instruções que revogam o relaxamento padrão da ordem de programa (como as barreiras de memória).

Dado um *swap* atômico, digamos X_a^i, ele é modelado como um par de operações L_a^i e S_a^i tal que nenhuma outra operação possa ser intercalada entre elas (MANOVIT; HANGAL, 2006); formalmente:

$$X_a^i \Rightarrow (L_a^i \leq S_a^i) \wedge (\forall Op_a^j : Op_a^j \leq L_a^i \vee S_a^i \leq Op_a^j).$$

[4] O máximo pode ser definido somente para uma ordem linear. Entretanto, como a Propriedade 2.1 vale para a maioria dos MCMs de interesse prático, a ordem parcial $\leq$ (no conjunto $\mathscr{O}$) induz uma ordem linear no conjunto de escritas para um mesmo endereço.

Uma barreira de memória, M^i, ordena as operações de memória adjacentes no processador emissor i, isto é:

$$(Op_1)^i_a <_+ M^i <_+ (Op_2)^i_b \Rightarrow (Op_1)^i_a \leq (Op_2)^i_b.$$

Observe que os axiomas suplementares são, na verdade, axiomas de ordem. Portanto, assume-se (sem perda de generalidade) que, para propósitos de verificação, eles sejam integrados aos Axiomas 2.1 e 2.2.

Como observação final, cabe destacar que a seguinte condição necessária para consistência de memória, apesar de ser óbvia, é uma propriedade bastante útil para a detecção antecipada de falhas:

Propriedade 2.3. Visibilidade global: Exceto para as leituras que satisfazem a Propriedade 2.2, que nunca chegam à memória compartilhada, todas as outras operações no programa devem ser observadas na interface entre o processador que a emitiu e o sistema de memória (ponto i^- na Figura 6).

2.4 FORMULAÇÃO DO PROBLEMA-ALVO

Sejam L, S, X e M tipos distintos de operações: leitura, escrita, *swap* e barreira de memória, respectivamente.

Definição 2.6. *Trace*: um *trace* é uma sequência $T = (\tau_1, \tau_2, \cdots \tau_j, \cdots \tau_n)$, onde cada elemento $\tau_j = (op, a, v)$ é um evento de memória tal que $op \in \{L, S, X, M\}$, $a \in A$ e $v = Val[op_a]$ quando $op \neq M$. Como barreiras de memória não produzem nem consomem valores, define-se $Val[M] = NIL$.

Definição 2.7. *Trace* de processador: Um *trace* de processador $T_i = (\tau_1, \tau_2, \cdots \tau_j, \cdots \tau_n)$ é uma sequência onde cada evento de memória $\tau_j = (op, a, v)$ refere-se a uma operação de memória L^i_a ou S^i_a monitorada em algum ponto no processador i.

O problema de se verificar se um subsistema de memória obedece a um dado MCM pode ser formalizado como segue:

Problema 1. VMCM: Dada uma coleção de *traces* $T_1, T_2, \cdots T_p$, existe um *trace* global T que satisfaz todos os axiomas de ordem e de valor do MCM?

Vamos assumir que se possa determinar o mapeamento entre cada leitura consumindo um valor e a escrita que produziu aquele valor, como definido abaixo:

Definição 2.8. Mapeamento de leituras: Dada uma coleção de *traces* $T_1, T_2, \cdots T_p$, seja $L = \{L^i_a | (a \in A) \wedge (1 \leq i \leq p)\}$ e seja $S = \{S^j_a | (a \in A) \wedge (1 \leq j \leq$

$p)\}$. Um mapeamento de leituras é a função $\rho : L \rightarrow S$ tal que $Val[L_a^i] = Val[\rho(L_a^i)]$.

Uma instância restrita de um VMCM pode ser definida quando o mapeamento de leituras é conhecido antecipadamente (MANOVIT; HANGAL, 2005), como segue:

Problema 2. VMCM-*read*: Dado um mapeamento ρ, existe um *trace* T como especificado no Problema 1?

Vamos assumir agora que a ordem de escritas para o mesmo endereço é previamente conhecida para cada endereço a referenciado.

Definição 2.9. Ordem total de escritas para um endereço: Dada uma coleção de *traces* $T_1, T_2, \cdots T_p$ e o endereço a, a ordem total de escritas para a é a relação $<_a$ tal que $\forall S_a^i, S_a^k \in S : (S_a^i <_a S_a^k) \vee (S_a^k <_a S_a^i)$, onde $S = \{S_a^j | (a \in A) \wedge (1 \leq j \leq p)\}$.

Se a ordem acima é conhecida, pode-se definir uma instância ainda mais restrita do problema-alvo (MANOVIT; HANGAL, 2005), como segue:

Problema 3. VMCM-*conflict*: Dada uma ordem total de escritas $<_a$ para cada endereço a, existe um trace T como especificado no Problema 2?

Os Problemas 1 e 2 são NP-completos (GIBBONS; KORACH, 1992), enquanto o Problema 3 pode ser resolvido em tempo polinomial (HANGAL et al., 2004; MANOVIT; HANGAL, 2005, 2006). Dado um MCM, um verificador é considerado completo, ao resolver uma dada instância de VMCM, se e somente se é capaz de provar que todo trace induzível pelo sistema de memória obedece ao MCM e é capaz de provar, para todo trace que não obedeça ao MCM, a sua não-conformidade com aquele modelo. Assim, a verificação completa é intratável para os Problemas 1 e 2, o que motiva o uso de verificadores dinâmicos.

Como a qualidade da verificação dinâmica (simulação em tempo de projeto ou teste após a prototipagem) depende da qualidade dos casos de teste utilizados, verificadores práticos baseiam-se frequentemente em testes gerados aleatoriamente para aumentar as chances de se expor erros (cobertura). Portanto, no contexto de verificação dinâmica, a **completude** costuma ser definida (HANGAL et al., 2004; MANOVIT; HANGAL, 2005, 2006; CHEN et al., 2009) sob a perspectiva de um caso de teste: um verificador dinâmico é completo se, e somente se, para o comportamento induzido por um caso de teste, todo erro exposto é detectado e nenhum erro aparente é sinalizado. Ou seja, é preciso usar uma **condição necessária e suficiente** para se detectar um erro.

O próximo capítulo discute verificadores dinâmicos de consistência usando a eficiência esperada (complexidade dos algoritmos subjacentes) e as garantias de verificação (grau de completude) como critérios de comparação.

3 TRABALHOS CORRELATOS EM VERIFICAÇÃO DINÂMICA

Este capítulo traz uma revisão bibliográfica dos principais trabalhos correlatos em verificação dinâmica de MCMs. Aborda-se primeiramente a verificação em tempo de projeto e, depois, o teste de consistência após a prototipagem. Em seguida, discute-se uma técnica desenvolvida para outro contexto de verificação, mas cujo uso para verificação de consistência é promissor. Ao final do capítulo, discute-se a contribuição científica do trabalho aqui proposto frente aos trabalhos correlatos.

3.1 VERIFICAÇÃO DE CONSISTÊNCIA EM TEMPO DE PROJETO

A ideia-base para detectar inconsistências, originalmente atribuída a Landin, Hagersten e Haridi (1991) e utilizada em várias técnicas convencionais de verificação (HANGAL et al., 2004; MANOVIT; HANGAL, 2005, 2006; ROY et al., 2006; CHEN et al., 2009), consiste em se anotar as relações de ordem parcial entre operações de memória em um grafo orientado. Como somente um grafo orientado *acíclico* (DAG: *directed acyclic graph*) pode representar uma ordem parcial, a detecção de um ciclo no grafo é uma prova de inconsistência da memória (já que a anti-simetria não é preservada).

Este tipo de grafo pode ser assim formalizado:

Definição 3.1. Grafo de restrições: Um grafo de restrições $CG(V,E)$ é um grafo orientado onde cada vértice $v \in V$ representa uma operação e cada aresta $(u,v) \in E$ representa uma restrição de ordem especificando que aquela operação u deve ocorrer antes da operação v.

No entanto, um grafo de restrição acíclico não necessariamente prova a consistência da memória, pois algumas relações de ordem existentes entre operações podem não ter sido inferidas (MANOVIT; HANGAL, 2006) e não foram codificadas no grafo. Desse modo, um verificador baseado em um grafo incompleto como esse pode sinalizar falsos positivos, pois verifica uma condição necessária mas não suficiente.

Muitas abordagens existentes utilizam os chamados testes de instruções aleatórias (RITs: *Random Instruction Tests*) (HANGAL et al., 2004; MANOVIT; HANGAL, 2005, 2006; ROY et al., 2006; CHEN et al., 2009). RITs são programas concorrentes e sintéticos de teste que possuem uma *thread* para cada processador, a qual executa instruções de acesso à memória (escritas, leituras, etc.) em uma faixa de endereços compartilhados, induzindo *data races*. A sequência de instruções e o conjunto de referências são gerados

pseudo-aleatoriamente para se tentar obter cobertura adequada.

TSOTool é uma ferramenta que gera RITs e, após sua execução, analisa os valores retornados pelas operações de leitura, sinalizando sucesso ou falha (HANGAL et al., 2004). Essa técnica força a geração de RITs para que valores únicos sejam escritos em endereços distintos. Como consequência, o mapeamento de leituras ρ (Definição 2.8) é conhecido por construção. Portanto, a ferramenta não resolve o Problema 1, mas resolve o Problema 2 (VTSO-*read*). Ao ser alimentada com a ordem de escritas $<_a$ para cada endereço (Definição 2.9), ela resolve o Problema 3 (VTSO-*conflict*). Entretanto, pode ser aplicada tanto para verificar em tempo de projeto quanto para teste após a prototipagem, onde a observabilidade do hardware é limitada. A técnica baseia-se em inferências sobre a ordem de operações, construídas a partir dos axiomas. A aplicação sucessiva de inferências resulta na inserção de arestas adicionais no grafo de restrições até que seja detectado um ciclo ou não se possa mais fazer inferências. A análise tem complexidade $O(n^5)$, no pior caso, onde n é o número de instruções submetidas à análise.

Uma extensão de TSOTool reduziu o tempo de execução e a complexidade temporal utilizando relógios vetoriais e melhores heurísticas (MANOVIT; HANGAL, 2005). Relógios vetoriais são utilizados para gerar uma ordenação parcial de eventos em sistemas distribuídos. O estado do sistema é armazenado em um vetor de relógios lógicos e cada processo tem uma cópia desse vetor, que é atualizado quando ocorre troca de mensagens. Estas melhorias resultaram na complexidade temporal $O(pn^3)$, no pior caso, onde p é o número de processadores.

Posteriormente, o algoritmo anterior foi aprimorado para tornar a análise completa (MANOVIT; HANGAL, 2006). Para tanto, foram adicionadas heurísticas combinadas com *backtracking*. Após o término da execução do algoritmo-base (MANOVIT; HANGAL, 2005), aplica-se a seguinte heurística: para cada escrita, na ordem topológica, regras são aplicadas iterativamente ao grafo de restrições até não haver mais aresta a ser adicionada. Quando nenhuma outra instrução pode ser selecionada na ordem topológica sem violar o MCM, o algoritmo retorna a um ponto de decisão anterior e tenta outra ordem de escolha. O algoritmo com derivação e *backtracking* (completo) leva cerca de 2,6 vezes mais tempo que o algoritmo-base (incompleto) para executar. A garantia de completude aumenta a complexidade para $O((n/p)^p pn^3)$, no pior caso.

Inspirada em TSOTool, uma técnica foi proposta para verificar processadores da Intel® (ROY et al., 2006). Ela é mais genérica, pois os processadores-alvo implementam vários modelos de consistência e admitem diferentes comportamentos de escrita em memória (*write through*, *write back*, *write protect*, *write combining* e *uncacheable*). A técnica também utiliza grafos de

restrições para capturar a ordem e adota o algoritmo de Warshall incremental (CORMEN et al., 2009), juntamente com uma matriz de adjacências, aumentando assim o desempenho em uma ordem de magnitude. O algoritmo final foi paralelizado para aproveitar o poder computacional disponível (pois ele é executado no próprio protótipo sendo testado) e amortizar suas complexidades temporal e espacial ($O(n^4)$ e $\Theta(n^2)$, respectivamente, no pior caso).

Na ferramenta LCHECK (CHEN et al., 2009), a verificação de consistência foi analisada com base em ordenação temporal. A ordenação temporal é induzida pelo fato de se poder associar instantes de tempo ao início e ao fim de cada instrução. Os intervalos de duração das instruções são usados para permitir checagem local (ao invés de global), reduzindo a complexidade do algoritmo. Isto é possível por causa das restrições temporais adicionais. LCHECK pode ser aplicado na verificação em tempo de projeto e no teste após a prototipagem; porém, neste último cenário, requer registradores adicionais em cada processador que sejam visíveis na interface hardware-software. Uma versão do algoritmo faz verificação sem garantia de completude com complexidade $O(p^3n)$, no pior caso. A versão que garante completude tem complexidade $O(C^p p^2 n^2)$, no pior caso, onde C é uma constante.

Alguns trabalhos sobre CMP baseiam-se em relógios de Lamport, que são ferramentas bem conhecidas para trabalhar com ordenação global de eventos em sistemas distribuídos (LAMPORT, 1978). A ideia principal é associar um contador a cada processo e aplicar uma marca de tempo (*timestamp*) a cada mensagem trocada entre processos. Quando um evento ocorre em um processo, o seu contador é incrementado e o valor resultante é propagado como uma marca de tempo. Quando um processo recebe uma mensagem, o seu contador é atualizado com a marca de tempo da mensagem.

