

Olav Philipp Henschel

**VERIFICAÇÃO DE CONSISTÊNCIA E COERÊNCIA DE
MEMÓRIA COMPARTILHADA PARA MULTIPROCESSAMENTO
EM CHIP**

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Luiz Cláudio Villar dos Santos

Florianópolis

2014

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

Henschel, Olav Philipp

Verificação de consistência e coerência de memória
compartilhada para multiprocessamento em chip / Olav
Philipp Henschel ; orientador, Luiz Cláudio Villar dos
Santos - Florianópolis, SC, 2014.

91 p.

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

Inclui referências

1. Ciência da Computação. 2. Multicores. 3. Verificação.
4. Modelos de memória. I. Santos, Luiz Cláudio Villar dos.
II. Universidade Federal de Santa Catarina. Programa de Pós-
Graduação em Ciência da Computação. III. Título.

Olav Philipp Henschel

**VERIFICAÇÃO DE CONSISTÊNCIA E COERÊNCIA DE
MEMÓRIA COMPARTILHADA PARA MULTIPROCESSAMENTO
EM CHIP**

Esta Dissertação foi julgada aprovada para a obtenção do Título de “Mestre em Ciência da Computação” e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 29 de agosto 2014.

Prof. Dr. Ronaldo dos Santos Mello
Coordenador do Programa

Banca Examinadora:

Prof. Dr. Luiz Cláudio Villar dos Santos
Orientador
Universidade Federal de Santa Catarina

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

Prof. Dr. José Luís Almada Güntzel
Universidade Federal de Santa Catarina

Prof. Dr. Djones Vinicius Lettnin
Universidade Federal de Santa Catarina

AGRADECIMENTOS

À minha família, pelo incentivo.

Ao meu orientador, Prof. Dr. Luiz Cláudio Villar dos Santos, por suas indispensáveis sugestões e contribuições técnicas.

Aos membros da banca, Prof. Dr. Rodolfo Jardim de Azevedo, Prof. Dr. José Luís Almada Güntzel e Prof. Dr. Djones Vinicius Lettnin, pelas diversas contribuições recebidas na avaliação do trabalho.

Aos colegas do ECL e NIME. Em particular a Eberle A. Rambo, Leandro S. Freitas, Gabriel G. Gava e Gabriel A. G. Andrade por terem contribuído diretamente com a infraestrutura utilizada neste trabalho.

Ao CNPq, no âmbito do edital GM/GD-Cotas do Programa de Pós-Graduação, processo número 131114/2012-3, pelo fomento à execução deste trabalho.

RESUMO

O multiprocessamento em chip sob a crescente demanda por desempenho leva a um número crescente de núcleos de processamento, que interagem através de uma complexa hierarquia de memória compartilhada, a qual deve obedecer a requisitos de coerência e consistência, capturados na interface hardware-software na forma de um modelo de memória. Dada uma execução de um programa paralelo, verificar se a hierarquia obedece aqueles requisitos é um problema intratável quando a observabilidade do sistema restringe-se a um *trace* de memória para cada processador, tal como ocorre em um *checker* dinâmico pós-silício. Esses *checkers* (baseados em inferências sobre *traces*) requerem o uso de *backtracking* para excluir falsos negativos. Por outro lado, *checkers* pré-silício podem se beneficiar da observabilidade ilimitada de representações de projeto para induzir um problema de verificação que pode ser resolvido em tempo polinomial (sem o uso de *backtracking*) e com plenas garantias de verificação (sem falsos negativos nem falsos positivos). Esta dissertação faz uma avaliação experimental comparativa de *checkers* dinâmicos baseados em diferentes mecanismos (inferências, emparelhamento em grafo bipartido, *scoreboard* única e múltiplas *scoreboards*). Os *checkers* são comparados para exatamente o mesmo conjunto de casos de teste: 200 programas paralelos não sincronizados, gerados de forma pseudo-aleatória, obtidos variando a frequência de ocorrência de instruções (4 mixes), o número de endereços compartilhados (entre 2 e 32) e o número total de operações de memória (entre 250 e 64K). A partir de uma mesma representação pré-validada do sistema, foram construídas oito representações derivadas, cada uma contendo um erro de projeto distinto. Para reproduzir condições compatíveis com as tendências arquiteturais, os *checkers* foram comparados ao verificar um modelo com máxima relaxação de ordem de programa (bastante similar ao usado, por exemplo, nas arquiteturas Alpha e ARMv7) para sistemas contendo de 2 a 32 núcleos de processamento. Não é do conhecimento do autor a existência na literatura de uma avaliação experimental tão ampla. Os resultados mostram a inviabilidade do uso de *checkers* baseados em inferências em tempo de projeto: têm o mais alto esforço computacional e a maior taxa de crescimento com o aumento do número de processadores. A avaliação indica que a forma mais eficiente de construir um *checker* pré-silício corresponde a uma observabilidade de três pontos de monitoramento por processador, ao uso de verificação *on-the-fly* (ao invés de análise *post-mortem*) e à utilização de múltiplos mecanismos para verificar separadamente e em paralelo os subespaços de verificação definidos pelo escopo individual de cada processador, enquanto os subespaços

entre processadores são verificados globalmente. Como um desdobramento da avaliação experimental, a dissertação identifica uma deficiência comum a todos os *checkers* analisados: sua inadequação para verificar modelos de memória com fraca atomicidade de escrita, exatamente aqueles apontados como tendência e já presentes em arquiteturas recentes (e.g. ARMv8). Diante disso, a dissertação propõe algoritmos generalizados capazes de verificar tais modelos.

Palavras-chave: multicores, verificação, modelos de memória

ABSTRACT

Chip multiprocessing under the growing demand for performance leads to a growing number of processing cores, which interact through a complex shared-memory hierarchy that must satisfy coherence and consistency requirements captured as a memory model in the hardware-software interface. Given an execution of a parallel program, verifying if the hierarchy complies to those requirements is an intractable problem when the system observability is limited to a memory trace per processor, as in dynamic post-silicon checkers. Those checkers (based on inferences over traces) require the use of backtracking to avoid false negatives. On the other hand, pre-silicon checkers may benefit from the unlimited observability of design representations to induce a verification problem that may be solved in polynomial time (without the use of backtracking) with full verification guarantees (i.e. neither false negatives nor false positives). This dissertation provides an experimental evaluation of dynamic checkers based on different mechanisms (inferences, bipartite graph matching, single scoreboard and multiple scoreboards). The checkers are compared under exactly the same set of test cases: 200 non-synchronized parallel programs, generated pseudo-randomly, obtained by varying the frequency of instructions (4 mixes), the number of shared addresses (between 2 and 32) and the total number of memory operations (between 250 and 64K). From the same pre-validated system representation, eight distinct representations were built, each one containing a single and unique design error. To reproduce conditions compatible with architectural trends, the checkers were compared while verifying a memory model with maximal relaxation of program order (similar, for example, to those used in Alpha and ARMv7 architectures) and systems containing 2 to 32 processing cores. To the author's best knowledge, no broader experimental evaluation is available in the literature. The results show that the use of inference-based checkers at design time is impractical: they have the highest computational effort and the highest rate of growth with the number of cores. The evaluation shows that the most efficient way of building a pre-silicon checker corresponds to three observable points per core, the use of on-the-fly analysis (instead of post-mortem) and the usage of multiple engines to check the verification subspaces defined by the scope of each processor independently and in parallel, while checking globally the inter-processor subspaces. As a spin-off from the experimental evaluation, the dissertation identifies a deficiency common to all analyzed checkers: their unsuitability to handle memory models with weak write atomicity, which are precisely those pointed out as the trend and are present in architectures already

in the market (e.g. ARMv8). In face of this, the dissertation proposes generic algorithms capable of verifying such models.

Keywords: multicores, verification, memory models

LISTA DE FIGURAS

Figura 1	Exemplo ilustrativo para diferentes modelos de memória	20
Figura 2	Um programa, seus <i>traces</i> e dois comportamentos atômicos . .	30
Figura 3	Ordenamento de eventos sob forte atomicidade de escrita	30
Figura 4	Um modelo genérico de sistema com memória compartilhada	32
Figura 5	Diferentes eficácias para erros distintos	57
Figura 6	Eficácia para casos de teste com tamanho crescente	58
Figura 7	Eficiência para casos de teste com tamanho crescente	59
Figura 8	Eficácia com o aumento do número de processadores	61
Figura 9	Eficiência com o aumento do número de processadores	62
Figura 10	Eficiência com o número crescente de endereços	63
Figura 11	Eficácia com o número crescente de endereços	64
Figura 12	Um programa, seus <i>traces</i> e dois comportamentos não-atômicos	68
Figura 13	Ordenamento de eventos sob fraca atomicidade de escrita	69
Figura 14	Um modelo genérico de sistema com memória compartilhada	72
Figura 15	Estrutura do checker	74
Figura 16	Rotina <code>parallel-checker()</code>	75
Figura 17	Rotina <code>consistent-order(<i>i</i>)</code>	75
Figura 18	Processo <code>check-on-commit(<i>i</i>)</code>	76
Figura 19	Processo <code>check-on-squash(<i>i</i>)</code>	77
Figura 20	Processo <code>check-on-completion(<i>i</i>)</code>	78
Figura 21	Rotina <code>match(<i>i</i>, <i>e_m</i>)</code>	78
Figura 22	Processo <code>coherent-values(<i>i</i>)</code>	79
Figura 23	Processo <code>coherent-order(<i>i</i>)</code>	80