O trabalho de Sorin et al. (1998) realiza a verificação de consistência sequencial baseada na ordenação de eventos de memória com relógios de Lamport. Com os eventos de memória em ordenação total é possível tentar verificar a correção do sistema. Ele realiza, com o uso de relógios de Lamport, uma marcação de épocas de coerência para um bloco de memória no sistema de memória compartilhada, as quais definem o estado de compartilhamento de um bloco em um determinado instante. Essas épocas garantem uma ordem global de operações em um bloco e são usadas para verificar a correção do sistema com relação a um modelo de consistência. Esse trabalho foi estendido para trabalhar com modelos de memória mais relaxados (TSO e ARO) (CONDON et al., 1999). Ambos os trabalhos provam que a técnica funciona, mas não fornecem resultados experimentais que possam ser comparados ou utilizados para analisar a sua eficiência.

Uma abordagem recente verifica a consistência utilizando um *scoreboard* relaxado (SHACHAM et al., 2008). Ao invés de registrar apenas um

valor esperado de saída para cada estímulo de entrada, como ocorre em um *scoreboard* convencional, múltiplos valores esperados são mantidos enquanto um único valor não puder ser deterministicamente identificado. Esse tipo de verificador é fracamente acoplado ao projeto específico, o que desacopla a implementação da construção do verificador, permitindo construir o *scoreboard* incrementalmente (o que permite seu uso nas fases iniciais do projeto). Os acessos à memória são monitorados e, toda vez que ocorre um acesso, a entrada do *scoreboard* correspondente ao endereço de memória acessado é atualizada, eliminando os valores impossíveis, caso seja uma leitura, e inserindo novos valores possíveis, no caso de uma escrita. Essas atualizações são feitas de acordo com regras que definem o comportamento esperado, ou seja, conforme os axiomas de um modelo de consistência. Esse é um método, que executa em paralelo com a simulação e para a verificação assim que um erro é encontrado.

3.2 TESTE DE CONSISTÊNCIA EM PROTÓTIPO

Além de técnicas baseadas em grafos de restrições (HANGAL et al., 2004; MANOVIT; HANGAL, 2005, 2006; ROY et al., 2006; CHEN et al., 2009), que também podem ser aplicados após a prototipagem, há abordagens de verificação dinâmica de consistência com suporte em hardware (MEIXNER; SORIN, 2005, 2009; CANTIN; LIPASTI; SMITH, 2001). A verificação é realizada com o auxílio de hardware adicional embutido no processador o qual acompanha a execução do sistema, atualizando a sua estrutura de dados interna e verificando se ocorreu alguma violação na última atualização.

Algumas técnicas permitem não somente a detecção como também a recuperação em face de alguns tipos de falhas menos graves, causando uma interferência no sistema para restaurá-lo a um estado anterior válido (MEIXNER; SORIN, 2005, 2009).

Embora seja muito interessante para aumentar a disponibilidade e segurança do hardware, os trabalhos foram direcionados para servidores, não levando em consideração o *overhead* em MPSoCs, onde o poder computacional é limitado e a eficiência energética precisa ser maximizada.

3.3 TÉCNICA NÃO CONVENCIONAL PROMISSORA

O emparelhamento estendido de grafos bipartidos (E-matching: *extended bipartite graph matching*) (MARCILIO et al., 2009; MARCíLIO, 2008) aborda a verificação funcional de IPs (*intellectual property blocks*) em proje-

tos ESL. Ele se baseia em um grafo bipartido de verificação (BVG: *bipartite verification graph*) cujas partições representam eventos monitorados em pontos equivalentes de representações executáveis distintas (modelo de referência e dispositivo sob verificação) e cujas arestas representam a compatibilidade dos valores observados. Como consequência, a inexistência de um emparelhamento (*matching*), indica uma falha no dispositivo sob verificação. Entretanto, ao contrário do emparelhamento convencional, os vértices em um E-matching também devem satisfazer a uma restrição de ordem especificada por uma relação *R*.

Apesar de ser originalmente projetado para verificação funcional, vislumbra-se o uso de E-matching como um mecanismo promissor para a verificação de consistência de memória, como explicado a seguir.

Em contraste com as técnicas convencionais, E-matching não depende de inferências (que nem sempre têm sucesso (MANOVIT; HANGAL, 2006)). Ao invés de analisar individualmente se duas operações de memória formam um par ordenado, o E-matching analisa as consequências da execução fora-de-ordem em todos os pares de valores. Verificadores convencionais empregam um único grafo para codificar todas as restrições, enquanto o E-matching pode ser aplicado independentemente a múltiplos grafos bipartidos.

Diferentemente das abordagens convencionais, que requerem RITs capazes de forçar as escritas a produzirem valores únicos, E-matching pode ser usado quando o subsistema de memória é estimulado por programas reais, nos quais a produção de valores não está condicionada às necessidades do verificador. Embora não requeira valores únicos para funcionar, a técnica de E-matching beneficia-se do uso de RITs no que se refere à melhoria de eficiência, pois os grafos bipartidos gerados a partir de RIT são esparsos por construção, reduzindo o tempo para encontrar um emparelhamento. Um outro aspecto importante deve ser notado: ao contrário dos verificadores convencionais, cuja eficiência aumenta com o maior número de restrições codificadas no grafo, a técnica de E-matching não se torna menos eficiente ao verificar MCMs altamente relaxados.

O aspecto promissor desta técnica não convencional de verificação motivou-nos a utilizá-la como base para um algoritmo paralelo de verificação de consistência, conforme discutido na próxima seção.

3.4 CONTRIBUIÇÃO CIENTÍFICA

No âmbito de um fluxo de projeto ESL (BAILEY; MARTIN; PIZIALI, 2007) de MPSoCs, dispõe-se comumente de uma representação executável da plataforma (descrita, por exemplo, em SystemC (BLACK; DONO-

VAN, 2005)) para a qual existem técnicas que estendem sua observabilidade (por exemplo, a técnica *white-box* em (ALBERTINI et al., 2007)). A maior observabilidade das representações executáveis permite descartar algoritmos computacionalmente caros (com *backtracking*) e estender as garantias de verificação enquanto utiliza-se um algoritmo eficiente com complexidade polinomial. A extensão da observabilidade, no entanto, não necessariamente torna uma técnica dependente da microarquitetura, tal como ocorre em algumas técnicas reportadas na literatura (LUDDEN et al., 2002).

Estes fatos motivaram a decomposição do Problema 1 de forma que dele resultem p instâncias *independentes*, moldadas como problemas de E-matching. Como resultado dessa moldagem deliberada, é possível reutilizar o algoritmo de E-matching, dele herdando algumas garantias de verificação. Como consequência da decomposição judiciosa, induz-se um algoritmo paralelo de verificação, cuja complexidade do caso médio é bem inferior à do pior caso.

Para uma dada execução de um programa, a técnica proposta garante a completude da verificação, ou seja:

- **Não-detecção de erros aparentes**: o verificador proposto não sinaliza falsos positivos em um sistema que, na verdade, é correto;
- **Detecção efetiva de erros reais**: o verificador proposto não sinaliza falsos negativos em um sistema que, na verdade, contém erros.

A Tabela 2 compara qualitativamente a técnica proposta com os trabalhos correlatos sob os seguintes aspectos: complexidade, completude, estrutura básica e modelos suportados. Além disso, ela indica se uma técnica é aplicada durante (*on-the-fly*) ou após a simulação (*post-mortem*).

O próximo capítulo descreve a técnica de verificação proposta e avalia a sua complexidade computacional.

Tabela 2 – A técnica proposta comparada às correlatas

Trabalho	**Complexidade**	Completude	Estrutura Básica	Modelos	Análise
HANGAL et al., 2004	$O(n^5)$		DAG	TSO	*Post-mortem*
MANOVIT; HANGAL, 2005	$O(pn^3)$		DAG	TSO	*Post-mortem*
MANOVIT; HANGAL, 2006	$O((n/p)^p pn^3)$	✓	DAG	TSO	*Post-mortem*
ROY et al., 2008	$O(n^4)$		DAG	TSO e outros	*Post-mortem*
CHEN et al., 2009	$O(p^3n)$		DAG	WC	*Post-mortem*
	$O(C^p p^2 n^2)$	✓	DAG	WC	*Post-mortem*
SHACHAM et al., 2008	$O(p^2n^2)$		*Scoreboard*	TSO, TCC	*On-the-fly*
Este trabalho	$O(n^6/p^5)$	✓	Grafo bipartido	Arbitrário[a]	*Post-mortem*

[a] Qualquer modelo que não requeira ordenação total de escritas e que possua coerência de cache.

4 A TÉCNICA DE VERIFICAÇÃO PROPOSTA

Ao invés de reutilizar, para propósitos de verificação, em tempo de projeto, os mecanismos convencionais desenvolvidos no contexto de testes pós-implementação de memória compartilhada, a técnica proposta é construída sobre a observabilidade estendida, que é disponibilizada pelas representações executáveis usadas em fluxos de projetos em ESL. Ela requer o monitoramento de dois pontos por processador (i^+ e i^- na Figura 6), selecionados para serem genéricos o suficiente para manter a abordagem independente da escolha de microarquitetura.

Esta abordagem baseia-se nos dois requisitos principais da maioria dos MCMs, isto é, a ordem de programa (relaxada) e a atomicidade de escrita (ADVE; GHARACHORLOO, 1996), para induzir uma decomposição do problema-alvo de verificação, dividindo-o em dois passos sucessivos:

- Passo 1: A verificação do comportamento **local** observado na interface de cada processador com a sua memória **privada**.
- Passo 2: A verificação do comportamento **global** resultante da interação entre todos os processadores através da memória **compartilhada**.

O primeiro passo assume que *traces* podem ser monitorados na saída da unidade de consolidação de resultados (*commit unit*) (i^+ na Figura 6), onde as operações de memória podem ser observadas na ordem de programa. Ele também assume que *traces* podem ser monitorados na interface entre as filas de escrita/leitura e a memória de dados privada do processador (i^- na Figura 6), onde as operações de memória podem estar fora da ordem original do programa. Esse passo consiste em verificar se a ordem $<_-$ satisfaz a *projeção local* da ordem global especificada $\leq$ (Axiomas 2.1 e 2.2). Essa projeção representa um relaxamento da ordem de programa $<_+$. Esse passo também verifica se o valor consumido por uma leitura corresponde ao valor produzido localmente por uma escrita pendente (Axioma 2.3 aplicado aos produtores locais de escritas pendentes).

O segundo passo não assume observabilidade extra, mas supõe que um *trace* global pode ser construído como um entrelaçamento dos *traces* de processador monitorados localmente. Ele consiste em verificar se a ordem global $\leq$ mantém atomicidade de escrita (ADVE; GHARACHORLOO, 1996), isto é, se a mesma ordem $\leq$ é observada por todos os processadores de modo que induza o consumo consistente dos valores produzidos (Axioma 2.3 aplicado aos produtores globais).

Antes de descrever formalmente os algoritmos, a técnica será apresentada informalmente através de um exemplo ilustrativo. A Figura 7 mostra

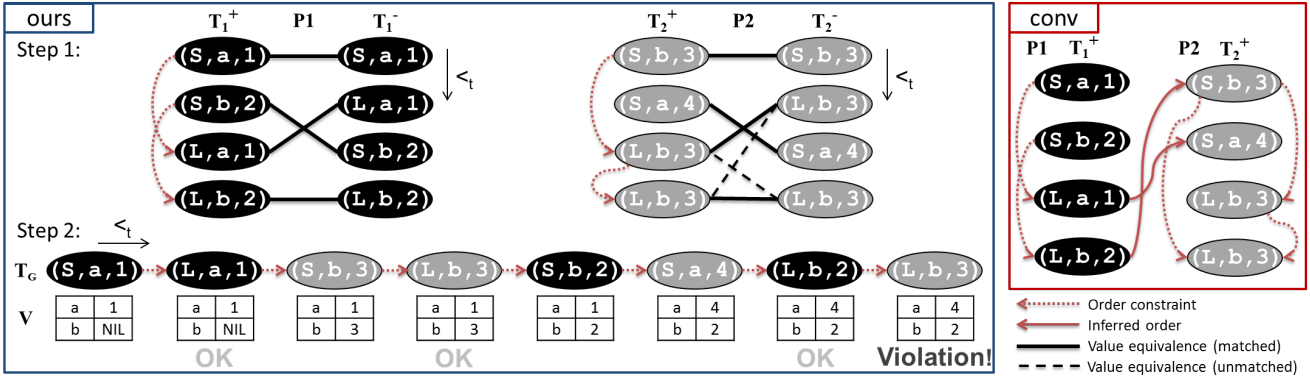

Figura 7 – Exemplo ilustrativo: o verificador proposto comparado ao convencional

uma comparação conceitual entre a técnica proposta (do lado esquerdo) e um verificador convencional baseado em detecção de ciclo (do lado direito). Como o verificador convencional requer um caso de teste no qual valores distintos são escritos em endereços diferentes (de modo a construir o mapeamento ρ), a mesma suposição foi feita para fins de comparação, apesar de este não ser um requisito para a técnica proposta.