LISTA DE TABELAS

Tabela 1	Principais características dos <i>checkers</i> dinâmicos	39
Tabela 2	Proporções de operações de cada tipo	44
Tabela 3	Caracterização dos erros de projeto	47
Tabela 4	Crescimento do número de núcleos em <i>smartphones</i>	49
Tabela 5	<i>Checkers</i> sob comparação	50
Tabela 6	Responsáveis pelas implementações dos <i>checkers</i> utilizados .	84
Tabela 7	Autores da generalização proposta no Capítulo 6.	85

LISTA DE ABREVIATURAS E SIGLAS

CMP	<i>Chip Multiprocessing</i>
ESL	<i>Electronic System Level</i>
PC	<i>Processor Consistency</i>
PSO	<i>Partial Store Ordering</i>
ROB	<i>Reorder Buffer</i>
RTL	<i>Register-Transfer Level</i>
SoC	<i>System-on-chip</i>
TSO	<i>Total Store Order</i>
WO	<i>Weak Ordering</i>
EXM	<i>Checker baseado em emparelhamento de grafos bipartidos</i> ..
IBE	<i>Checker de melhor-esforço baseado em inferências</i>
IBT	<i>Checker baseado em inferências com backtracking</i>
SSB	<i>Checker baseado em scoreboard relaxada</i>
MSB	<i>Checker baseado em múltiplas scoreboards relaxadas</i>

SUMÁRIO

1	INDRODUÇÃO	19
1.1	MODELO DE PROGRAMAÇÃO E TENDÊNCIAS ARQUITETURAIS	19
1.2	MODELOS DE MEMÓRIA: O DESAFIO DA VERIFICAÇÃO	21
1.3	A RELEVÂNCIA DA VERIFICAÇÃO DINÂMICA	22
1.4	JUSTIFICATIVA E ESCOPO DESTA DISSERTAÇÃO	24
1.5	CONTRIBUIÇÕES	25
1.5.1	Contribuições técnicas	25
1.5.2	Contribuições científicas	26
1.6	ORGANIZAÇÃO DESTA DISSERTAÇÃO	26
1.7	REPERCUSSÃO E IMPACTOS ESPERADOS	27
2	PROBLEMAS-ALVO	29
2.1	EXEMPLO ILUSTRATIVO	29
2.2	A OBSERVABILIDADE NA PLATAFORMA	31
2.3	FORMULAÇÃO DO PROBLEMA	33
3	TRABALHOS CORRELATOS	37
3.1	UM PANORAMA DA VERIFICAÇÃO DE MODELOS DE MEMÓRIA	37
3.2	TRABALHOS EM VERIFICAÇÃO DINÂMICA	38
3.2.1	Verificação <i>post-mortem</i>	40
3.2.2	Verificação <i>on-the-fly</i>	41
3.3	LIMITAÇÕES DOS <i>CHECKERS</i> DINÂMICOS	41
4	CONFIGURAÇÃO EXPERIMENTAL	43
4.1	GERAÇÃO DE CASOS DE TESTE	43
4.1.1	Formato dos programas paralelos	44
4.1.2	Tamanho dos casos de teste	44
4.1.3	Número de endereços compartilhados	45
4.2	ARQUITETURA SOB VERIFICAÇÃO	46
4.2.1	Simulação de erros de projeto	47
4.2.2	Migração de <i>snooping</i> para diretórios	48
4.2.3	Número de núcleos	48
4.3	IMPLEMENTAÇÃO DE CHECKERS	49
5	RESULTADOS EXPERIMENTAIS	55
5.1	SENSIBILIDADE AO TIPO DE ERRO	56
5.2	IMPACTO DO NÚMERO CRESCENTE DE OPERAÇÕES	56
5.3	IMPACTO DO NÚMERO CRESCENTE DE PROCESSADORES	60
5.4	IMPACTO DO NÚMERO CRESCENTE DE ENDEREÇOS	63

6	PROPOSTA DE GENERALIZAÇÃO PARA CHECKERS	
	ON-THE-FLY	67
6.1	MOTIVAÇÃO	67
6.2	EXEMPLO ILUSTRATIVO	67
6.3	ARQUITETURA DA PLATAFORMA	71
6.4	ESTRUTURA DO CHECKER E IDEIAS-BASE	73
6.5	ALGORITMOS	74
6.6	STATUS DE VALIDAÇÃO E PERSPECTIVAS	80
7	CONSIDERAÇÕES FINAIS	83
7.1	CONCLUSÕES	83
7.2	TRABALHO EM ANDAMENTO	83
7.3	TRABALHOS FUTUROS	84
7.4	RECONHECIMENTOS	84
	REFERÊNCIAS	87

1 INTRODUÇÃO

1.1 MODELO DE PROGRAMAÇÃO E TENDÊNCIAS ARQUITETURAIS

No contexto de multiprocessadores, é provável que a programação de propósitos gerais continue a requerer uma abstração de memória compartilhada mesmo em uma escala de centenas de processadores (MARTIN; HILL; SORIN, 2012) (DEVADAS, 2013). Tal modelo de memória, que faz parte do modelo de programação de multiprocessadores, deve combinar regras de consistência e requisitos de coerência, os quais devem ser suportados em *hardware*.

Essencialmente, um modelo de memória captura dois aspectos de um sistema de memória compartilhada: a relaxação da ordem de programa entre operações de leitura e escrita (para endereços distintos) e o grau de atomicidade das operações de escrita (para o mesmo endereço) (ADVE; GHARACHORLOO, 1996) (ARVIND; MAESSEN, 2006). O primeiro aspecto define as regras de consistência; o segundo, os requisitos de coerência. Apesar de operações de escrita terem efeitos em diversos pontos do subsistema de memória, alguns modelos de memória exigem que elas pareçam ocorrer de forma atômica para todos os processadores.

Em um programa sequencial, executado em um único processador, é simples definir o resultado esperado de uma operação de leitura da memória: ela deve fornecer o último valor escrito no endereço correspondente. Porém, na execução de um programa paralelo, processadores distintos executam operações independentemente e, possivelmente, acessam os mesmos endereços compartilhados de memória. Nesse caso, não é trivial determinar qual foi o último valor escrito em um endereço, pois a memória está distribuída em *caches* locais e memória global, que podem armazenar cópias do dado endereçado com valores distintos.

Assim, deve haver uma especificação, formulada pelos projetistas da interface *hardware-software*, que defina como e quando o valor de uma operação de escrita pode ser observado por uma operação de leitura.¹

Para ilustrar como um modelo de memória afeta a interface *hardware-software*, a Figura 1 mostra o ordenamento das operações de um programa paralelo (Figura 1a), quando submetido a processadores com modelos de memória diferentes: um que conserva a ordem de programa (Figura 1b), outro que a relaxa para operações referenciando endereços distintos (Figura 1c). Esse exemplo foi adaptado a partir de um congênere (GHARACHORLOO, 1995).

¹Embora alguns autores chamem esse modelo de *memory consistency model* (ADVE; GHARACHORLOO, 1996), há quem o denomine apenas de modelo de memória (DEVADAS, 2013), pois ele captura não somente as regras de consistência, mas também os requisitos de coerência.

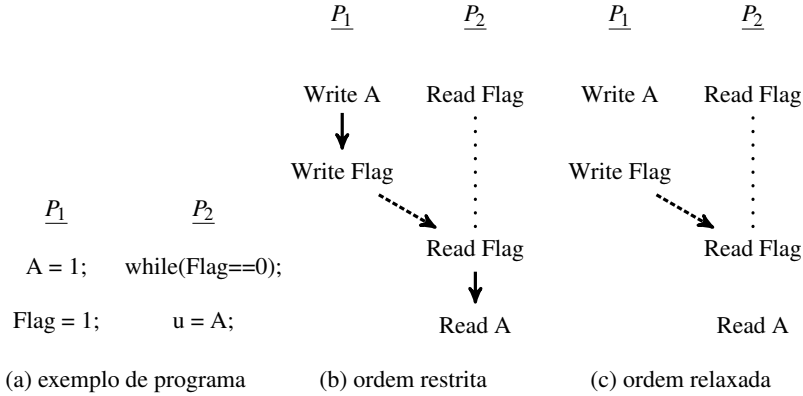


Figura 1: Exemplo ilustrativo para diferentes modelos de memória

O programa referencia duas variáveis compartilhadas (A e $Flag$) alocadas em memória e uma variável privativa (u), alocada em registrador. A ordem de programa é ilustrada pelas arestas contínuas. Note que a operação que escreve em $Flag$ e a operação que a lê não são ordenadas pelo programa, mas sua ordem é determinada por uma dada execução do programa. No exemplo, supõe-se o cenário em que o valor “1” escrito em $Flag$ tenha sido finalmente observado após as várias operações de leitura induzidas pelo laço. Nesse cenário, a precedência entre essas operações é ilustrada pela seta pontilhada.

Quando executado sob um modelo de memória restrito (Figura 1b), o programa resultaria em um comportamento “natural” para um programador: a leitura de um valor “1” em $Flag$ é um indício de que o valor “1” foi atribuído a A na primeira *thread*, sendo portanto observado na segunda *thread* e atribuído à variável u . Ou seja, é proibida a observação de um resultado $u == 0$ (valor inicial) para um modelo de memória restrito. Note que esse comportamento “natural” para o programador pressupõe que a ordem de programa se conserve entre operações de memória para endereços distintos. Ora, conservar a ordem para endereços diferentes tende a degradar o desempenho de programas em processadores que admitam execução fora de ordem.