Suponha que se queira verificar se um sistema com dois processadores (P1 e P2) obedecem ao modelo ARO, analisando os seus *traces*. Assuma que os *traces* capturam as sequências de execução que foram afetadas por uma falha no mecanismo de coerência (um valor escrito em uma cache local não foi propagado para outras caches). Na figura, as setas rotuladas como $<_t$ indicam a ordem em que os eventos foram monitorados (*timestamping*). Para cada processador, a técnica constrói um grafo bipartido de verificação (BVG), cujos vértices representam eventos monitorados e cujas arestas representam eventos equivalentes. As restrições de reordenamento especificadas pelos Axiomas 2.1 e 2.2 estão *esquematicamente* indicadas pelas setas pontilhadas (que *não* são arestas do BVG, mas elementos de uma relação de ordem externa R). Observe que cruzamentos de arestas em um BVG indicam que as operações observadas na interface com o sistema de memória (i^-) estão fora da ordem de programa (i^+).

No Passo 1, cada BVG é submetido ao algoritmo de E-matching, que tenta encontrar um emparelhamento completo (isto é, um mapeamento de um-a-um) que não induz cruzamentos de arestas se as operações envolvidas estão em R (isto é, se elas estão ordenadas pela especificação do MCM). Se tal emparelhamento *próprio* não é encontrado, ou a relação de ordem não é satisfeita ou operações estão faltando na interface com a memória. Note que, para P1, existe um único emparelhamento (e que ele é próprio). Para P2, no entanto, existem dois emparelhamentos possíveis. Observe que a inversão de ordem representada pelas arestas tracejadas viola a restrição de ordem do ARO e induziria um emparelhamento impróprio. Por isso o algoritmo E-matching descarta aquelas arestas. As arestas restantes levam a um emparelhamento próprio. Como resultado, nenhum erro foi encontrado da perspectiva do comportamento intra-processador (o que era de se esperar, pois a falha de coerência somente será exposta ao analisar o comportamento inter-processador).

O Passo 2 verifica o Axioma 2.3. Primeiro, um *trace* global (T_G) é construído de acordo com a ordem das marcas de tempo ($<_t$). À medida que percorre o *trace* global na ordem crescente das marcas de tempo, o estado atual de cada endereço de memória é mantido em uma tabela (V), que é atualizada a cada evento de escrita e verificada a cada evento de leitura. Note que, para as primeiras três leituras, o valor consumido pela leitura corresponde ao

valor armazenado pela última escrita ao mesmo endereço. Entretanto, uma incompatibilidade ocorre para a última leitura, expondo um comportamento inter-processador errado, que foi induzido pela falha de coerência.

A partir do conjunto inicial das restrições da ordem de programa, o verificador convencional tenta inferir novas relações de ordem por meio da transitividade. Um dos mecanismos que habilitam a inferência é o consumo de valores. Observe que a última leitura emitida por P2 deveria consumir o valor produzido pela última escrita emitida por P1. No entanto, como a falha de coerência impediu a propagação de valores entre processadores, o mecanismo de inferência falha em detectar a relação entre eles. Para superar esta limitação, os verificadores convencionais podem depender de *backtracking* (MANOVIT; HANGAL, 2006). Entretanto, mesmo ao preço de maior esforço computacional, o verificador convencional é inapto a encontrar a falha. Tal verificador acaba encontrando uma ordem que valida a execução: por exemplo, ao ordenar a última leitura emitida por P1 para um dado endereço e a primeira escrita para o mesmo endereço emitida por P2 (como indicado pelas setas sólidas). Como o grafo resultante é um DAG, o verificador convencional assume como correta uma execução que é, na verdade, incorreta.

A simplicidade deliberada do exemplo não deve levar a subestimar uma vantagem importante de se adotar E-matching (ao invés de um algoritmo guloso mais simples): a sua habilidade de trabalhar com múltiplos eventos equivalentes, ao explorar a combinação de restrição de ordem e equivalência de valores, para reduzir o espaço de pesquisa. Por um lado, quando RITs forçam a unicidade de valores armazenados por diferentes escritas, cada vértice representando uma escrita tem grau um (vértices representando leituras podem ter grau maior, já que o mesmo valor pode ser lido mais de uma vez). Por outro lado, quando programas concorrentes reais são rodados (para os quais a unicidade dos valores escritos é um tanto improvável), o número de eventos equivalentes e o grau dos vértices crescem substancialmente. Além disso, a partição à direita de um BVG pode ter mais vértices que a partição à esquerda. Por exemplo, quando leituras executadas especulativamente são monitoradas (T_i^-), elas não são necessariamente consolidadas (*commited*) (T_i^+). Portanto, leituras especulativas tendem a expôr mais valores redundantes, aumentando as alternativas de emparelhamento. Em tais cenários, os BVGs mais densos beneficiam-se da eliminação (*pruning*) de arestas impróprias que caracteriza a técnica de E-matching (MARCILIO et al., 2009).

4.1 PASSO 1: VERIFICANDO O COMPORTAMENTO LOCAL

O Passo 1 molda o subproblema de checar os comportamentos locais em um problema E-matching (MARCILIO et al., 2009). Para checar comportamentos locais na interface de cada processador, um par de monitores, m_i^+ e m_i^-, são inseridos em cada processador i nos pontos i^+ e i^-, respectivamente, de modo a observar os *traces* T_i^+ e T_i^-. A partir dos *traces*, um *grafo bipartido de verificação* BVG_i é construído de modo que cada vértice representa um evento da forma (op, a, v). Portanto, se um vértice u^+ representa um evento (op^+, a^+, v^+) e um vértice u^- representa um evento (op^-, a^-, v^-), existe uma aresta (u^+, u^-), se e somente se $op^+ = op^-$, $a^+ = a^-$ e $v^+ = v^-$.

Seja R^i a restrição de ordem genérica especificada no problema de E-matching (MARCILIO et al., 2009). No contexto de verificação de consistência, ela se torna a *projeção* no processador i da ordem global especificada $\leq$, isto é:

$$R^i = \{((Op_1)^j, (Op_2)^k) : (Op_1)^j \leq (Op_2)^k \wedge (j = k = i)\}.$$

Tal modelagem pode ser interpretada como segue. Ao se executar um programa concorrente, deve-se observar exatamente os mesmos valores da perspectiva dos pontos i^+ e i^- (*equivalência de valor*) e deve existir um mapeamento de um-pra-um entre os eventos de memória observados em i^+ e i^- (*completude* e *unicidade*). Em outras palavras, os eventos de memória vistos do ponto de vista de i^+ e i^- devem ser os mesmos, apesar de as suas sequências poderem exibir ordens diferentes. A execução fora de ordem com respeito a i^+ é aceitável desde que a ordem observada no ponto i^- não viole a restrição de ordem R^i especificada pelo MCM (*causalidade*). Se um E-matching não pode ser encontrado, isto significa que existe um evento de memória faltando em i^-, ou uma leitura em i^- retornou um valor errado, ou os valores foram retornados em uma ordem que viola os axiomas de ordem de um MCM.

Dado um processador i, o Algoritmo 1 preliminarmente verifica se o Axioma 2.3 vale para produtores locais (linha 2). Em caso positivo, as leituras consumindo valores de escritas locais pendentes (*consumidores locais*) são excluídas de V^+ (linha 4), já que elas não chegam até o sistema de memória. Isto garante que cada evento em V^+ tem pelo menos um evento correspondente em V^-. Então o Algoritmo 1 modela a verificação do comportamento local na forma de um problema de E-matching (linhas 4–9), como explicado acima. Na linha 10, emprega-se para verificação de consistência de memória o algoritmo proposto por (MARCILIO et al., 2009).

O algoritmo de E-matching tem complexidade temporal $O(|E|^3)$ (MAR-

Algoritmo 1: `local-behavior-OK(MCM, $T_i^+$, $T_i^-$)`

```
1  let ≤ be the order specified by the MCM;
2  if ¬ read-own-write-ok(T_i^+) then
3      return false;
4  V^+ = T_i^+ − local-consumers(T_i^+);
5  V^− = T_i^−;
6  V = V^+ ∪ V^−;
7  E = {(v^+, v^−) ∈ V^+ × V^− | v^+ = v^−};
8  O^i = {op | (op, a, v) ∈ V^+};
9  R^i = {(op_1, op_2) ∈ O^i × O^i | op_1 ≤ op_2};
10 return proper-matching(R^i, V, E);
```

CILIO et al., 2009), onde $|E|$ é o número de arestas no BVG. A utilização de um gerador de RITs que força uma distribuição uniforme de operações entre processadores resulta em $|E| = O(n^2/p^2)$. Portanto, o E-matching leva tempo $O(n^6/p^6)$ no pior caso para verificar o comportamento local de um único processador. No entanto, a complexidade média tende a ser muito menor na prática (o Capítulo 5 contém suporte experimental para esta afirmação).

As garantias de verificação do Algoritmo 1 são formalizadas a seguir.

Lema 4.1. O Algoritmo 1 retorna verdadeiro se e somente se os Axiomas 2.1, 2.2 e 2.3, bem como a Propriedade 2.3, são válidos para $\mathscr{O}^i$.

Prova: Veja Apêndice A. ◇

4.2 PASSO 2: VERIFICANDO O COMPORTAMENTO GLOBAL

A ideia fundamental deste passo é construir um *trace* global que combine todos os traces de processador para então verificar se a produção e consumo global de valores obedece o Axioma 2.3.

A chave para se construir o *trace* global é utilizar o tempo em que os eventos de memória foram observados como um critério para ordenar as operações em uma sequência global. Assume-se que uma marca de tempo, digamos $t(\tau_j)$, é atribuída a cada evento τ_j quando ele é observado pelo monitor m_i^-. Como resultado, quando a simulação termina, todos os eventos gravados em T_i^- têm uma marca de tempo. Esta noção pode ser formalizada como segue.

Definição 4.1. Seja $\mathscr{T} = \{\tau_j \,|\, \tau_j \in T_i^- \wedge (1 \leq i \leq p)\}$ o conjunto de todos os eventos monitorados por m_i^-. Seja c_j o tempo da simulação quando τ_j foi observado. A função ***timestamp*** é um mapeamento $t : \mathscr{T} \rightarrow \aleph$ tal que $t(\tau_j) = c_j$.

Como cada operação de memória, digamos x, é representada por um evento τ_j, $t(\tau_j)$ pode ser usado como uma marca de tempo por x, o que é denotado por $t(x)$. Para se abordar formalmente as garantias de verificação do Passo 2, a consequência deste *timestamping* será formulada a seguir.

Definição 4.2. Ordem linear induzida por *timestamping*: Seja $\ll$ uma ordem linear arbitrária no conjunto $\mathscr{O}$. A ordem $<_t$ é definida como segue:

$$\forall x, y \in \mathscr{O} : x <_t y \Leftrightarrow (t(x) < t(y)) \vee (t(x) = t(y) \wedge x \ll y).$$

O Algoritmo 2 usa a ordem $<_t$ para construir um *trace* global. Ele assume que a correspondência entre cada evento de memória e o processador onde o evento ocorreu é previamente conhecida (linha 2). Ele se baseia em uma fila de prioridades Q que mantém os eventos na ordem $<_t$. Inicialmente, Q é inicializada com o primeiro evento observado em cada processador (linhas 4–6). Então cada evento é movido, na ordem crescente de *timestamps*, do *trace* do respectivo processador para o *trace* global (linhas 7–12), como segue. O evento com o menor *timestamp* (linha 8) é concatenado ao *trace* global (linha 9) e removido do respectivo *trace* de processador (linha 10). Enquanto um *trace* de processador não resultar em uma sequência vazia (linha 11), o seu próximo evento é inserido em Q (linha 12). Observe que (ao invés de inserir todos os n eventos na fila para obter a ordenação desejada) o algoritmo faz com que no máximo p eventos possam estar simultaneamente na fila. Como resultado, o Algoritmo 2 tem complexidade temporal $O(n \log p)$ quando Q é implementada como uma *heap* (ao invés de $O(n \log n)$).

Algoritmo 2: `build-global-trace(`T_1^-`, `T_2^-`, ⋯, `T_p^-`)`

```
1  let T_G be an empty sequence of events;
2  let Π : {τ} → {1, 2, ⋯, p} such that Π(τ) = i ⇒ τ ∈ T_i^-;
3  let Q be a min-priority queue;
4  for i = 1 to p do
5      τ = extract-first(T_i^-);
6      insert(Q, τ, t(τ));
7  while Q ≠ ∅ do
8      τ = extract-min(Q);
9      concatenate(T_G, τ);
10     τ = extract-first(T_Π(τ)^-);
11     if τ ≠ NIL then
12         insert(Q, τ, t(τ));
13 return T_G;
```

O Algoritmo 3 verifica se a produção e consumo de valores induzida pelo *trace* global é consistente com os axiomas de um MCM. Ele mantém uma tabela de *hash* (linhas 1–2) que guarda os valores observados para aquele

trace. Depois de construir um *trace* global (linha 3), ele inicializa a tabela (linhas 4–6) antes de visitar os eventos de memória na ordem induzida pelo *trace* global (linhas 7–13). Quando uma escrita é visitada (linha 9) o seu valor é guardado na tabela de *hash* (linha 10). Quando uma leitura é visitada (linha 11), o algoritmo verifica se o valor consumido corresponde ou não ao último valor produzido por uma escrita para o mesmo endereço (Axioma 2.3). Ele retorna falso tão logo uma incompatibilidade de valores é detectada. Quando retorna verdadeiro, a consistência entre a produção e o consumo de valores está garantida.