A adoção de um modelo relaxado evita que essa degradação de desempenho impacte todo o programa, mas torna menos amigável o modelo de programação (Figura 1c). Como a ordem de programa é revogada na especificação do modelo, o programador deveria notar que há dois valores permitidos ($u == 0$ ou $u == 1$) sob um modelo relaxado. Para resolver essa ambiguidade, o programador tem que introduzir instruções que restituam a

ordem de programa,² mas apenas no caminho que quiser definir (como o das setas na Figura 1b). Assim, operações entre endereços distintos em outras partes do programa não sofreriam degradação de desempenho.

Obviamente, quando uma operação precede outra em uma *thread* do programa, sua reordenação é possível se elas se referem a endereços distintos e não houver uma dependência de dados via registrador. Entretanto, inicialmente, o modelo de consistência sequencial (*Sequential Consistency - SC*) (LAMPORT, 1979) conservava integralmente a ordem de programa para manter simples o modelo de programação. Mais tarde, algumas arquiteturas permitiram que operações de leitura passassem à frente de operações de escrita, levando a diversos modelos de memória: *Total Store Ordering* (TSO) (SPARC International Inc., 1992), IBM 370 (IBM, 1983), *Processor Consistency* (PC) (GHARACHORLOO et al., 1990) (GHARACHORLOO; GUPTA; HENNESSY, 1993), Intel 64 (Intel Corporation, 2014), Godson-3 (CHEN et al., 2009)³, etc. Outros também permitiam o reordenamento entre operações de escrita, como *Partial Store Ordering* (PSO) (SINDHU; FRAILONG; CEKLEOV, 1992) (SPARC International Inc., 1992).

Finalmente, algumas arquiteturas relaxaram todas as ordens, exceto para operações conflitantes (operações para o mesmo endereço tal que pelo menos uma seja uma escrita) da mesma *thread*, levando a modelos de memória como *Weak Ordering* (WO) (DUBOIS; SCHEURICH; BRIGGS, 1986) e suas variações (ADVE; GHARACHORLOO, 1996). A alta relaxação da ordem de programa é suportada, por exemplo, pelos processadores Alpha (SITES, 1992), PowerPC (MATEOSIAN, 1994), ARMv7 (ARM, 2012) e ARMv8 (ARM, 2013a).

Como é esperado que a maioria dos programas contem com bibliotecas padrão para sincronização, a maioria dos programadores não deve mais ter que se preocupar com regras de consistência, i.e. modelos de memória com alta relaxação da ordem de programa tendem a ser preferidos por seu maior desempenho (HENNESSY; PATTERSON, 2011).

1.2 MODELOS DE MEMÓRIA: O DESAFIO DA VERIFICAÇÃO

Para prover ao programador uma visão coerente da memória e, ao mesmo tempo, suportar a relaxação da ordem de programa, sistemas de memória tornam-se mais sofisticados e suscetíveis a erros de projeto conforme o número de processadores aumenta. Além disso, cobrir completamente o

²Barreiras de memória e pares *load-acquire/store-release* são exemplos de instruções dessa natureza (ADVE; GHARACHORLOO, 1996).

³Este modelo também permite reordenamento entre operações de leitura.

espaço de estados é intratável, pois ele explode combinatoriamente com o crescimento do número de processadores (SHIM et al., 2013). Como resultado, o problema de verificar se o projeto obedece a um dado modelo de memória torna-se cada vez mais desafiador.

Para garantir que o escopo de verificação não cresça com o número de processadores, alguns autores recomendam a coerência sem o uso de diretórios (SHIM et al., 2013) (FENSCH; CINTRA, 2008). Entretanto, uma simplificação tão radical pode abrir mão, desnecessariamente, de todo um legado de *software*. Por outro lado, outros afirmam que protocolos de coerência estado-da-arte baseados em diretório podem ser escaláveis a centenas de processadores (MARTIN; HILL; SORIN, 2012), com a vantagem de permitir retrocompatibilidade de software (e.g. sistemas operacionais). Esse segundo cenário, que parece ser mais pragmático e realista, leva, entretanto, a um maior desafio de verificação, que requer a combinação de métodos formais e verificadores dinâmicos (ABTS; SCOTT; LILJA, 2003).

A maior parte dos verificadores de modelos de memória dinâmicos reportados (deste ponto em diante, referenciados simplesmente como *checkers*) foi originalmente construído para a observabilidade limitada que restringe a verificação pós-silício (MANOVIT; HANGAL, 2006) (ROY; ZEISSET, 2006) (CHEN et al., 2009) (HU et al., 2012). Como resultado, quando reusados em tempo de projeto, eles se tornam ineficientes e ineficazes com o aumento do número de processadores.

Apesar de ser esperado que arquiteturas suportem modelos de memória com fraca atomicidade de escrita (HENNESSY; PATTERSON, 2011) (ARM, 2013a), todos os *checkers* dinâmicos reportados até então se baseiam implicitamente na forte atomicidade de escrita (i.e. escrita de todas as cópias da memória no mesmo instante), uma suposição não realista diante de múltiplos níveis hierárquicos de consulta (GHARACHORLOO, 1995), distâncias intra-chip crescentes (SCHUCHHARDT et al., 2013) e um tráfego de coerência expressivo (KRISHNA et al., 2013).

1.3 A RELEVÂNCIA DA VERIFICAÇÃO DINÂMICA

Por um lado, o crescimento do número de núcleos em um único circuito integrado (*chip multiprocessing* – CMP) pode permitir o projeto de núcleos mais simples desde que o paralelismo em nível de *threads* compense o menor suporte ao paralelismo em nível de instrução. Por outro lado, o projeto do subsistema de memória torna-se mais suscetível a erros, pois aumenta a dificuldade em satisfazer os requisitos de coerência e consistência da memória compartilhada.

Como a execução de um programa em uma representação executável pode ser diversas ordens de magnitude maior que no *hardware* real, grande parte dos *checkers* desenvolvidos em ambientes industriais (MANOVIT; HANGAL, 2006)(HU et al., 2012)(DEORIO; WAGNER; BERTACCO, 2009) foram originalmente projetados para a verificação pós-silício, onde o esforço de geração de *traces* (sequências de operações de memória executadas pela plataforma) é reduzido por sua execução direta no próprio *hardware* sob verificação. Tais *checkers* são variações de uma mesma classe: todos são baseados em regras de inferência que procuram determinar a ordem das operações de memória a partir dos *traces*.

O problema de verificação pós-silício é intratável (GIBBONS; KORACH, 1997) mesmo quando aplicado aos *traces* de execução de um único programa paralelo. Além disso, são necessários diversos programas paralelos para efetivamente expôr um erro no projeto do subsistema de memória, o que também contribui para o esforço de verificação.

Entretanto, por pragmatismo, os *checkers* desenvolvidos para verificação pós-silício são frequentemente reusados na etapa pré-silício. Como a geração de *traces* em representações executáveis requer mais tempo, esse reuso acaba reduzindo a cobertura que se consegue alcançar em função dos limites de tempo de projeto. Por conta disso, alguns autores afirmam (DEORIO; WAGNER; BERTACCO, 2009) que a verificação pós-silício é crucial para compensar a limitada cobertura resultante da verificação pré-silício. Contudo, deve ser notado que é justamente o reuso de *checkers* pós-silício para verificação pré-silício que acaba reforçando o próprio papel da verificação pós-silício! Em outras palavras, há provavelmente uma supervalorização do papel da verificação pós-silício, resultado do seu reuso inadequado.

Essa inadequação resulta do fato de os *checkers* pós-silício terem sido projetados para lidar com a observabilidade limitada de plataformas de *hardware* real: um único ponto de cada núcleo (sua interface com a memória compartilhada) é amostrado para determinar um *trace*.

A exploração da maior observabilidade de representações de projeto leva a instâncias muito mais simples do problema de verificação (RAMBO et al., 2012)(FREITAS et al., 2013). Apesar de resultados preliminares terem mostrado a superioridade de *checkers* pré-silício (RAMBO et al., 2012)(FREITAS et al., 2013) quando comparados a um *checker* baseado em inferência (HANGAL et al., 2004), eles merecem um esforço de qualificação mais amplo em relação a outros *checkers* estado-da-arte: não apenas *checkers* baseados em inferência mais recentes (HU et al., 2012)(MANOVIT; HANGAL, 2006), mas também em relação ao *checker on-the-fly*⁴ reportado em (SHACHAM et al., 2008).

⁴ *Checkers on-the-fly* realizam a análise simultaneamente à execução do programa.

A maioria dos *checkers* relatados na literatura não são comparados uns com os outros. A principal razão dessa falta de comparação parece ser os diferentes modelos-alvos de memória. Outra dificuldade para comparar diretamente os *checkers* é a diversidade dos erros reportados (HANGAL et al., 2004)(CHEN et al., 2009)(HU et al., 2012)(DEORIO; WAGNER; BERTACCO, 2009)(SHACHAM et al., 2008).

1.4 JUSTIFICATIVA E ESCOPO DESTA DISSERTAÇÃO

Diante da abundância de *checkers* pós-silício e da aparente inadequação de seu reuso para verificação em tempo de projeto, esta dissertação se concentra na verificação pré-silício. Diante de evidências de que a aplicação de técnicas puramente formais não é suficiente para atacar o desafio de verificação (ABTS; SCOTT; LILJA, 2003) em face das tendências esperadas em termos de arquitetura e de modelo de programação (HENNESSY; PATTERSON, 2011), assume-se que a verificação pré-silício terá de contar com uma combinação de métodos formais e dinâmicos para atender aos desafios de verificar não somente modelos de memória com relaxação da ordem de programa, mas também modelos com relaxação da atomicidade de escrita.