Algoritmo 3: `global-behavior-ok(`T_1^- , T_2^- , $\cdots$, T_p^-`)`

```
 1 let V be a hash table;
 2 let h : A → {1,2,···,|A|} be a hash function;
 3 T_G = (τ_1, τ_2, ···τ_j, ···τ_n) = build-global-trace(T_1^-, T_2^-, ···, T_p^-);
 4 A = {a | τ_j = (op, a, v) ∧ 1 ≤ j ≤ n};
 5 for i = 1 to |A| do
 6     V[i] = NIL;
 7 for j = 1 to n do
 8     let τ_j be (op, a, v);
 9     if op = S then
10         V[h(a)] = v;
11     if op = L ∧ v ≠ V[h(a)] then
12         return false;
13 return true;
```

Como o Algoritmo 3 realiza $O(n)$ acessos à tabela de *hash*, assumindo-se que cada acesso leva $O(1)$, ele é dominado pela construção do *trace* global, isto é, tem complexidade temporal $O(n \log p)$ no pior caso.

As garantias de verificação do Algoritmo 3 são formalizadas a seguir.

O lema abaixo garante que verificar a serialização de escritas na coerência de cache equivale a se verificar o axioma de valor. [1]

Lema 4.2. O Axioma 2.3 vale para os produtores globais se e somente se a Propriedade 2.1 for válida.

Prova: Veja Apêndice A. ◇

O próximo lema garante que a ordem observável na representação executável de um sistema com coerência de cache pode ser usada como substituta à ordem especificada pelo MCM.

Lema 4.3. Para qualquer sistema de memória com coerência de cache, a ordem $\leq$ especificada por um MCM genérico é indistinguível da ordem de *timestamps* $<_t$.

[1] Este lema será usado para provar o Teorema 4.1

Prova: Veja Apêndice A. ◇

Finalmente, um terceiro lema usa o lema anterior para provar a condição necessária e suficiente para a verificação do axioma de valor.

Lema 4.4. Para qualquer sistema de memória com coerência de cache, o Algoritmo 3 retorna verdadeiro se e somente se o Axioma 2.3 é satisfeito para produtores globais.

Prova: Veja Apêndice A. ◇

4.3 COMBINANDO COMPLEXIDADES E GARANTIAS TEÓRICAS

O Algoritmo 4 simplesmente invoca o Algoritmo 1 para cada processador, antes de invocar o Algoritmo 3.

Algoritmo 4: `behavior-ok(MCM,` $T_1^+, \cdots, T_p^+,\ T_1^-, \cdots, T_p^-$`)`

```
1 for i = 1 to p do
2     if ¬local-behavior-ok(MCM, T_i^+, T_i^-) then
3         return false;
4 return global-behavior-ok(T_1^-, T_2^-, ···, T_p^-);
```

Como o algoritmo de E-matching é invocado p vezes, o Passo 1 leva $O(n^6/p^5)$ no pior caso. Como o Algoritmo 3 é invocado somente uma vez, o Passo 2 leva $O(n \log p)$. Como resultado, o esforço total de verificação é $O(n^6/p^5)$ no pior caso. Entretanto, como as p invocações do algoritmo de E-matching são totalmente independentes, quando o laço sequencial (linhas 1–3) é transformado em um laço totalmente paralelizado, a complexidade geral é reduzida para $O(n^6/p^6)$. O Capítulo 5 fornece evidências de que a complexidade de caso médio é na verdade bem menor que a de pior caso.

As garantias de verificação da técnica proposta estão formalizadas no teorema abaixo.

Teorema 4.1. Para qualquer sistema de memória com coerência de cache e para qualquer MCM que não requeira ordenação total de escritas, o Algoritmo 4 retorna verdadeiro se e somente se todos os axiomas do MCM são válidos para o comportamento observado, induzido por um dado caso de teste.

Prova: Veja Apêndice A. ◇

O Teorema 4.1 garante que o Algoritmo 4 é completo para MCMs tais como ARO, PSO e WO (ADVE; GHARACHORLOO, 1996), isto é, ele verifica todos os axiomas de um dado MCM e, para um dado caso de teste, nunca

induz falsos positivos nem falsos negativos. Mesmo ao verificar MCMs que requerem a ordenação total de escritas (como SC e TSO), o Algoritmo 4 fornece fortes garantias. Como ele garante a ordenação total de escritas para cada processador (Lema 4.1) e a ordenação total de escritas entre processadores distintos para o mesmo endereço (Lemas 4.2 e 4.4), falta-lhe somente a garantia de ordenação de escritas emitidas por diferentes processadores para endereços distintos.

4.4 O GERADOR DE CASOS DE TESTE PROPOSTO

Esta seção apresenta um método de geração de testes com instruções pseudo-aleatórias (RITs), que pode ser usado em combinação com a técnica de verificação proposta. Embora seu uso não seja mandatório (a técnica funcionaria com qualquer programa concorrente), ele é recomendável por permitir melhor controle da cobertura de verificação. O algoritmo proposto para a geração de casos de teste pode ser usado pela maioria dos verificadores dinâmicos (HANGAL et al., 2004; MANOVIT; HANGAL, 2005, 2006; CHEN et al., 2009; ROY et al., 2006; SHACHAM et al., 2008), os quais pressupõem algoritmos de geração similares.

O gerador fornece controle da composição (*mix*) de instruções, como segue. Sejam L, S, X e M os seguintes tipos de operações, respectivamente: leitura, escrita, *swap* e barreira de memória. Seja π_t a probabilidade de ocorrência de uma instrução de tipo t e seja $\pi = (\pi_L, \pi_S, \pi_X, \pi_M)$ a composição desejada de instruções, com $\pi_L + \pi_S + \pi_X + \pi_M = 1$. Para manter o compromisso entre cobertura e tempo de execução, o gerador aceita três parâmetros: p é o número de processadores, n é o número total de instruções geradas e s é o número de endereços compartilhados.

Primeiramente, será abordada a geração das *threads* e das instruções e depois serão explicados os detalhes de gerenciamento de *threads*. Para sintetizar um RIT, o Algoritmo 5 gera exatamente uma *thread* por processador, isto é, p *threads* (linhas 5–22). Cada *thread* consiste em uma sequência de n/p instruções contendo π_t operações de memória de cada tipo t, de modo que as *threads* compartilhem exatamente os mesmos s endereços (linhas 11–20).

Um número gerado aleatoriamente é usado como chave para selecionar o tipo de operação t da instrução a ser sintetizada de modo que a probabilidade especificada π_t seja obedecida (linha 11). As instruções são sintetizadas de acordo com a necessidade de referenciar variáveis compartilhadas (linha 12) ou a necessidade de armazenar um valor na memória (linha 14). As instruções são efetivamente geradas de acordo com o seu tipo e suas necessidades de armazenamento ou consumo de valores (linhas 16, 18, 20).

Um valor é atribuído a uma instrução de escrita ou *swap* (linhas 15-16). Cada valor é escolhido para ser único (linha 6), como em (HANGAL et al., 2004). Em outras palavras, o mapeamento entre leituras e escritas requerido por um verificador convencional para tratar o problema de verificação como um problema VMCM-*read* é induzido por construção. As instruções fazem referências aleatórias (linha 13) a s variáveis compartilhadas distintas com probabilidade $1/s$. Apesar de as variáveis poderem ser interpretadas como endereços, suas referências somente se tornarão endereços de fato depois de serem submetidas ao ligador.

Algoritmo 5: `generate_RIT(`p`, `n`, `s`, `π`)`

```
 1 let R be a sequence of s shared references;
 2 let v be a data value;
 3 generate_includes();
 4 R ← (a_1, ..., a_k, ..., a_s);
 5 for i ← 1 to p do
 6     v ← (n/p) × (i − 1);
 7     start_method();
 8     generate_sync_barrier();
 9     generate_flag();
10     for j ← 1 to n/p do
11         t ← random_select(π);
12         if t ∈ {L, S, X} then
13             k ← random(1,s);
14             if t ∈ {S, X} then
15                 v ← v + 1;
16                 generate_SX(a_k, v);
17             else
18                 generate_L(a_k);
19         else
20             generate_M();
21     generate_flag();
22     end_method();
23 generate_main(p, R);
```

Agora será explicada a geração de um gerenciamento de *threads* adequado. O gerador sintetiza diretivas para incluir rotinas de bibliotecas de *threads* (linha 3). Para ter certeza que todas as *threads* geradas terão a sua execução iniciada simultaneamente em cada processador, elas passam por uma barreira de sincronização (linha 8). Quando o programa sintetizado é compilado, código auxiliar[2] será injetado antes e depois do código RIT gerado. Para evitar interferência entre o código auxiliar e o código do caso de teste (estaticamente condicionado), o algoritmo marca os seus pontos de en-

[2] Inicialização de pilha, empilhamento e desempilhamento da convenção de chamadas, junção de *threads*

trada e saída. Cada marca (entrada ou saída) consiste de uma escrita *dummy*, cercada de barreiras de memória, que faz a função de um sinalizador (*flag*). Os sinalizadores são colocados no início do código de cada *thread* (linha 9) e no final (linha 21). Um sinalizador pode ser implementado através da escrita de um valor reservado; sendo único, o valor reservado é usado para distinguir as escritas dos sinalizadores das escritas do RIT. As linhas 7 e 22 geram a declaração e abertura do método de uma *thread* e o seu fechamento, respectivamente. Finalmente, o algoritmo sintetiza a rotina principal (linha 23), que é responsável por criar as *threads* e inicializar as variáveis compartilhadas.

O gerador proposto foi utilizado nos experimentos descritos no próximo capítulo.

5 VALIDAÇÃO E AVALIAÇÃO DA TÉCNICA PROPOSTA

Este capítulo apresenta a validação experimental da técnica proposta para complementar as garantias teóricas apresentadas no Capítulo 4.

Depois de descrever a infraestrutura usada para experimentação e reportar, para fins de reprodutibilidade, as condições exatas em que os experimentos foram realizados, a técnica proposta é comparada com um verificador convencional para 240 casos de teste e 10 tipos diferentes de falhas, perfazendo 2400 casos de uso.

5.1 INFRAESTRUTURA EXPERIMENTAL

A técnica proposta foi validada e avaliada com o auxílio do *framework* de simulação dirigida a eventos denominado de GEM5 (anteriormente chamado de M5) (BINKERT et al., 2006; GEM5, 2011), que serve de representação executável de uma arquitetura CMP. O *framework* possui diversos modelos de processadores, incluindo o modelo funcional atemporal, o modelo funcional temporizado (*cycle approximate*) e o modelo detalhado com precisão de ciclos (*cycle accurate*) com execução fora-de-ordem. GEM5 suporta seis arquiteturas (Alpha, SPARC, MIPS, ARM, x86 e Power) e tem dois modos de operação: o modo de emulação de chamadas de sistema (*syscall emulation*), onde a carga de trabalho (*workload*) é executada sem sistema operacional na plataforma, e o modo de sistema completo (*full system*), onde a carga de trabalho é executada com um sistema operacional completo (por exemplo, um kernel Linux).

Pôde-se também contar com infraestrutura desenvolvida em trabalho anterior: o artefato de verificação baseado em emparelhamento estendido em grafos bipartidos desenvolvido por (MARCILIO et al., 2009; MARCíLIO, 2008), que foi incorporado na forma de uma biblioteca e que é invocada pela ferramenta de verificação aqui proposta.

O protótipo da ferramenta de verificação aqui proposta foi implementado na linguagem C++. Inicialmente, para representar os grafos, tentou-se utilizar as estruturas de dados pertencentes à biblioteca de grafos *Boost Graph Library* (BGL) (SIEK; LEE; LUMSDAINE, 2002), mas o consumo de memória tornou-se impraticável com o crescimento do número de arestas. Finalmente, optou-se por uma implementação própria, finamente ajustada, baseada em lista de adjacências. O uso de matriz de adjacências foi inicialmente considerado para representar grafos de restrições, mas posteriormente descartado, ao se observar uma piora considerável no tempo de execução da

ferramenta.

Para fins de comparação, implementou-se (também em C++) um verificador convencional (baseado em grafos de restrições) que é em tudo similar ao descrito em (HANGAL et al., 2004), exceto por ter sido adaptado para verificar o modelo ARO ao invés de TSO.

5.2 CONFIGURAÇÃO EXPERIMENTAL

A arquitetura da plataforma adotada para a validação experimental foi espelhada no estado da arte das arquiteturas SMP para MPSoCs (ARM HOLDINGS, 2011), em particular a arquitetura da família Cortex A. As instâncias de plataforma utilizadas nos experimentos adotaram as mesmas características de hierarquia de memória, resumidas na Tabela 3, mas diferentes números de elementos processantes ($p \in \{2, 4, 8\}$).

Tabela 3 – Configuração experimental da hierarquia de memória

Tipo	**Caches**			**Memória principal**
Nível	1	1	2	3
Conteúdo	dados	instruções	unificada	unificada
Capacidade	32KB	32KB	4MB	128MB
Associatividade	4-way	1-way	8-way	–
Uso	privada	privada	compartilhada	compartilhada
Coerência	*snooping*	*snooping*	–	–

Os experimentos restringem-se à verificação do modelo *Alpha Relaxed Order* (ARO). Apesar de o modelo TSO ser mais utilizado nos trabalhos correlatos, a infraestrutura disponível não está preparada para este modelo. A modificação do modelo de consistência nativo do *framework* GEM5 é tarefa trabalhosa e propensa a erros. Por isso, preferiu-se utilizar o modelo nativo (ARO) por pragmatismo e confiabilidade. O *framework* original foi utilizado por vários trabalhos de pesquisa anteriores, tais como (JAHRE; GRANNAES; NATVIG, 2009; ZHENG et al., 2009; SRIDHARAN; KAELI, 2010), de modo que é improvável que alguma falha de implementação possa prejudicar a avaliação dos resultados de verificação.

Para a geração de estímulos foi utilizado o algoritmo proposto na Seção 4.4, o qual aceita quatro parâmetros: p é o número de processadores, n é o número total de operações de memória, s é o número de endereços compartilhados e $\pi = (\pi_L, \pi_S, \pi_X, \pi_M)$ é a composição (*mix*) das instruções de memória, onde π_t é a probabilidade de ocorrência de uma operação de me-

Tabela 4 – Composição de instruções dos casos de teste

Mix	**Loads** (π_L)	**Stores** (π_S)	**Swaps** (π_X)	**Membars** (π_M)
π_1	33%	33%	30%	4%
π_2	16%	50%	30%	4%
π_3	50%	16%	30%	4%
π_4	60%	30%	7%	3%

Tabela 5 – Parâmetros para geração dos casos de teste

# Endereços (s)	**# Operações** (n)	**# Processadores** (p)
2	2K	2
4	4K	4
8	8K	8
16	16K[2]	–
32	–	–

mória de tipo $t \in \{L, S, X, M\}$. Foram gerados 240 RITs distintos variando os parâmetros de acordo com as Tabelas 4[1] e 5.

Os experimentos foram realizados em uma estação de trabalho baseada em um processador Intel Xeon com 4 núcleos, operando à frequência de 2,66 GHz, com 4 GB de memória principal (SDRAM, DDR-2, 667 MHz).

5.3 RESULTADOS EXPERIMENTAIS

Em todos os experimentos, para cada simulação da execução de um dado teste em uma dada plataforma, foram monitorados e extraídos os *traces* de processador a serem submetidos a análise. A inclusão dos monitores na representação executável da plataforma sob verificação não apresentou impacto significativo no tempo de simulação.

Os *traces* de processador foram então submetidos a ambos os verificadores: o convencional e o proposto (denotados por *conv* e *ours* nesta seção).

Primeiramente, foram executados 240 casos de teste em uma plataforma sem falhas. Os *traces* gerados foram submetidos ao verificador proposto. Como esperado do Teorema 4.1, nenhum erro foi detectado, o que é

[1] π_1, π_2 e π_3 foram extraídos de (SHACHAM et al., 2008), e π_4 foi escolhido arbitrariamente.

[2] Valores superiores a este causam exaustão de memória para o verificador convencional tornando inviável sua comparação com a técnica proposta.

uma evidência experimental que o verificador proposto não induz falsos positivos.

Em seguida, foram modelados dez diferentes tipos de falhas[3]. A partir de uma instância isenta de erros, foram derivadas dez plataformas defeituosas, cada uma com uma falha diferente. Note que, se todas as falhas fossem injetadas na mesma plataforma, a verificação pararia tão logo uma falha fosse encontrada, deixando as demais ocultas. A separação das falhas em plataformas distintas foi uma estratégia pragmática para ampliar a avaliação da técnica. Cada RIT foi executado em cada plataforma, resultando em 2400 casos de uso.

Para fins de reprodutibilidade, a Tabela 6 caracteriza formalmente o comportamento de cada falha e indica em que componente da arquitetura ela foi inserida. A tabela assume que $i \neq j$. Para melhor legibilidade, as falhas são informalmente descritas a seguir.

A falha 1 modela uma anomalia no mecanismo de comparação de endereços da fila de escritas que leva à não-detecção de uma escrita pendente, desse modo forçando uma leitura a consumir um valor obsoleto da *cache* de dados. A falha 2 se encontra no controle do *buffer* de reordenamento (ROB); ela modela uma anomalia que impede as escritas de serem validadas na ordem de programa. A falha 3 se localiza no caminho de adiantamento de escritas entre as filas de escrita e de leitura (*store queue bypass*), isto é, no caminho que permite uma leitura emitida consumir diretamente o valor de uma escrita local pendente; ela modela uma anomalia do tipo *bit-stuck-at-one* que induz o consumo de um valor errado. A falha 4 modela uma anomalia similar, mas está localizada no caminho entre a cache de dados e uma leitura emitida. A falha 5 induz a violação da restrição de reordenamento imposta por uma barreira de memória. A falha 6 modela uma anomalia no mecanismo de invalidação de modo que o valor inicial de uma variável compartilhada é retornado, ao invés do valor efetivo produzido por uma escrita de outro processador. A falha 7 se localiza na interface entre o controlador do protocolo de coerência e o barramento compartilhado; modela uma anomalia do tipo *bit-stuck-at-one* nos blocos recebidos de outro processador por ação do protocolo de coerência. A falha 8 se localiza na interface entre o controlador do protocolo de coerência e o controlador da *cache*: ao receber uma mensagem de invalidação, a lógica de controle do protocolo falha em desativar o bit de validade, levando ao consumo de um valor obsoleto. A falha 9 modela uma anomalia no controlador da *cache* quando este realiza uma operação de *swap*; o valor correto é escrito na memória mas o valor retornado pelo *swap* é afe-

[3] A maioria foi inspirada nas falhas utilizadas em (SHACHAM et al., 2008) mas não necessariamente reflete as mesmas condições daquele experimento pois aquele trabalho não as descreve com suficiente precisão para serem fielmente reproduzidas.

Tabela 6 – Caracterização formal das falhas injetadas

f	**Descrição comportamental**	**Localização estrutural**
1	$(Max_{\leq}[\{S_a^k \mid S_a^k \leq L_a^i\}] = (S_2)_a^i) \wedge ((S_1)_a^j \leq (S_2)_a^i) \wedge (Val[L_a^i] = Val[(S_1)_a^j])$	*Store queue bypass*
2	$((S_1)_a^i <_+ (S_2)_a^i) \wedge ((S_2)_a^i \leq (S_1)_a^i)$	*Reorder buffer*
3	$(Max_{\leq}[\{S_a^k \mid S_a^k \leq L_a^i\}] = S_a^i) \wedge (Val[S_a^i] \neq Val[L_a^i])$	*Store queue bypass*
4	$(Max_{\leq}[\{S_a^k \mid S_a^k \leq L_a^i\}] = S_a^i) \wedge (Val[S_a^i] \neq Val[L_a^i])$	*Data cache interface*
5	$((Op_1)^i <_+ M <_+ (Op_2)^i) \wedge ((Op_2)^i \leq (Op_1)^i)$	*Execution Unit Control*
6	$(Max_{\leq}[\{S_a^k \mid S_a^k \leq L_a^i\}] = S_a^j) \wedge (Val[L_a^i] = 0)$	*Data cache bus interface*
7	$(Max_{\leq}[\{S_a^k \mid S_a^k \leq L_a^i\}] = S_a^j) \wedge (Val[S_a^j] \neq Val[L_a^i])$	*Data cache bus interface*
8	$(Max_{\leq}[\{S_a^k \mid S_a^k \leq L_a^i\}] = (S_2)_a^j) \wedge ((S_1)_a^i \leq (S_2)_a^j) \wedge (Val[L_a^i] = Val[(S_1)_a^i])$	*Cache block validity bit*
9	$(X_a^i = ((L_1)_a^i, (S_1)_a^i)) \wedge (Max_{\leq}[\{S_a^k \mid S_a^k \leq (L_1)_a^i\}] = (S_2)_a^j) \wedge (Val[(S_2)_a^j] \neq Val[(L_1)_a^i])$	*Data cache interface*
10	$(Max_{\leq}[\{S_a^k \mid S_a^k \leq L_a^i\}] = (S_2)_a^j) \wedge ((S_1)_a^i \leq (S_2)_a^j) \wedge (Val[L_a^i] = Val[(S_1)_a^i])$	*Cache block dirty bit*

Tabela 7 – Caracterização informal das falhas injetadas

f	**Descrição da falha**
1	Fila de escritas não verificada por leituras
2	Reordenamento indevido de escritas para o mesmo endereço
3	Adiantamento de valores incorretos à partir da fila de escritas
4	Valores incorretos retornados pela cache
5	Violação da restrição do tipo barreira de memória
6	Violação total da integridade de um bloco requisitado via coerência
7	Violação parcial da integridade de um bloco requisitado via coerência
8	Anomalia no mecanismo de invalidação de blocos
9	Valores incorretos retornados por *swaps*
10	Anomalia na gerência do *dirty-bit*

tado por uma falha do tipo *bit-stuck-at-zero*. A falha 10 está localizada no controlador da *cache*: a lógica de controle falha em gerenciar o bit de modificação (*dirty bit*) para indicar que uma escrita modificou os dados daquele bloco, prejudicando as políticas de *write-back* e de coerência, levando assim ao consumo de um valor obsoleto.

A Tabela 7 resume as falhas descritas, cuja localização é mostrada na Figura 8 (que é uma reprodução local da Figura 6 já descrita).

Para avaliar a eficiência da técnica proposta em comparação à de um verificador convencional, computou-se o *tempo* médio *de verificação*, como explicado a seguir. Primeiramente, mediu-se o tempo que cada verificador levou para encontrar uma dada falha para cada RIT. Então, calculou-se a média dos tempos observados para todos os 240 casos de teste. Por pragmatismo, estipulou-se um limite superior para o tempo de verificação: se um verificador tentou encontrar uma falha por 2 horas, assume-se que a falha nunca será encontrada e a verificação é terminada. Para comparar a efetividade da técnica proposta com a do verificador convencional, computou-se, para cada verificador, o percentual dos casos de teste que expõem uma dada falha, isto é a *cobertura de verificação*.

A Figura 9 mostra o tempo médio de verificação para cada falha. Note que o verificador proposto (*ours*) supera o desempenho do convencional (*conv*) em pelo menos duas ordens de magnitude para todas as falhas, exceto três. Para entender porque o verificador proposto não pôde ser sempre mais rápido do que o convencional, lembre que este requer que um mapeamento ρ seja previamente conhecido. As falhas 4, 6 e 9 acabam induzindo valores para os quais ρ não foi definida, o que pode ser detectado prontamente. Assim, o verificador convencional nem precisa encontrar um ciclo para detectá-las. O

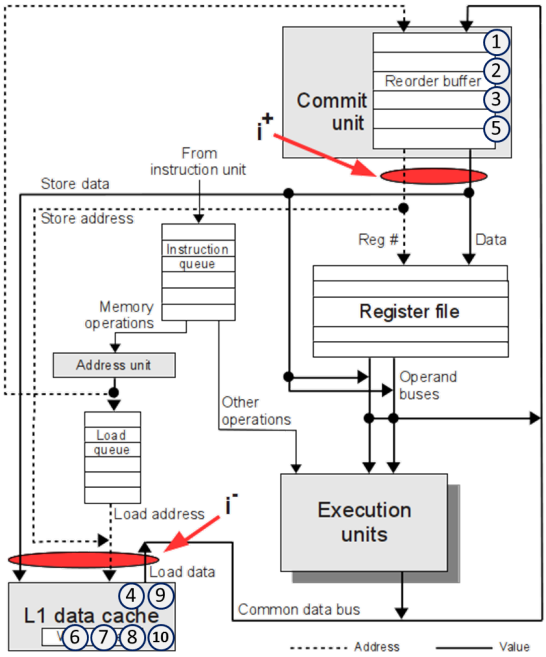

Figura 8 – Localização das falhas injetadas

verificador proposto não se beneficia dessa detecção antecipada, pois foi projetado para resolver também outras instâncias do problema de verificação e não somente VMCM-*read*. Por outro lado, o verificador convencional exibiu detecção tardia para as falhas 1, 5 e 7: foi necessário inserir um maior número de arestas até que as restrições induzidas pelo axioma de valor pudessem produzir um ciclo no grafo de restrições. Algumas vezes, checando estas falhas, o verificador convencional atingiu o limite superior do tempo de verificação para os testes mais longos ($n = 16K$). O verificador proposto, ao contrário, nunca atingiu o limite superior para nenhuma falha. Em média, o verificador proposto foi 272 vezes mais rápido que o convencional. Isto é uma evidência que o emparelhamento estendido é um mecanismo mais eficaz para detectar violações do que a detecção de ciclo em grafo de restrições.

A Figura 10 mostra a cobertura de verificação para cada falha. Observe que o verificador proposto levou a uma cobertura maior ou igual para todas as falhas, comparado ao convencional. Em média, o verificador proposto cobre 90% das falhas, enquanto o convencional cobre 69% delas.