Nesse contexto, o foco desta dissertação é o de avaliar a eficiência e a eficácia de *checkers* dinâmicos pré-silício para modelos de memória relaxados. Para essa avaliação, foram comparados cinco *checkers* dinâmicos (três deles originalmente desenvolvidos para verificação pós-silício e dois para verificação pré-silício) para um modelo de memória com *máxima relaxação da ordem de programa*, mas que considera as operações de escrita como (fortemente) atômicas.

Para os experimentos, diversos erros relatados na literatura (alguns deles sendo *bugs* reais encontrados em plataformas industriais) foram individualmente inseridos em representações de projeto pré-validadas.

A avaliação permitiu identificar que todos os *checkers* reportados na literatura até o momento limitam-se a modelos que assumem forte atomicidade de escrita. Diante de modelos onde essa atomicidade é revogada, como por exemplo o adotado na arquitetura ARMv8 recentemente lançada (ARM, 2013a), todos os *checkers* reportados poderão emitir *diagnósticos de falso positivo*, ou seja, erros aparentes seriam indevidamente sinalizados, tornando a verificação mais trabalhosa. Como um desdobramento dessa constatação, o escopo da dissertação foi ampliado para conter uma proposta de algoritmo de verificação generalizado capaz de garantir uma *abordagem adequada para modelos sob fraca atomicidade de escrita*.

Ora, para abordar adequadamente a originalidade dessa proposta, é ne-

cessário um esforço de pesquisa não compatível com os prazos de um trabalho de mestrado, pois requer: a construção de uma especificação formal, para fins de verificação, de um modelo de memória com máxima relaxação de ordem de programa e fraca atomicidade de escrita, a prova de teoremas para garantir que o algoritmo generalizado proposto satisfaz a especificação formal (sem emitir diagnósticos de falso positivo ou falso negativo para um dado caso de teste), a implementação de um tal modelo de memória na plataforma experimental, a concepção e a simulação na plataforma de novos erros de projeto que possam afetar os protocolos de coerência sob fraca atomicidade e, finalmente, a execução de testes com o novo *checker* proposto quando comparado aos cinco já avaliados sob forte atomicidade de escrita. Por isso, esse esforço está sendo objeto de um trabalho de pesquisa complementar ao escopo original da dissertação, em colaboração com o orientador, a ser submetido para publicação em periódico. Por essa razão, esse trabalho paralelo não está reportado como parte do escopo do texto dessa dissertação, exceto pela *ideia-base e pela proposta do algoritmo generalizado completo* (que são reportados no Capítulo 6, mas sem apresentar garantias teóricas e experimentais).

1.5 CONTRIBUIÇÕES

1.5.1 Contribuições técnicas

Vários artefatos, construídos durante o trabalho de pesquisa desta dissertação, têm o potencial de viabilizar trabalhos de pesquisa futuros. Dentre eles, os principais são:

- **Representação executável para protocolos de coerência baseados em diretório:** A infraestrutura experimental foi adaptada para permitir a migração dos testes do protocolo de *snooping* para diretórios, conforme descrito na Seção 4.2.2.⁵
- **Implementação de *checkers* não disponíveis em domínio público:** Diversos *checkers* não disponíveis em domínio público foram implementados, conforme a descrição de seus algoritmos por seus respectivos autores. Tais *checkers* estão listados, juntamente com as descrições de seus detalhes de implementação, na Seção 4.3.
- **Definição de um conjunto de erros de projeto para fins experimen-**

⁵Essa adaptação não foi trivial. Apesar de a plataforma possuir suporte, ele não era satisfatório para os fins de avaliação dos *checkers* e foram feitas adaptações durante cerca de dois meses até se chegar à representação executável desejada.

tais: Para avaliar a eficácia e eficiência dos *checkers* na detecção de erros de projeto, vários erros tiveram que ser inseridos na plataforma de simulação. Eles são descritos na Seção 4.2.1.

1.5.2 Contribuições científicas

As principais contribuições científicas desta dissertação são:

- **A mais ampla comparação experimental de *checkers* pré-silício já reportada:** *Checkers* baseados em quatro classes diferentes de algoritmos foram submetidos a 9000 casos de uso, sob o mesmo modelo de memória e as mesmas condições experimentais (HENSCHTEL; SANTOS, 2013).
- **A identificação de deficiência (não reportada na literatura) para *checkers* dinâmicos:** Sabe-se da literatura que *checkers* baseados em inferências sobre *traces* de memória podem resultar em falsos negativos (HANGAL et al., 2004), a menos que recorram ao uso de *backtracking* (MANOVIT; HANGAL, 2006). Sabe-se também que *checkers* baseados em regras (SHACHAM et al., 2008) podem levar a falsos positivos. Entretanto, descobriu-se ao avaliar os *checkers*, que nenhum deles é capaz de evitar falsos positivos para modelos de memória com fraca atomicidade de escrita, tal como o das arquiteturas ARMv7 (ARM, 2012) e ARMv8 (ARM, 2013a).
- **A proposta de algoritmos cooperantes capazes de tratar adequadamente modelos com fraca atomicidade de escrita:** Embora o projeto completo dos algoritmos cooperantes tenha sido um trabalho interativo desenvolvido em colaboração com o orientador, tenha se apoiado em trabalhos anteriores para generalização e tenha sido parcialmente induzido pelas provas teóricas de garantias de verificação, um dos algoritmos propostos (denominado *coherent-order*), fundamental para suportar a fraca atomicidade de escrita, foi inteiramente desenvolvido pelo autor desta dissertação.

1.6 ORGANIZAÇÃO DESTA DISSERTAÇÃO

O restante desta dissertação é organizado da seguinte maneira. O Capítulo 2 descreve os problemas resolvidos pelos *checkers* a serem avaliados. O Capítulo 3 faz uma revisão crítica da literatura. O Capítulo 4 mostra as

configurações das plataformas simuladas e dos casos de teste gerados. O Capítulo 5 faz uma análise dos resultados obtidos com os experimentos, comparando os diversos *checkers* sob as mesmas condições. O Capítulo 6 explica porque e como construir um *checker on-the-fly* que trate apropriadamente uma importante instância do problema de verificação: aquela induzida por modelos de memória com fraca atomicidade de escrita. O Capítulo 7 resume as conclusões, reporta um trabalho em andamento e propõe trabalhos futuros, além de reconhecer o papel daqueles que contribuíram, direta ou indiretamente, para a viabilidade deste trabalho.

1.7 REPERCUSSÃO E IMPACTOS ESPERADOS

Os resultados experimentais e teóricos de uma das técnicas comparadas no Capítulo 4 tiveram seu mérito reconhecido pelo comitê de programa do evento *Design, Automation & Test in Europe (DATE 2012)*, resultando na publicação de trabalho intitulado “On ESL verification of memory consistency for system-on-chip multiprocessing” nos anais daquele evento (RAMBO et al., 2012). O mecanismo de geração de casos de teste foi publicado nos anais do evento *IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2011)*, em trabalho intitulado “Automatic generation of memory consistency tests for chip multiprocessing”.⁶

Análises comparativas de algumas das técnicas mencionadas no Capítulo 4 tiveram seu mérito reconhecido pelo comitê de programa da *IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2013)*, resultando na publicação de trabalho intitulado “Pre-silicon verification of multiprocessor SoCs: The case for on-the-fly coherence/consistency checking” nos anais daquele evento (HENSCHTEL; SANTOS, 2013).

A implementação dos algoritmos propostos para o *checker* generalizado descrito no Capítulo 6 está em curso, assim como as provas teóricas das garantias de verificação. Eles serão submetidos na forma de um artigo, em co-autoria com o orientador, para o periódico *IEEE Transactions on Computers (TC)*. Espera-se que o *checker* resolva a deficiência dos verificadores dinâmicos identificada nesta dissertação. Espera-se também que a especificação formal de um modelo de memória sob fraca atomicidade de escrita, além de prover suporte para as garantias técnicas dos algoritmos aqui propostos, também auxilie no desenvolvimento de futuras técnicas.

⁶O autor desta dissertação participou desses dois trabalhos como co-autor.

2 PROBLEMAS-ALVO

Antes de formalizar os problemas abordados pelos *checkers* dinâmicos, este capítulo ilustra, através de um exemplo, as principais noções usadas em sua formulação e resolução.

Em seguida, determina-se a observabilidade da plataforma sob verificação, que distingue os diferentes problemas.

Finalmente, define-se formalmente os problemas-alvo independentemente do modelo de memória, isto é, diferentes instâncias do mesmo problema podem ser induzidas para modelos de memória distintos.

2.1 EXEMPLO ILUSTRATIVO

O exemplo a seguir utiliza grafos direcionados, por serem uma representação usual para relações de ordem entre operações de memória. Para denotar operações de leitura ou escrita referenciando um endereço a , escreve-se $(L_j)_a$ e $(S_j)_a$, respectivamente. Assume-se um modelo de memória com máxima relaxação de ordem de programa.