Observe que a falha 5 quase nunca foi encontrada pelo verificador con-

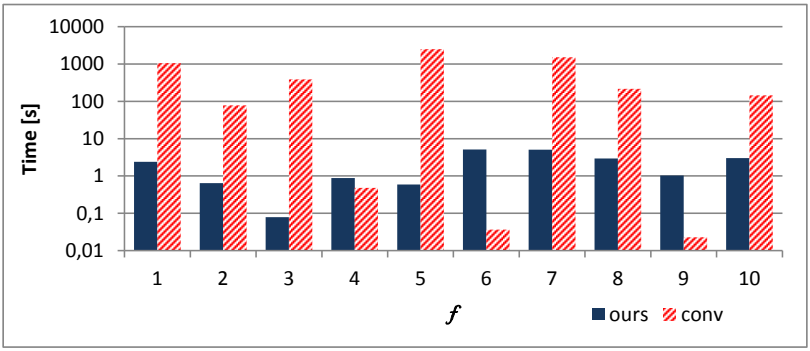

Figura 9 – Tempo médio de verificação

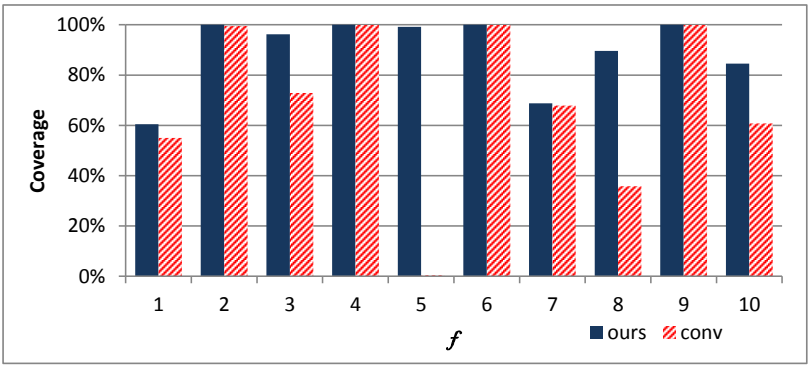

Figura 10 – Cobertura da verificação

vencional. A cobertura extremamente baixa pode ser explicada da seguinte maneira. O MCM sob verificação é um tanto relaxado. Por conta disso, o número de restrições induzidas pelo axioma de ordem é pequeno. Como a falha 5 não induz uma violação do axioma de valor como efeito colateral, o número de restrições é insuficiente para formar um ciclo.

É de se esperar que o verificador convencional tenha mais sucesso em detectar uma falha quando ela resulta numa violação do axioma de ordem que *também* induza uma violação do axioma de valor. Mesmo nesse cenário, a incompletude do verificador convencional pode ainda reduzir substancialmente a cobertura. Por exemplo, embora a falha 8 recaia nesse cenário de dupla violação, o verificador convencional atinge menos da metade da cobertura obtida pelo verificador proposto. Para mais de 60% dos casos de teste, o verificador convencional não consegue inferir que um valor tornou-se obsoleto: a rela-

ção faltante impede que um ciclo se forme no grafo de restrições, levando à não-detecção daquela falha.

Para refletir o fato de que não se conhece *a priori* os parâmetros de gerador de estímulos que levam à melhor cobertura, utilizamos casos de teste com diferentes profundidades (n) e abrangências (s). Portanto, o fato de a técnica proposta não ter encontrado todas as falhas injetadas não é surpreendente, já que nem todos os RITs utilizados nos experimentos são capazes de garantir cobertura total (seja pelo pequeno número de operações seja pelo número exíguo de endereços compartilhados).

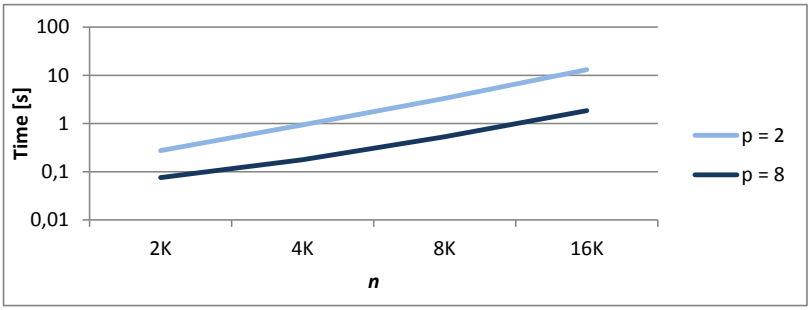

Figura 11 – Crescimento do tempo médio de verificação com o número de instruções

A Figura 11 mostra evidência experimental de que a complexidade de caso médio do verificador proposto é bastante inferior à do pior caso. Com base no universo de 2400 casos de uso gerados, as linhas na figura representam o tempo médio de verificação em dois cenários distintos (p=2 e p=8) conforme varia o número de operações dos 240 casos de teste (n). Embora o tempo de verificação e a complexidade sejam noções distintas, o crescimento do tempo de verificação pode ser usado como uma estimativa para o crescimento assintótico do número médio de operações do Algoritmo 4 [4]. Através de interpolação, verificou-se que o polinômio que melhor aproxima as curvas mostradas na Figura 11 é dominado pelo termo n^3; assim, estima-se que a complexidade média da técnica proposta seja aproximadamente $O(n^3)$ para um dado número de processadores (p).

A baixa complexidade do caso médio pode ser explicada pelo fato de que o esforço computacional do algoritmo de E-matching (que domina a complexidade da verificação) correlata com o grau médio dos vértices de um BVG. Ora, a restrição de unicidade de valores associados a cada endereço (que é um requisito comum à maioria dos verificadores convencionais) resulta

[4] Negligenciando os efeitos do subsistema de memória da estação de trabalho.

em BVGs com baixo grau médio. Para um dado caso de teste, o grau de pior caso seria n/p. Para os casos de teste utilizados nos experimentos (que são inspirados naqueles usados pela maioria dos verificadores convencionais), o valor de n/p está no intervalo $[2K/8, 16K/2]$, enquanto que o grau médio é de apenas $1,13$ arestas incidentes em cada vértice.

Vamos agora distinguir a sensitividade de ambos os verificadores aos parâmetros n, p e s. Primeiramente, a cobertura é analisada do ponto de vista de cada falha (Figuras 12, 13 e 14). Depois, a cobertura e o tempo de execução são avaliados, independente do tipo de falha (Figuras 15, 16 e 17).

A Figura 12 mostra que a cobertura geralmente aumenta quando o número de operações aumenta de 2K para 8K. Isso se explica pelo fato de que para serem expostas, algumas falhas requerem uma certa sequência de operações para se manisfestar. Quanto maior n, maior o número de combinações de sequências de operações presentes no caso de teste, levando a maiores coberturas, inclusive de falhas que necessitam de acessos sequenciais ao mesmo endereço para se manifestar. É justamente por isso que a cobertura não aumenta com o tamanho do caso de teste para falhas que não dependem de um padrão de acesso (2, 4, 5, 6 e 9).

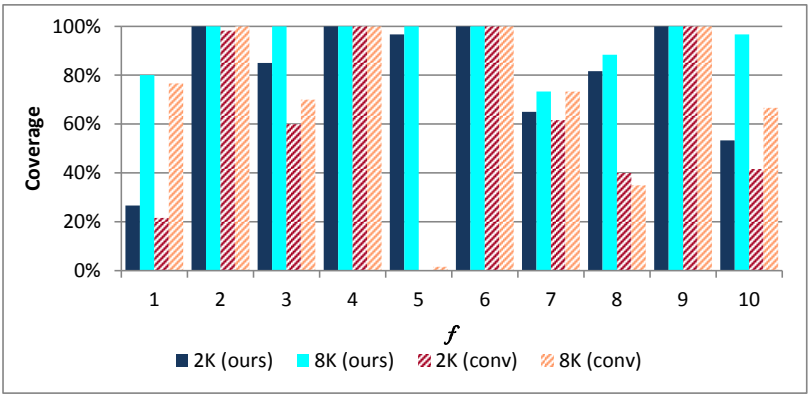

Figura 12 – Cobertura de cada falha (variando n)

A Figura 13 mostra que, para a maioria das falhas (2, 4, 5, 6, 8, 9 e 10), a cobertura do verificador proposto essencialmente não muda ao se aumentar o número de variáveis compartilhadas nos casos de teste. Por um lado, a cobertura da falha 7 aumentou substancialmente. Isso se deve ao fato de a falha requerer um valor de s suficientemente grande para cobrir o pedaço defeituoso do bloco da cache. Por outro lado, a cobertura da falha 1 foi reduzida significativamente, pois necessita de um grande número de

acessos referenciando o mesmo endereço para se manifestar (ao aumentar s, este número diminuiu). A redução na cobertura da falha 3 se explica de maneira similar (embora o impacto tenha sido menor).

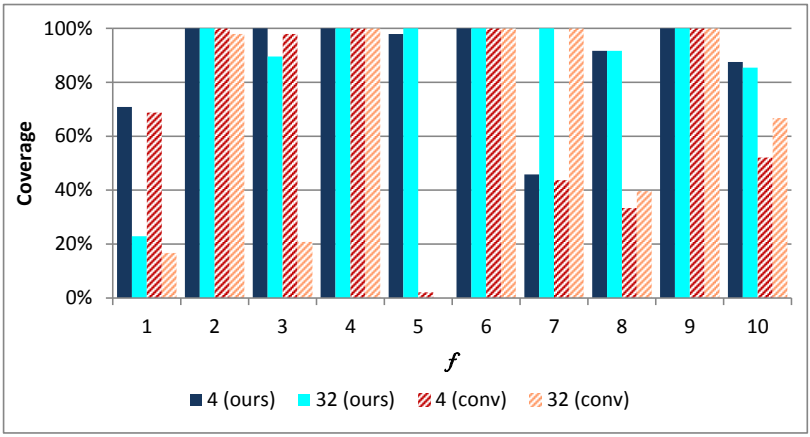

Figura 13 – Cobertura de cada falha (variando s)

A Figura 14 mostra que, quando o número de processadores aumenta de 2 para 4, a cobertura do verificador proposto continuou praticamente a mesma para a maioria das falhas (2, 3, 4, 5, 6, 7 e 9) e melhorou levemente para uma falha que requer maior interação entre processadores para se manifestar (falha 10). Para a falha 1, a redução da cobertura foi causada pela redução de n/p, que é o número de operações realizadas em um dado processador, quando p aumentou. Para a falha 8, a redução da cobertura com o aumento de p é explicada da seguinte maneira: a falha se manifesta quando o mecanismo de coerência falha em processar uma mensagem de invalidação. Ao aumentar o número de processadores, o número de mensagens de controle de coerência requisitando invalidação de um bloco também aumenta, tornando a falha menos evidente.

A Figura 15 mostra o comportamento geral de ambos os verificadores quando o número de operações aumenta, isto é, quando o tamanho do caso de teste cresce. Observe que o verificador proposto alcança maior cobertura com testes menores devido à observabilidade estendida e encontra 20% a mais de falhas que o convencional, independentemente do tamanho do caso de teste. Apesar da maior cobertura, o verificador proposto é duas ordens de magnitude mais rápido.

A Figura 16 mostra o comportamento dos dois verificadores quando o número de variáveis compartilhadas do caso de teste é aumentado. Como

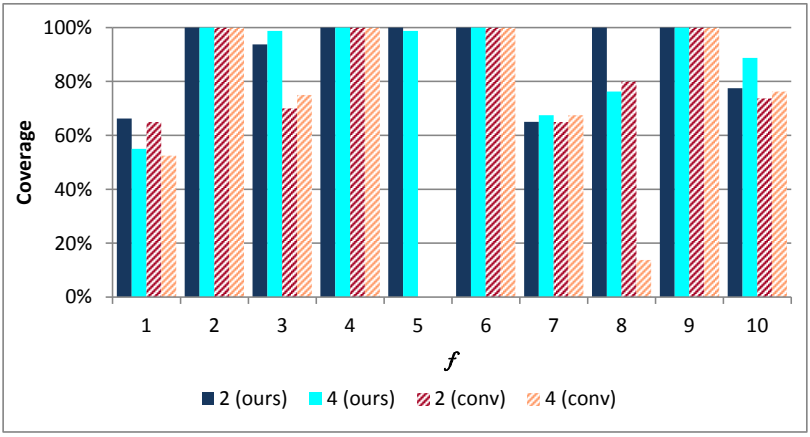

Figura 14 – Cobertura de cada falha (variando p)

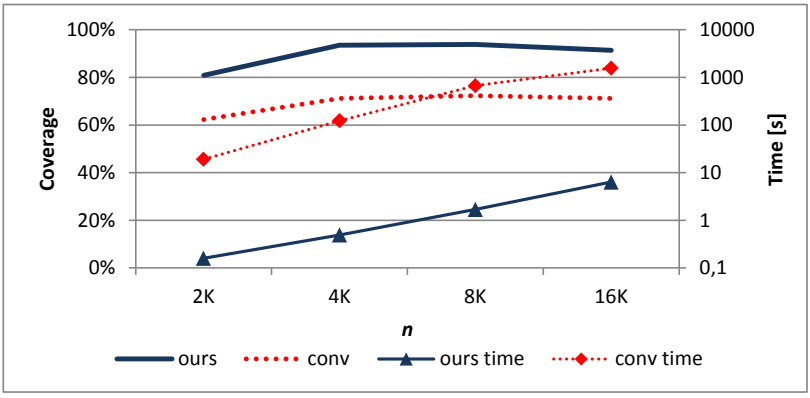

Figura 15 – Cobertura global (variando n)

já observado, o verificador proposto encontra 20% a mais de falhas e a sua cobertura não foi influenciada pelo grau de compartilhamento, enquanto que o tempo de verificação sofreu pouca mudança, ao contrário do verificador convencional, cuja cobertura diminuiu.