A Figura 2 mostra um programa paralelo (não-sincronizado) simples, um conjunto de *traces* de eventos de memória observados como resultado de uma execução e duas possíveis execuções correspondendo ao mesmo conjunto de *traces*. Os distintos ordenamentos de operações na Figura 2 são ilustrados (como arestas pretas) na Figura 3, que também mostra como sua combinação (arestas cinzas) pode ser usada para a análise de compatibilidade com o modelo de memória.

A Figura 2a mostra que o programa consiste em duas *threads* (T_1 e T_2). A Figura 3a captura restrições de ordem de programa impostas por cada *thread*.

A Figura 2b mostra dois *traces* de eventos de memória, cada qual observado na interface entre cada processador (P_1 e P_2) e o sistema de memória. Cada evento é denotado por uma tripla que define se o evento ocorrido é de leitura (R) ou escrita (W), qual foi o endereço referenciado (a) e qual foi o valor lido da ou escrito na memória. A Figura 3b ilustra alguns requisitos de ordem inferidos dos *traces*. Como ambos L_3 e L_4 observam o valor escrito por S_1 , infere-se que (S_1, L_3) e (S_1, L_4) valem. Além disso, pela restrição de ordem de programa, (S_1, S_2) vale, mas como nem L_3 nem L_4 observam o valor escrito por S_2 , também se infere que ambos (L_3, S_2) e (L_4, S_2) valem.

Visto que, da análise dos *traces* observados, nenhum ciclo é detectado,¹

¹ Somente um grafo direcionado acíclico pode representar uma ordem parcial; um ciclo violaria

T_1	T_2
$(S_1)_a$	$(L_3)_a$
$(S_2)_a$	$(L_4)_a$

(a) programa

P_1	P_2
$S_1 = (W, a, 1)$	$L_3 = (R, a, 1)$
$S_2 = (W, a, 2)$	$L_4 = (R, a, 1)$

(b) *traces*

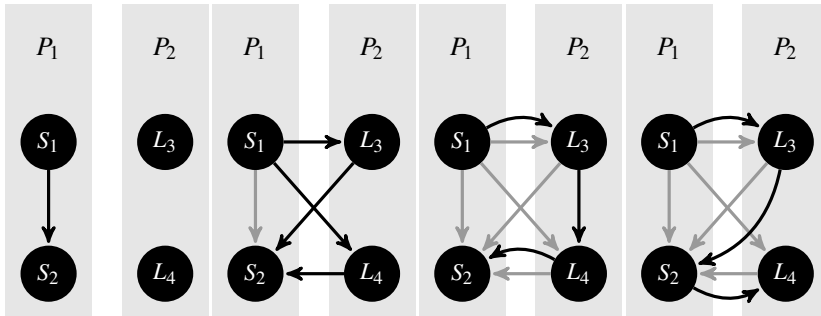
time	P_1	P_2
1	$S_1 = (W, a, 1)$	
2		$L_3 = (R, a, 1)$
3		$L_4 = (R, a, 1)$
4	$S_2 = (W, a, 2)$	

(c) um comportamento

time	P_1	P_2
1	$S_1 = (W, a, 1)$	
2		$L_3 = (R, a, 1)$
3	$S_2 = (W, a, 2)$	
4		$L_4 = (R, a, 1)$

(d) outro comportamento

Figura 2: Um programa, seus *traces* e dois comportamentos atômicos



(a) ordem de programa

(b) ordem inferida

(c) uma ordem temporal válida

(d) uma ordem temporal inválida

Figura 3: Ordenamento de eventos sob forte atomicidade de escrita

um *checker baseado em traces* concluiria que existe pelo menos uma ordem total satisfazendo a ordem parcial na Figura 3b. Como nenhum comportamento incorreto pôde ser inferido dos *traces*, tal *checker* não indicaria um erro de projeto.

As Figuras 2c e 2d mostram comportamentos alternativos da memória que produziram os mesmos *traces* da Figura 2b. Do ponto de vista de um *checker* baseado em *traces*, eles são indistinguíveis. Assumiremos agora que um *checker baseado em comportamentos* pode observar e analisar cada um deles individualmente.

A Figura 3c mostra a ordem linear de eventos correspondente às marcas temporais da Figura 2c. Note que nenhum ciclo se forma.

Em contraste com a Figura 2c, para o comportamento na Figura 2d, um ciclo (S_2, L_4, S_2) é formado, como mostrado na Figura 3d. Portanto, ao analisar modelos de memória com forte atomicidade de escrita, um *checker* baseado em comportamentos indicaria corretamente um erro de projeto.²

Embora imune a falsos positivos, *checkers* baseados em *traces* não representam a solução definitiva para tratar modelos relaxados em tempo de projeto, como o exemplo pode sugerir. Esse simples exemplo não ilustra o fato que, em casos mais gerais, *checkers* baseados em *traces* necessitam de *backtracking* para evitar *falsos negativos* (MANOVIT; HANGAL, 2006), que ocorreriam quando um *trace* aparentemente correto (i.e. sem ciclos inferidos) resulta apenas em comportamentos incorretos. Infelizmente, o *backtracking* resulta em grandes tempos de análise, que são somados aos já altos tempos de simulação de uma representação de projeto. Ademais, o *backtracking* limita a escalabilidade a longo prazo de *checkers* baseados em *traces* para um crescente número de processadores.³

2.2 A OBSERVABILIDADE NA PLATAFORMA

A Figura 4 mostra um diagrama esquemático genérico para um multi-processador com memória compartilhada. Ele ilustra que a arquitetura consiste em p elementos de processamento $(P_1, \dots, P_i, \dots, P_p)$, cada qual acessando dados através de uma *cache* privativa. Assume-se que um *buffer de reordenamento* (*reorder buffer* - ROB) (HENNESSY; PATTERSON, 2011) ou uma estrutura similar é usada para garantir a consolidação das instruções em ordem

a propriedade de anti-simetria dessa relação.

²Contudo, sob fraca atomicidade de escrita, isso poderia ser um comportamento perfeitamente correto e o erro apontado seria um *falso positivo*, como veremos no Capítulo 6.

³É por esses motivos que *checkers* baseados em comportamentos, os quais não dependem de *backtracking*, devem ser investigados como uma alternativa para atacar a fraca atomicidade de escrita com o aumento do número de processadores, como veremos no Capítulo 6.

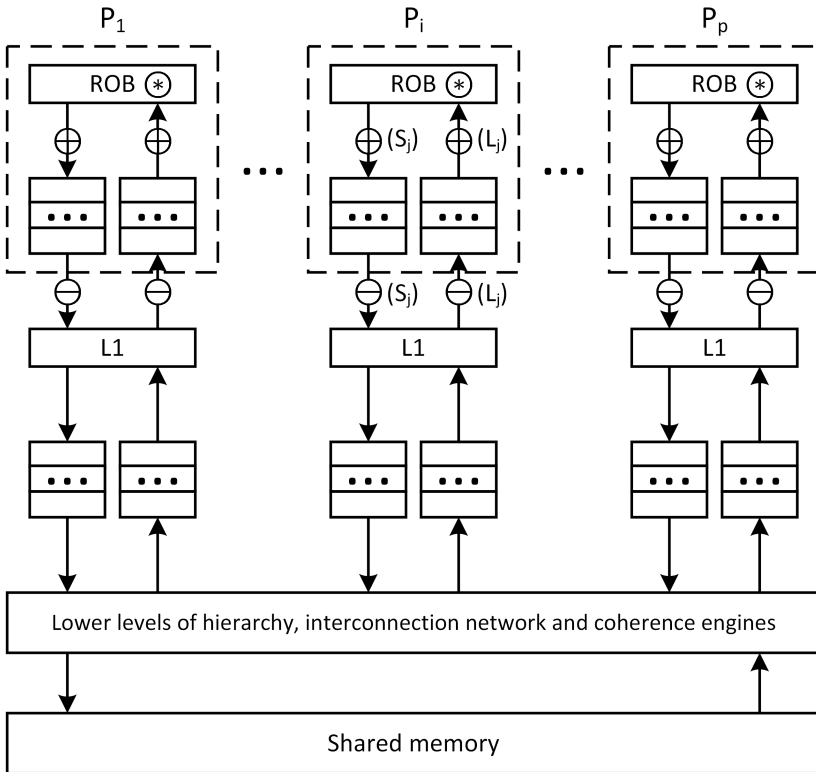


Figura 4: Um modelo genérico de sistema com memória compartilhada

de programa. *Buffers* de entrada e saída são destacados nas fronteiras da *cache* de dados privativa. Como os *checkers* devem ser amplamente independentes da organização da memória, o diagrama abstrai os níveis mais baixos da hierarquia de *caches*, a rede de interconexão e os mecanismos de coerência.

Pontos relevantes onde os eventos de memória podem ser observados são indicados pelos símbolos \oplus , \ominus e \otimes . Cada um desses pontos é chamado de *monitor*.

Usemos o esquemático para discutir a observabilidade de *checkers* distintos. Devido às limitações de observabilidade intra-chip, *checkers* pós-silício (MANOVIT; HANGAL, 2006) (HU et al., 2012) capturam um único *trace* por elemento de processamento, amostrando eventos como se eles fossem observados em cada monitor \ominus .⁴ Por outro lado, *checkers* pré-silício podem se beneficiar da maior observabilidade de representações de projeto para resolver instâncias mais simples do problema de verificação (como será mostrado na próxima seção). Por exemplo, dois *checkers* pré-silício (RAMBO et al., 2012) (FREITAS et al., 2013) amostram duas sequências de eventos por elemento de processamento, amostrando eventos nos monitores \oplus e \ominus . Em (FREITAS et al., 2013), o ROB é também observado por um terceiro monitor \otimes .

2.3 FORMULAÇÃO DO PROBLEMA

Esta seção formula instâncias relevantes do problema de verificação alvo. Daqui em diante, p e n denotarão, respectivamente, o número de processadores e o número total de operações de memória.

Primeiramente, deve ser formalizado o principal conceito na verificação de modelos de memória:

Definição 1 *Um trace é uma sequência $(\tau_1, \tau_2, \dots, \tau_j, \dots, \tau_m)$, onde $\tau_j = (op, a, v)$ é um evento de memória tal que $op \in \{R, W\}$, a é um endereço e v é um valor.*

Traces locais $T_{\ominus}^1, T_{\ominus}^2, \dots, T_{\ominus}^p$ são obtidos amostrando eventos em cada processador, conforme indicado pelos monitores \ominus na Figura 4. Eles podem ser combinados em um *trace* global T entrelaçando os *traces* locais enquanto mantém-se a ordem de amostragem local, i.e. se τ_j precede τ_k em algum *trace* T_{\ominus}^x , com $x = 1, \dots, p$, então τ_j deve preceder τ_k em T .

⁴A popularização do multiprocessamento em chip (*Chip Multiprocessing* - CMP) e sistemas embarcados multiprocessados (*System-on-chip* - SoC) criou a necessidade de suporte para instrumentar essa observabilidade. Por exemplo, (CHEN et al., 2009) insere instruções de leitura no programa para armazenar *traces* na RAM interna, que são então transferidos externamente, através de um mecanismo de acesso direto à memória.

Quando a observabilidade é restringida à amostragem de *traces* nos monitores \ominus , o problema de verificação pode ser formulado da seguinte forma:

Problema 1 Verificação baseada em traces: *Dada uma coleção de traces $T_{\ominus}^1, T_{\ominus}^2, \dots, T_{\ominus}^p$, existe um trace global T que satisfaça todas as restrições do modelo de memória?*

Como o problema geral é NP-completo (GIBBONS; KORACH, 1997), a maior parte dos *checkers* pós-silício (e.g. (MANOVIT; HANGAL, 2006) (CHEN et al., 2009) (HU et al., 2012)) aborda instâncias do Problema 1 que resultam de informações adicionais, tais como a ordem total de escritas (ordenamento de operações de escrita para um mesmo endereço) e mapeamento de leituras (ligando cada operação de leitura com a operação de escrita que produziu seu valor). O problema torna-se polinomial quando ambas as informações estão disponíveis (GIBBONS; KORACH, 1997).

Assuma que, em tempo de execução ou simulação, o instante em que um evento é amostrado pode ser determinado, além de sua operação, valor e endereço. Isso leva ao seguinte conceito:

Definição 2 *Um comportamento é uma sequência $(\beta_1, \beta_2, \dots, \beta_j, \dots, \beta_m)$, onde $\beta_j = (op, a, v, t)$ é um evento de memória com marca temporal tal que $op \in \{R, W\}$, a é um endereço, v é um valor e t é o tempo em que o evento foi observado. A marca temporal de algum evento β_j será denotada como $t(\beta_j)$.*

De forma similar aos *traces*, comportamentos locais $B_{\ominus}^1, B_{\ominus}^2, \dots, B_{\ominus}^p$ são obtidos amostrando eventos em cada processador, conforme indicado pelos monitores \ominus na Figura 4. Eles podem ser combinados em um comportamento global B entrelaçando os comportamentos locais enquanto mantém-se a ordem de marcas temporais, i.e. se $t(\beta_j) < t(\beta_k)$, então β_j deve preceder β_k em B .

Isso leva a outro problema de verificação:

Problema 2 Verificação baseada em comportamentos: *Dada uma coleção de comportamentos $B_{\ominus}^1, B_{\ominus}^2, \dots, B_{\ominus}^p$, existe um comportamento global B que satisfaça todas as restrições do modelo de memória?*

Como os *checkers* pré-silício se beneficiam da observabilidade adicional de representações de projeto e ambientes de simulação, eles podem resolver instâncias mais simples desse problema, conforme abaixo:

Problema 3 Verificação aumentada baseada em comportamentos: *Dada uma coleção de comportamentos $B_{\ominus}^1, B_{\ominus}^2, \dots, B_{\ominus}^p, B_{\oplus}^1, B_{\oplus}^2, \dots, B_{\oplus}^p, \dots$, existe um comportamento global B que satisfaça todas as restrições do modelo de memória?*

Quando comportamentos locais são amostrados por pelo menos dois monitores (\oplus e \ominus), a instância resultante do Problema 3 pode ser resolvida em tempo polinomial e com garantias totais de verificação sob forte atomicidade de escrita (RAMBO et al., 2012) (FREITAS et al., 2013).⁵

⁵Exceto para modelos de memória que requerem uma ordem total de escritas.

3 TRABALHOS CORRELATOS

Este capítulo primeiramente situa o escopo deste trabalho de pesquisa no contexto de verificação de memória compartilhada e então enfoca em trabalhos mais diretamente relacionados com a avaliação aqui pretendida. Finalmente, o capítulo discute as limitações dos *checkers* dinâmicos apresentados na literatura.

3.1 UM PANORAMA DA VERIFICAÇÃO DE MODELOS DE MEMÓRIA

Idealmente, o *problema de corretude* (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002) a seguir deveria ser tratado: dada uma especificação de um modelo de memória e uma implementação de um sistema multiprocessado, verifique se *todas* as execuções geradas por esse sistema satisfazem o modelo para *qualquer* programa paralelo.

Por um lado, abordagens formais (HENZINGER; QADEER; RAJAMANI, 1999) (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002) (ABTS; SCOTT; LILJA, 2003) tratam instâncias simplificadas desse problema, verificando a corretude de uma abstração da real implementação à frente da especificação do modelo de memória. Apesar de tais abordagens encontrarem erros nos estágios iniciais do projetos (e.g. erros de protocolo), eles deixam escapar erros presentes apenas na implementação.

Por outro lado, métodos dinâmicos podem verificar execuções que exercitam todos os detalhes do sistema: o *hardware* real (HANGAL et al., 2004) (ROY; ZEISSET, 2006) (MANOVIT; HANGAL, 2005) (MANOVIT; HANGAL, 2006) (CHEN et al., 2009) (HU et al., 2012), um protótipo (LENOSKI et al., 1990) ou sua representação executável (SHACHAM et al., 2008) (RAMBO et al., 2012) (FREITAS et al., 2013). Entretanto, eles resolvem instâncias do problema de corretude que são limitadas a *uma* execução de *um* dado programa paralelo (i.e. um caso de teste), apesar de que diversas instâncias são resolvidas para melhorar as chances de expôr erros (i.e. um conjunto de casos de teste).

As características complementares e as desvantagens inerentes às abordagens formais e aos métodos dinâmicos motivam sua combinação (ABTS; SCOTT; LILJA, 2003) como estratégia para reduzir o esforço total de verificação.

A maioria dos *checkers* está atrelado a uma certa fase de projeto. Alguns dos *checkers* projetados para o uso pós-silício necessitam de modificações no *hardware* do subsistema de memória. Por exemplo, Dacota (DEORIO;

WAGNER; BERTACCO, 2009) modifica a hierarquia de memória e a interconexão para observar o mapeamento entre leituras e escritas e a ordem total de escritas, o que permite a verificação em tempo linear. Entretanto, alguns projetos industriais podem não prever o uso dedicado de *hardware* para a verificação do modelo de memória. É por isso que diversos *checkers* pós-silício são inteiramente baseados em *software* (HANGAL et al., 2004) (ROY; ZEISSET, 2006) (MANOVIT; HANGAL, 2005) (MANOVIT; HANGAL, 2006) (HU et al., 2012) (exceto talvez pela instrumentação trivial do *hardware*, como em (CHEN et al., 2009)).

Para a verificação pré-silício, a literatura reporta técnicas formais (HENZINGER; QADEER; RAJAMANI, 1999) (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002), dinâmicas (SHACHAM et al., 2008) (RAMBO et al., 2012) (FREITAS et al., 2013) e híbridas (ABTS; SCOTT; LILJA, 2003). Além disso, *checkers* baseados em *software* criados para a verificação pós-silício são frequentemente reusados na verificação pré-silício.

3.2 TRABALHOS EM VERIFICAÇÃO DINÂMICA

Como o foco desta dissertação é na verificação pré-silício com *checkers* dinâmicos, abordagens formais e técnicas que exigem assistência do *hardware* estão fora do escopo. Como os *checkers* pós-silício que não exigem assistência do *hardware* são frequentemente reusados em etapas pré-silício, eles estão incluídos na avaliação. Portanto, de agora em diante, serão tratadas apenas as abordagens baseadas em *software* que resolvam instâncias dos Problemas 1, 2 e 3, as quais podem ser divididas em duas categorias: verificação *post-mortem* e verificação *on-the-fly*. As principais características de tais *checkers* estão descritas na Tabela 1.