A Figura 17 mostra o comportamento de ambos os verificadores quando o número de processadores é aumentado. Observe que a cobertura obtida com o verificador proposto é praticamente independente do número de processadores, ao contrário do convencional, cuja cobertura na verdade diminui. Além disso, o comportamento do tempo de execução revelado pelos experimentos,

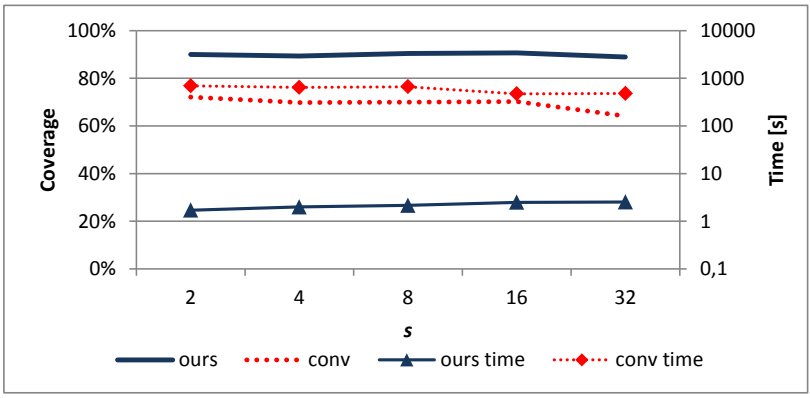

Figura 16 – Cobertura global (variando s)

proporcionado pela decomposição do problema em p instâncias locais, evidencia a qualificação do verificador proposto diante dos desafios de verificação impostos pelo crescimento do número de processadores em um chip.

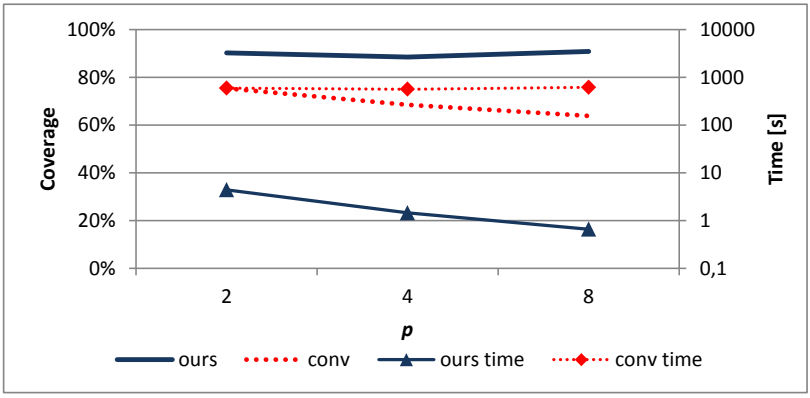

Figura 17 – Cobertura global (variando p)

O capítulo seguinte apresenta as conclusões deste trabalho e as perspectivas de trabalhos futuros.

6 CONCLUSÕES, PERSPECTIVAS E TRABALHOS FUTUROS

Este capítulo combina a discussão de garantias teóricas de verificação (apresentadas no Capítulo 4) com a avaliação experimental da eficiência e da eficácia do verificador (apresentadas no Capítulo 5), a fim de prover uma interpretação unificada do mérito da técnica de verificação proposta.

6.1 CONCLUSÕES E PERSPECTIVAS

Decomposição como chave para conciliar completude e eficiência

Esta dissertação formulou uma decomposição do problema de verificação de consistência de memória que é induzida pela maior observabilidade disponível em representações executáveis, típicas de um fluxo de projeto ESL (BAILEY; MARTIN; PIZIALI, 2007). Essa decomposição deliberada permitiu o mapeamento de parte do problema de verificação para um problema de emparelhamento estendido de grafos bipartidos (MARCILIO et al., 2009), dele herdando algumas garantias de verificação.

Técnica praticamente independente de microarquitetura

A técnica requer o monitoramento de dois pontos por processador, que foram selecionados judiciosamente para manter a técnica de verificação praticamente independente da escolha de microarquitetura. Apesar de os experimentos terem sido realizados com programas sintéticos (casos de teste aleatórios gerados automaticamente), a técnica pode ser aplicada à execução de programas concorrentes reais (embora neste caso não se beneficie do controle de cobertura).

Baixa complexidade média ao usar geradores de casos de testes aleatórios

As provas de completude foram confirmadas pela evidência experimental: no universo de 240 casos de teste não foram sinalizados falsos positivos e no universo de 2400 casos de uso não foram sinalizados falsos negativos. A evidência experimental mostrou que, para um dado número de processadores (p), a complexidade de caso médio (aproximadamente $O(n^3)$) da técnica de verificação proposta é bem inferior à de pior caso ($O(n^6)$) quando se utilizam geradores de casos de teste típicos (instruções aleatórias e valores únicos associados a cada endereço).

Eficiência e eficácia superiores às de verificadores convencionais

Os resultados experimentais mostraram que a técnica proposta, em-

bora completa, é mais rápida do que um verificador convencional incompleto (HANGAL et al., 2004). Ao explorar a maior observabilidade de uma representação executável, a técnica proposta levou a uma redução de duas ordens de magnitude no tempo de verificação e permitiu a detecção de 20% de falhas a mais. Obviamente, a técnica proposta se mostraria ainda mais rápida se fosse comparada a um verificador convencional completo (MANOVIT; HANGAL, 2006), devido ao esforço computacional requerido por este último para *backtracking*.

Portanto, pode-se concluir que vale a pena usar uma técnica de verificação especialmente desenvolvida para ser aplicada a representações executáveis de um sistema integrado multiprocessado, ao invés de reutilizar verificadores convencionais originalmente desenvolvidos para teste pós-prototipagem. Há três razões para isso: o menor tempo para encontrar uma falha, a maior cobertura de falhas e o menor tempo total gasto em verificação (redução esta que resulta da maior cobertura: menos tempo ocioso é gasto executando casos de teste que nunca encontrarão uma falha).

Eficiência em face da demanda por maior paralelismo entre instruções

A inferioridade do verificador convencional pode ser explicada por duas razões. A primeira razão resulta de sua incompletude: algumas falhas jamais são detectadas, ocasionando um aumento no tempo total de verificação. A segunda é devida à correlação entre restrições e desempenho: quanto mais arestas são adicionadas ao grafo de restrição, mais rápida é a detecção de um ciclo (se houver). A verificação da implementação de um MCM altamente relaxado (como o ARO) leva mais tempo se comparado com a de um MCM mais estrito (como o TSO).

Como as aplicações de Computação Embarcada e Móvel demandam níveis de *throughput* crescentes, é de se esperar que, além do aumento do número de processadores, tenha-se que garantir um melhor aproveitamento de paralelismo entre instruções (ILP). Assim, é de se esperar que os sistemas integrados multiprocessados tendam a requerer modelos de consistência mais relaxados para prover desempenho adequado. Sob esta hipótese, espera-se que técnicas como a proposta tenham vantagem sobre verificadores convencionais, que são menos eficientes em modelos mais relaxados.

Eficiência em face do crescimento do número de processadores

Os resultados experimentais mostraram que a cobertura da técnica proposta é praticamente independente do número de processadores. Além disso, o tempo de verificação pode se beneficiar da paralelização do algoritmo de verificação. Esses fatos tornam a técnica proposta mais adequada a enfrentar os desafios de verificação impostos por um número crescente de processadores.

As técnicas dinâmicas convencionais que assumem observabilidade limitada (*black-box*) e recorrem a uma única estrutura de dados centralizada (grafo de restrições) para a detecção de falhas dificilmente seriam capazes de enfrentar a complexidade crescente dos subsistemas de memória sem sensível redução na cobertura.

6.2 TRABALHOS FUTUROS

Comparação com verificador convencional completo

A implementação de uma versão completa de verificador convencional similar à proposta em (MANOVIT; HANGAL, 2006) está em andamento. Embora esta comparação seja desnecessária no que se refere a avaliar a eficiência da técnica proposta (como explicado anteriormente), ela é útil para aferir a cobertura da técnica proposta frente a outra técnica completa.

Comparação com verificador baseado em *scoreboard*

Pretende-se implementar um verificador baseado em *scoreboard* relaxado (SHACHAM et al., 2008) para compará-lo com o proposto e também com o verificador convencional. Esta comparação contribuirá para se avaliar o compromisso entre garantias e velocidades de verificação. Note que o *scoreboard* relaxado, apesar de ser incompleto como o convencional (*post-mortem*), tende a ser mais eficiente (*on-the-fly*).

Avaliação do impacto da técnica ao verificar outros modelos de consistência

Pretende-se avaliar a eficiência e a eficácia da técnica proposta ao verificar outros modelos de consistência, em especial o popular TSO e modelos mais recentes, tais como TCC.

Explorar o desempenho possibilitado pelo algoritmo paralelo

Embora o algoritmo de verificação proposto seja paralelo, sua prototipagem para fins de experimentação não explorou o paralelismo. Pretende-se reavaliar a eficiência da técnica proposta ao se explorar tal paralelismo.

REFERÊNCIAS BIBLIOGRÁFICAS

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

ALBERTINI, B. et al. A computational reflection mechanism to support platform debugging in SystemC. In: *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2007. p. 81–86. ISBN 978-1-59593-824-4.

ARM HOLDINGS. *ARM Cortex A-15*. 2011. <http://www.arm.com/products/processors/cortex-a/cortex-a15.php>.

ARVIND, A.; MAESSEN, J.-W. Memory model = instruction reordering + store atomicity. In: *Proceedings of the 33rd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006. (ISCA '06), p. 29–40. ISBN 0-7695-2608-X.

BAILEY, B.; MARTIN, G.; PIZIALI, A. *ESL design and verification: a prescription for electronic system-level methodology*. [S.l.]: Morgan Kaufmann, 2007. (The Morgan Kaufmann series in systems on silicon). ISBN 9780123735515.

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

BLACK, D.; DONOVAN, J. *SystemC: from the ground up*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 0387292403.

CANTIN, J. F.; LIPASTI, M. H.; SMITH, J. E. Dynamic verification of cache coherence protocols. In: *Workshop on Memory Performance Issues*. [S.l.: s.n.], 2001.

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

CONDON, A. et al. Using Lamport clocks to reason about relaxed memory models. In: *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*. [S.l.: s.n.], 1999. p. 270 –278.

CORMEN, T. et al. *Introduction to Algorithms*. 3. ed. [S.l.]: Cambridge, MA: The MIT Press, 2009. ISBN 0262033844.

DUBOIS, M.; THAKKAR, S. *Scalable shared memory multiprocessors*. [S.l.]: Kluwer Academic Publishers, 1992. ISBN 9780792392194.

GEM5. *GEM5 Simulator*. 2011. <http://www.m5sim.org/>.

GIBBONS, P.; KORACH, E. The complexity of sequential consistency. In: *IEEE Symposium on Parallel and Distributed Processing*. [S.l.: s.n.], 1992. p. 317 –325.

GIBBONS, P. B.; KORACH, E. Testing shared memories. *SIAM Journal on Computing*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 26, p. 1208–1244, August 1997. ISSN 0097-5397.

HAMMOND, L. et al. Transactional memory coherence and consistency. In: *Proceedings of the 31st International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2004. (ISCA '04). ISBN 0-7695-2143-6.

HANGAL, S. et al. TSOtool: a program for verifying memory systems using the memory consistency model. In: *Proceedings of the 31st International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2004. (ISCA '04), p. 114–. 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.

HILL, M. Multiprocessors should support simple memory consistency models. *Computer*, v. 31, n. 8, p. 28 –34, aug 1998. ISSN 0018-9162.

JAHRE, M.; GRANNAES, M.; NATVIG, L. A quantitative study of memory system interference in chip multiprocessor architectures. *High Performance Computing and Communications, 10th IEEE International Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, p. 622–629, 2009.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, ACM, New York, NY, USA, v. 21, p. 558–565, July 1978. ISSN 0001-0782.

LANDIN, A.; HAGERSTEN, E.; HARIDI, S. Race-free interconnection networks and multiprocessor consistency. *ACM SIGARCH Computer*

Architecture News, ACM, New York, NY, USA, v. 19, p. 106–115, April 1991. ISSN 0163-5964.

LOGHI, M.; PONCINO, M.; BENINI, L. Cache coherence tradeoffs in shared-memory MPSoCs. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 5, p. 383–407, May 2006. ISSN 1539-9087.

LUDDEN, J. M. et al. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems. *IBM J. Res. Dev.*, 2002.

MANOVIT, C.; HANGAL, S. Efficient algorithms for verifying memory consistency. In: *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2005. (SPAA '05), p. 245–252. ISBN 1-58113-986-1.

MANOVIT, C.; HANGAL, S. Completely verifying memory consistency of test program executions. In: *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: *Proceedings of the 46th ACM/IEEE Design Automation Conference*. [S.l.: s.n.], 2009. (DAC '09), p. 448 –453. ISSN 0738-100X.

MARCíLIO, G. M. *Verificação funcional pós-particionamento em sistemas integrados de hardware e software*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, SC, 2008.

MEIXNER, A.; SORIN, D. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *IEEE Transactions on Dependable and Secure Computing*, v. 6, n. 1, p. 18 –31, jan.-march 2009. ISSN 1545-5971.

MEIXNER, A.; SORIN, D. J. Dynamic verification of sequential consistency. In: *Proceedings of the 32nd International Symposium on Computer Architecture*. [S.l.: s.n.], 2005. (ISCA '05), p. 482 – 493. ISSN 1063-6897.

MIPS TECHNOLOGIES. *MIPS 1074K*. 2011. <http://www.mips.com/products/cores/32-64-bit-cores/mips32-1074k/>.

MOSBERGER, D. Memory consistency models. *SIGOPS Operating Systems Review*, ACM, New York, NY, USA, v. 27, p. 18–26, January 1993. ISSN 0163-5980.

PATTERSON, D.; HENNESSY, J. *Computer organization and design: the hardware/software interface*. 4. ed. [S.l.]: Elsevier Morgan Kaufmann, 2009. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780123744937.

RAMBO, E. A.; HENSCHEL, O. P.; SANTOS, L. C. V. Automatic generation of memory consistency tests for chip multiprocessing. In: *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. [S.l.: s.n.], 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)*. [S.l.: s.n.], 2012.

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

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

SIEK, J.; LEE, L.-Q.; LUMSDAINE, A. *The Boost Graph Library: User Guide and Reference Manual*. [S.l.]: Addison-Wesley, 2002.

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

SORIN, D. J. et al. *Lamport Clocks: Reasoning About Shared Memory Correctness*. [S.l.], 1998.

SRIDHARAN, V.; KAELI, D. R. Using hardware vulnerability factors to enhance AVF analysis. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 38, p. 461–472, June 2010. ISSN 0163-5964.

ZHENG, H. et al. Decoupled DIMM: building high-bandwidth memory system using low-speed DRAM devices. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 37, p. 255–266, June 2009. ISSN 0163-5964.

APÊNDICE A – Provas dos Lemas e do Teorema

As provas formais aqui apresentadas são incluídas por razões de completude, embora não tenham sido obtidas pelo autor desta dissertação. As provas foram elaboradas pelo Prof. Luiz Cláudio V. dos Santos como contrapartida ao trabalho cooperativo reportado em (RAMBO; HENSCHEL; SANTOS, 2012).

Os Lemas 4.1, 4.2, 4.3 e 4.4 e o Teorema 4.1 são reapresentados a seguir juntamente com as suas respectivas provas.

Lema 4.1: O Algoritmo 1 retorna verdadeiro se e somente se os Axiomas 2.1, 2.2 e 2.3, bem como a Propriedade 2.3, são válidos para $\mathscr{O}^i$.

Prova: Como `read-own-write-ok` é válido quando o Algoritmo 1 retorna verdadeiro, conclui-se que o Axioma 2.3 certamente é válido para produtores locais. Dadas duas operações x e y, sejam $ax_1(x,y)$ e $ax_2(x,y)$ predicados especificados pelos Axiomas 2.1 e 2.2 para um MCM genérico. Sejam $R_1^i = \{(x,y) \in \mathscr{O}^i \times \mathscr{O}^i | ax_1(x,y)\}$ e $R_2^i = \{(x,y) \in \mathscr{S}^i \times \mathscr{S}^i | ax_2(x,y)\}$, isto é, os conjuntos de pares ordenados satisfazendo os Axiomas 2.1 e 2.2, respectivamente. O Algoritmo 1 (na linha 9) atribui a R todos os elementos em $R_1^i \cup R_2^i$. Além disso, todos os valores equivalentes observados pelos monitores m_i^+ e m_i^- são atribuídos à relação E (na linha 7). Dois teoremas em (MARCILIO et al., 2009) garantem que, quando E é uma relação de equivalência, `proper-matching` retorna verdadeiro se e somente se duas condições são válidas simultaneamente:

- R^i foi satisfeito para todos os eventos observados por m_i^-;
- Um mapeamento de um-para-um $M \subseteq E$ foi encontrado.

Como E é de fato uma relação de equivalência, a primeira condição prova que os Axiomas 2.1 e 2.2 são válidos para operações emitidas pelo processador i. A segunda condição prova que a Propriedade 2.3 é válida para o processador i. ◇

Lema 4.2. O Axioma 2.3 vale para os produtores globais se e somente se a Propriedade 2.1 for válida.

Prova: Seja σ_a^i a notação abreviada para $\{S_a^k | S_a^k \leq L_a^i\}$ e seja $\sigma_a^i(j) \subset \sigma_a^i$ as escritas emitidas pelo processador j.

Necessidade: A Propriedade 2.1 garante que $\forall S_a^i, S_a^k \in \mathscr{S} : (S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i)$. Como $\sigma_a^i \subset \mathscr{S}$, tem-se que: $\forall S_a^i, S_a^k \in \sigma_a^i : (S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i)$. Como $\leq$ é uma ordem linear em σ_a^i, a afirmação $\exists s \in \sigma_a^i : s = Max_{\leq}[\sigma_a^i]$ é válida para todo L_a^i. Quando L_a^i executa, ela deve consumir o último valor armazenado no endereço a; portanto, $Val[L_a^i] = Val[\sigma_a^i]$ é válido para todo $a \in A$.

Suficiência: Dado um par de escritas (x,y), seja $p_1(x,y)$ o predicado especificado pela Propriedade 2.1. o Axioma 2.3 garante que, para todo L_a^i, tem-se $\exists s \in \sigma_a^i : s = Max_{\leq}[\sigma_a^i] = Max_{\leq}[\sigma_a^i(1) \cup \cdots \cup \sigma_a^i(j) \cup \cdots \cup \sigma_a^i(p)] = Max_{\leq}[\{Max_{\leq}[\sigma_a^i(1)], \cdots, Max_{\leq}[\sigma_a^i(j)], \cdots, Max_{\leq}[\sigma_a^i(p)]\}]$.

A existência de máximos locais prova que todas as escritas em $\sigma_a^i(j)$ são ordenadas linearmente por $\leq$, isto é, que $\forall (x,y) \in \sigma_a^i(j) \times \sigma_a^i(j) : p_1(x,y)$ é valido para todo L_a^i ou, equivalentemente, que a afirmação $\forall (x,y) \in \mathscr{S}^j \times \mathscr{S}^j : p_1(x,y)$ é válida, pois o valor produzido por cada escrita é consumido por pelo menos uma leitura. Por simplicidade, seja M_a^i a notação abreviada para o conjunto $\{Max_{\leq}[\sigma_a^i(1)], \cdots, Max_{\leq}[\sigma_a^i(j)], \cdots, Max_{\leq}[\sigma_a^i(p)]\}$. A existência de um máximo global prova que todas as operações de escrita no conjunto M_a^i estão linearmente ordenadas por $\leq$. Portanto a afirmação $\forall (x,y) \in M_a^i \times M_a^i | : p_1(x,y)$ é válida para todo L_a^i.

Como o valor produzido por toda escrita é consumido por pelo menos uma leitura L_a^i, conclui-se que o conjunto $\cup_{i,a} M_a^i$ é o conjunto de todos os produtores de valores, isto é, $\cup_{i,a} M_a^i = \mathscr{S}$. Como $\mathscr{S} = \{\mathscr{S}^1 \cup \cdots \cup \mathscr{S}^j \cup \cdots \cup \mathscr{S}^k \cdots \cup \mathscr{S}^p\}$ é uma partição e cada escrita em M_a^i foi emitida por um processador distinto, então cada escrita em M_a^i pertence a exatamente um componente da partição. Como, para cada par $(x,y) \in M_a^i$, as operações de escrita $x = Max_{\leq}[\sigma_a^i(j)]$ e $y = Max_{\leq}[\sigma_a^i(k)]$) foram emitidas por processadores distintos j e k, conclui-se que a afirmação $\forall (x,y) \in \mathscr{S}_j \times \mathscr{S}_k | : p_1(x,y)$ é válida para $j \neq k$. Como uma afirmação similar foi provada para $j = k$, conclui-se que a Propriedade 2.1 é válida. $\diamond$

Lema 4.3. Para qualquer sistema de memória com coerência de cache, a ordem $\leq$ especificada por um MCM genérico é indistinguível da ordem de *timestamps* $<_t$.

Prova: Precisa-se provar que $\forall S_a^i, S_a^k \in \mathscr{S} : (S_a^i \leq S_a^k) \Leftrightarrow (S_a^i <_t S_a^k)$.

Necessidade: Para a correta execução da representação do projeto do sistema de memória, duas operações especificadas como ordenadas devem ser observadas em tempos sucessivos, isto é, $x \leq y \Rightarrow t(x) < t(y)$. Assim, assumindo $x = S_a^i$ e $y = S_a^k$, tem-se: $S_a^i \leq S_a^k \Rightarrow t(S_a^i) < t(S_a^k)$ para todo $a \in A$.

Suficiência: Como $\leq$ é uma ordem parcial, um dos seguintes cenários deve ser observado:

- $x \leq y \Rightarrow t(x) < t(y)$
- $y \leq x \Rightarrow t(x) > t(y)$
- $\neg(x \leq y) \wedge \neg(y \leq x) \Rightarrow t(x)$ e $t(y)$ são arbitrários.

Portanto, conclui-se que:
$t(x) < t(y) \Rightarrow (x \leq y) \vee (\neg(x \leq y) \wedge \neg(y \leq x))$, i.e.

$t(x) < t(y) \Rightarrow (x \leq y) \vee \neg((x \leq y) \vee (y \leq x))$.

Assumindo que $x = S_a^i$ e $y = S_a^k$, tem-se:
$t(S_a^i) < t(S_a^k) \Rightarrow (S_a^i \leq S_a^k) \vee \neg((S_a^i \leq S_a^k) \vee (S_a^k \leq S_a^i))$ para todo $a \in A$. Como a Propriedade 2.1 deve ser válida para todo sistema de memória com coerência de cache, a segunda cláusula da disjunção é falsa. Portanto, $t(S_a^i) < t(S_a^k) \Rightarrow (S_a^i \leq S_a^k)$ é válido.

Logo, $(S_a^i \leq S_a^k) \Leftrightarrow t(S_a^i) < t(S_a^k)$ e, pela Definição 4.2, temos que $(S_a^i \leq S_a^k) \Leftrightarrow (S_a^i <_t S_a^k)$ para todo $a \in A$. ◇

Lema 4.4. Para qualquer sistema de memória com coerência de cache, o Algoritmo 3 retorna verdadeiro se e somente se o Axioma 2.3 é satisfeito para produtores globais.

Prova: Como os eventos são visitados na ordem crescente de *timestamps*, o último valor atribuído a $V[h(a)]$ pelo Algoritmo 3 (nas linhas 9–10) é $Max_{<_t}[\sigma_a^i]$. Como ele somente retorna falso se $Val[L_a^i] \neq Val[Max_{<_t}[\sigma_a^i]]$ para algum $a \in A$ (linhas 11–12), ele retorna verdadeiro se e somente se $Val[L_a^i] = Val[Max_{<_t}[\sigma_a^i]]$ para todo $a \in A$. Para assegurar que esta última afirmação é equivalente ao Axioma 2.3, tem-se que provar que $Max_{<_t}[\sigma_a^i] = Max_{\leq}[\sigma_a^i]$, i.e. é necessário mostrar que $(S_a^i \leq S_a^k) \Leftrightarrow (S_a^i <_t S_a^k)$ é válido para todo $S_a^i, S_a^k \in \sigma_a^i$. Ora, isto é garantido pelo Lema 4.3, já que $\forall a \in A : \sigma_a^i \subseteq \mathscr{S}$ ◇

Teorema 4.1. Para qualquer sistema de memória com coerência de cache e para qualquer MCM que não requeira ordenação total de escritas, o Algoritmo 4 retorna verdadeiro se e somente se todos os axiomas do MCM são válidos para o comportamento observado, induzido por um dado caso de teste.

Prova: Sejam l^i = `local-behavior-OK`(MCM, T_i^+, T_i^-) e
g = `global-behavior-ok`(T_1^-, T_2^-, $\cdots$, T_p^-).

Pelo Lema 4.1, tem-se $l^1 \wedge l^2 \wedge \cdots \wedge l^i \wedge \cdots \wedge l^p \Leftrightarrow$ (Axioma 2.1 vale) $\wedge$ (Axioma 2.2 vale para todo $(u,v) \in \mathscr{S}^i \times \mathscr{S}^i$) $\wedge$ (Axioma 2.3 vale para produtores locais) $\wedge$ (Propriedade 2.3 vale).

Pelo Lema 4.4, tem-se $g \Leftrightarrow$ (Axioma 2.3 vale para produtores globais).

Como para MCMs que não requeiram ordenação total de escritas vale a sentença Propriedade 2.1 $\Leftrightarrow$ Axioma 2.2 e como pelo Lema 4.2 vale a sentença Propriedade 2.1 $\Leftrightarrow$ (Axioma 2.3 para produtores globais), tem-se Axioma 2.2 $\Leftrightarrow$ (Axioma 2.3 para produtores globais).

Diante disso e considerando também que (Axioma 2.3 para produtores globais) $\Rightarrow$ Propriedade 2.3, conclui-se que $l^1 \wedge l^2 \wedge \cdots \wedge l^i \wedge \cdots \wedge l^p \wedge g \Leftrightarrow$ (Axiomas 2.1, 2.2 e 2.3 valem). ◇