Tabela 1: Principais características dos *checkers* dinâmicos

Referência	Análise	Ideia-chave	Garantias	Número de monitores	Complexidade de pior caso
(HANGAL et al., 2004)	<i>Post-mortem</i>	Inferência	Nenhuma	1	$O(n^5)$
(MANOVIT; HANGAL, 2005)	<i>Post-mortem</i>	Inferência	Nenhuma	1	$O(pn^3)$
(MANOVIT; HANGAL, 2006)	<i>Post-mortem</i>	Inferência	Plenas	1	$O((n/p)^p pn^3)$
(ROY; ZEISSET, 2006)	<i>Post-mortem</i>	Inferência	Nenhuma	1	$O(n^4)$
(SHACHAM et al., 2008)	<i>On-the-fly</i>	<i>Scoreboard</i>	Nenhuma	1	$O(p^2 n^2)$
(CHEN et al., 2009)	<i>Post-mortem</i>	Inferência	Nenhuma	1	$O(p^3 n)$
	<i>Post-mortem</i>	Inferência	Plenas	1	$O(C^p p^2 n^2)$
(HU et al., 2012)	<i>Post-mortem</i>	Inferência	Plenas	1	$O(C^p p^3 n)$
(RAMBO et al., 2012)	<i>Post-mortem</i>	Emparelhamento de grafos bipartidos	Plenas	2	$O(n^6/p^5)$
(FREITAS et al., 2013)	<i>On-the-fly</i>	Múltiplas <i>scoreboards</i>	Plenas	3	$O(n^2/p)$

3.2.1 Verificação *post-mortem*

A maioria dos *checkers* dinâmicos requer que os *traces* sejam gerados previamente e baseia-se na detecção de ciclos em grafos orientados. Nesse tipo de técnica, os vértices representam operações de memória (escritas e leituras) e as arestas representam uma relação de ordem parcial. As ferramentas que operam dessa forma criam um grafo a partir das operações e tentam inferir o maior número possível de arestas, indicando um erro se um ciclo for detectado.¹ O tempo de análise acaba sendo limitado pelo fato de que nem todas as arestas podem ser inferidas, o que torna esse tipo de algoritmo incompleto, porque nem sempre encontra todos os erros existentes. TSOtool (HANGAL et al., 2004) é um exemplo de ferramenta de verificação incompleta que se utiliza desse método e que, posteriormente, foi estendida para se tornar completa (MANOVIT; HANGAL, 2006).

Para esse tipo de técnica se tornar completa, é necessário fazer *backtracking* para tentar-se inferir todas as possibilidades de arestas, o que torna sua complexidade exponencial. O XCHECK (HU et al., 2012), uma versão aprimorada do LCHECK (CHEN et al., 2009), usa como informações adicionais os intervalos temporais das operações, e, com isso, consegue fazer um *backtracking* que depende apenas do número de processadores, e não do número de operações, que é o principal limitador de desempenho dos *checkers* atuais. No entanto, com a crescente paralelização dos processadores, através do aumento do número de núcleos, essa técnica tende a se tornar cada vez mais ineficiente.

O Emparelhamento Estendido em Grafos Bipartidos (RAMBO et al., 2012) é outro método baseado em grafos, mas que funciona de uma forma totalmente diferente, sem a detecção de ciclos. Ele foi projetado especialmente para a verificação de representações executáveis de plataformas multiprocessadas. Devem existir dois monitores em cada núcleo de processamento: um na saída da unidade de consolidação de resultados (*commit unit*) e outro na interface com a memória privativa. Um grafo bipartido é construído a partir desses monitores, com seus vértices representando operações de memória e arestas representando equivalência. Outro grafo, com os mesmos vértices, representa o ordenamento de operações, imposta pelo modelo de memória, através de arestas. Com esses dois grafos é possível determinar se o comportamento local de cada processador está correto. Para verificar o comportamento global, são usados relógios de Lamport (LAMPOR, 1978), que utilizam marcas temporais geradas pelos monitores para cada operação. A partir deles,

¹Quando há um ciclo, existe um paradoxo, pois entende-se que as operações envolvidas devem ocorrer ao mesmo tempo antes e depois umas das outras (a assimetria da ordem parcial é violada).

é criado um registro de execução global, o qual é analisado por um algoritmo que determina se ele viola as regras do modelo de memória.

3.2.2 Verificação *on-the-fly*

Relaxed Scoreboard (SHACHAM et al., 2008) é uma técnica de verificação que possui um princípio de funcionamento diferente. Ela mantém uma tabela de possíveis valores para cada endereço de memória, a qual se adapta durante a execução do programa. Quando é feita uma escrita na memória, um novo valor é adicionado à tabela para o endereço da operação; quando ocorre uma leitura, os possíveis valores são filtrados, reduzindo o número de possibilidades. A principal vantagem em relação às técnicas baseadas em grafos é que, dessa forma, a ferramenta pode fazer a análise simultaneamente à execução do programa, e assim detectar erros mais rapidamente.

A técnica proposta em (FREITAS et al., 2013) foi, em parte, baseada na técnica de Emparelhamento Estendido em Grafos Bipartidos (RAMBO et al., 2012), pois usa o mesmo mecanismo de verificação global. Entretanto, a verificação local é totalmente diferente. Sua principal vantagem é também a possibilidade da verificação *on-the-fly* (sem a necessidade de esperar pelo término da execução do programa), graças à maior observabilidade fornecida por um terceiro monitor para cada núcleo.² Enquanto todas as operações de leitura e escrita são monitoradas na interface com a memória, um monitor na unidade de consolidação se encarrega de observar as operações que foram consolidadas e outro no *buffer* de reordenamento observa quais operações concluídas foram canceladas. Para cada processador, um *scoreboard* relaxado é atualizado a cada operação monitorada, verificando se cada operação que chega à memória tem uma correspondente observada em algum dos outros monitores e se não há inversão ilegal da ordem especificada pelo modelo de memória.

3.3 LIMITAÇÕES DOS *CHECKERS* DINÂMICOS

Baseando-se na análise da literatura, pode-se encontrar vários indícios de deficiências dos *checkers* dinâmicos reportados. Alguns desses indícios merecem uma avaliação experimental comparativa para serem apropriadamente elucidados, através de uma análise quantitativa de eficiência e eficácia (como

²O terceiro monitor permite distinguir leituras consolidadas de leituras especulativas, o que evita que estes últimos sejam indicados como erros, ou seja, diagnósticos de falsos positivos são descartados.

a apresentada no Capítulo 5). A seguir, resumem-se as principais deficiências dos *checkers* analisados neste capítulo.

Os *checkers* baseados em *traces* possuem alta complexidade. Os que fazem inferências não se adequam ao crescente número de processadores (MANOVIT; HANGAL, 2006) (HU et al., 2012). Um *checker* que usa emparelhamento estendido (RAMBO et al., 2012) adequa-se melhor ao crescente número de processadores, porém, tem uma complexidade relativamente alta em relação ao número de operações. Assim como os anteriores, precisa também aguardar a execução completa dos casos de teste para realizar a verificação (i.e. realiza análise *post-mortem*).

Os *checkers* que analisam um dado comportamento (ao invés de *traces*) são capazes de encontrar erros antes do término dos testes (i.e. realizam análise *on-the-fly*) e levam a baixos tempos de execução. Quando apenas um *scoreboard* relaxado (SHACHAM et al., 2008) é usado, não podem ser dadas garantias de verificação (i.e. falsos negativos ou falsos positivos podem ocorrer), o que limita consideravelmente a eficácia dos casos de teste analisados. Por fim, quando múltiplas *scoreboards* são utilizadas, pode-se dar *plenas garantias*³ mantendo-se a baixa complexidade (FREITAS et al., 2013).

Apesar da vantagem relativa de uns sobre os outros e embora não explicitado na literatura, todos esses *checkers* foram projetados para plataformas com forte atomicidade de escrita, podendo indicar falsos positivos quando aplicados a modelos de memória suportados em arquiteturas mais recentes, como a ARMv7 (ARM, 2012) e a ARMv8 (ARM, 2013a).

³Diz-se que um *checker* dinâmico provê plenas garantias de verificação se ele provavelmente não emite diagnóstico de falso negativo nem de falso positivo para os erros expostos por um dado caso de teste.

4 CONFIGURAÇÃO EXPERIMENTAL

A avaliação comparativa de *checkers* passíveis de utilização pré-silício foi realizada através de experimentos que caracterizam cenários realistas de projeto, levando em conta as arquiteturas atuais e sua evolução esperada nos próximos anos.

Os experimentos são constituídos de três partes fundamentais: a geração de casos de teste, a execução dos testes em uma plataforma de simulação de sistemas e a análise dos resultados da execução pelos diferentes *checkers* usados na avaliação.

4.1 GERAÇÃO DE CASOS DE TESTE

O gerador de casos de teste usado foi motivo de uma publicação anterior (RAMBO; HENSCHERL; SANTOS, 2011) e foi aprimorado desde então. Ele gera programas paralelos pseudo-aleatoriamente, aceitando os seguintes parâmetros: o número de *threads* criadas pelo programa, o número de instruções por *thread*, o número de endereços compartilhados pelas *threads* e a probabilidade de ocorrência de instruções de leitura (*load*), escrita (*store*) e de restauração da ordem de programa (barreiras de memória).

Para diminuir a correlação entre casos de teste e, ao mesmo tempo, permitir sua geração determinística, todos esses parâmetros são usados no cálculo de uma semente aleatória (*random seed*), que determina o tipo e a ordem das operações que são inseridas no programa. Um último parâmetro é um valor numérico usado no cálculo da semente aleatória, que possibilita a geração de programas diferentes com a mesma configuração.

Os programas paralelos gerados (deliberadamente) não são sincronizados, para forçar o surgimento de *data-races* e assim expôr possíveis erros de projeto, uma prática muito comum em ambientes industriais (HANGAL et al., 2004).

Foram usados quatro conjuntos com diferentes proporções de operações de memória, conforme mostra a Tabela 2.

Para cada combinação de parâmetros, dois casos de teste com sementes aleatórias distintas foram gerados e executados.

Tabela 2: Proporções de operações de cada tipo

Conjunto nº	Leituras	Escritas	Barreiras de memória
1	30%	66%	4%
2	48%	48%	4%
3	66%	30%	4%
4	80%	16%	4%

4.1.1 Formato dos programas paralelos

Para marcar pontos importantes durante a execução do programa, o gerador usa instruções artificiais denominadas *memory flags*.

Antes do primeiro *memory flag* há uma sequência de instruções de inicialização do programa, grande parte gerada automaticamente pelo compilador ou pela biblioteca de *threads* utilizada.

A primeira e a segunda *memory flags*, que aparecem somente na *thread* principal, marcam o início e o fim de uma sequência de inicialização dos endereços compartilhados entre as demais *threads*. Durante a execução, a plataforma armazena esses endereços e observa nos monitores apenas operações que os referenciam.

Após a segunda *memory flag*, a *thread* principal inicializa as demais. Cada *thread* aguarda por uma barreira de sincronização, para que todas elas iniciem a execução do caso de teste simultaneamente.

A terceira e quarta *memory flags*, presentes em todas as *threads*, marcam o início e fim da sequência de instruções que constituem o caso de teste. Essa sequência contém apenas operações de leitura e escrita e barreiras de memória e é a parte da execução submetida aos *checkers*.

Após a quarta *memory flag*, as *threads* finalizam sua execução e sinalizam à *thread* principal que terminaram, para que esta possa informar ao sistema operacional o fim do programa.

4.1.2 Tamanho dos casos de teste

Primeiramente, será explicado o raciocínio que justifica a ordem de magnitude dos casos de teste empregados nos experimentos. Os tamanhos dos casos de teste usados em ambientes industriais para verificação pós-silício estão no intervalo entre 50K (HANGAL et al., 2004) e 500K (MANOVIT; HANGAL, 2006) operações. Apesar de tamanhos ainda maiores serem usados em alguns experimentos para avaliar o impacto da complexidade no tempo

de execução (CHEN et al., 2009) (HU et al., 2012), eles são improváveis na prática devido aos limites de tempo disponível para verificação (o uso de um número maior de casos de teste mais curtos é mais provável). A verificação pós-silício permite casos de teste maiores porque a geração de *traces* é realizada executando os programas paralelos num protótipo do *hardware*. Na verificação pré-silício, entretanto, a geração de *traces* é realizada simulando a execução dos programas paralelos em uma representação executável. Supondo que uma representação ESL da plataforma seja uma ordem de magnitude mais lenta que a plataforma de *hardware*, casos de teste no intervalo de 50K a 500K operações levariam a um esforço de geração de *traces* similar ao intervalo de 5K a 50K operações para a verificação pré-silício.¹

Se os tamanhos dos casos de teste forem mantidos fixos, não se pode esperar a mesma *qualidade de verificação* quando o número de processadores aumenta (i.e. a cobertura pode diminuir com menos operações por *thread*). Por isso, para manter a mesma qualidade de verificação para um determinado *checker* deve-se manter o número de operações *por processador* fixo e deixar que os tamanhos dos casos de teste cresçam linearmente com os números de processadores. Como o gerador de casos de teste distribui as operações uniformemente entre as *threads*, isso significa que deve-se manter n/p fixo conforme p aumenta. Portanto, um *checker* pode ser avaliado em cenários de qualidade de verificação distintos pela atribuição de diferentes valores a n/p , que é o tamanho do caso de teste normalizado ao número de processadores, chamado sucintamente de *tamanho normalizado do caso de teste*. Foram realizados experimentos com $n/p = 125, 250, 500, 1K$ e $2K$.

4.1.3 Número de endereços compartilhados

Como explicado na seção anterior, a verificação pré-silício deve empregar casos de teste mais curtos que a pós-silício. Um ponto fundamental para mantê-los curtos é impôr o compartilhamento intenso de dados (HANGAL et al., 2004), induzindo a sobrecarga do subsistema de memória (DEORIO; WAGNER; BERTACCO, 2009). Isso pode ser obtido através da minimização do número de endereços usados pelo programa paralelo (SHACHAM et al., 2008). A literatura mostra que casos de teste com muitos endereços (entre 100 e 1000) ou com *benchmarks* contendo programas reais raramente induzem *data-races* e são, portanto, ineficazes (SHACHAM et al., 2008) (DEORIO;

¹Como o tempo de análise de um *checker post-mortem* é o mesmo se ele for usado para verificar uma representação executável ou um protótipo em *hardware*, seu reuso como um *checker* pré-silício somente é viável se o esforço total de geração de *traces* em tempo de projeto não for superior ao esforço total de geração de *traces* no protótipo de *hardware*.

WAGNER; BERTACCO, 2009). Por essas razões, o número de endereços compartilhados para a geração dos casos de teste foi restrito a um pequeno intervalo de valores: 2, 4, 8, 16 e 32.

4.2 ARQUITETURA SOB VERIFICAÇÃO

Como as técnicas a serem avaliadas são amplamente independentes de arquitetura, foram selecionados cenários com um conjunto de características comum a uma grande variedade de sistemas. Isso permite manter o foco na escalabilidade dos *checkers*.

Foi empregada a plataforma gem5 (BINKERT et al., 2011) para criar representações de projeto a serem verificadas. Seu nível de abstração é compatível com representações ESL, as quais devem ter um papel de importância cada vez maior, dadas as limitações de se modelar sistemas em *Register-Transfer Level* (RTL) com um número crescente de cores.

Como a escolha da arquitetura do conjunto de instruções (*instruction-set architecture* - ISA) tem pouco impacto na avaliação, adotou-se a arquitetura que tem melhor suporte na infraestrutura do gem5 (BINKERT et al., 2011).² Trata-se da UltraSPARC Architecture 2005 (Sun Microsystems Inc., 2008).

Foi escolhido um *template* do gem5 que possui uma hierarquia de memória *cache* estritamente inclusiva de três níveis. O primeiro nível é privativo e dividido em *cache* de instruções e *cache* de dados (ambas com 4KiB), o segundo (64KiB) também é privativo, porém unificado, e o terceiro nível (4MiB) é compartilhado entre os processadores. Seus blocos são de 64 bytes.

O mecanismo de coerência adota um protocolo MESI e é baseado em diretórios, contidos no terceiro nível da *cache*.

Como é necessária uma ampla relaxação de ordens entre operações para endereços distintos para se chegar a um alto desempenho, qualquer modelo que relaxe todas as quatro possíveis ordens entre operações de leitura e escrita ($L \rightarrow L$, $L \rightarrow S$, $S \rightarrow L$, $S \rightarrow S$) é representativo o suficiente para a avaliação. Foi adotado o modelo já implementado pelo gem5 (que é uma variação da arquitetura Alpha). O modelo adotado assemelha-se a diversos modelos populares, como *Weak Ordering* (WO) (DUBOIS; SCHEURICH; BRIGGS, 1986), cujas variações são usadas em processadores atuais com arquiteturas como ARMv7 (ARM, 2012) e ARMv8 (ARM, 2013a). Além disso, o uso amplo e prolongado desse modelo no simulador pela comunidade de Arquitetura de Computadores dá a confiança de que a plataforma adotada possa ser considerada, para fins de experimentação, como uma representação

²Esse melhor suporte é decorrente, provavelmente, do seu uso mais difundido.

Tabela 3: Caracterização dos erros de projeto

ID	Descrição	Localização
<i>e1</i>	Violação de barreira de memória para leituras	Unidade de controle de execução
<i>e2</i>	L2 não escreve dado no bloco da <i>cache</i> quando recebe uma resposta da L1	Controlador da <i>cache</i> L2
<i>e3</i>	L1 sempre vai ao estado exclusivo quando recebe dado para leitura	Controlador da <i>cache</i> L1
<i>e4</i>	L2 não envia mensagem de invalidação para L1 quando seu estado é Compartilhado ou Exclusivo	Controlador da <i>cache</i> L2
<i>e5</i>	Escrita inacabada não é vista por leitura para o mesmo endereço	Mecanismo de adiantamento
<i>e6</i>	Leitura não obtém o valor da escrita consolidada quando ela está na última posição da fila de escritas	Mecanismo de adiantamento
<i>e7</i>	Quando uma barreira de memória completa, outras barreiras de memória em execução são ignoradas pelas leituras	Unidade de controle de execução
<i>e8</i>	L1 vai para o estado Compartilhado quando recebe dado antigo para leitura, ao invés de permanecer Inválido	Controlador da <i>cache</i> L1

de projeto *pré-validada*.

4.2.1 Simulação de erros de projeto

Tentou-se implementar um conjunto significativo de erros de projeto, a fim de validar o *checker* de referência frente a erros comuns e descobrir particularidades de cada *checker* em relação a diferentes tipos de erros. Cada um desses erros foi injetado separadamente; em cada execução, há no máximo uma falha ativa na plataforma de simulação.

Uma grande variedade de erros de projeto é relatada na literatura, alguns encontrados em processadores reais, outros em modelos executáveis. Diversos deles foram implementados, porém apenas aqueles considerados adequados para a comparação dos *checkers* são mencionados nesta dissertação.³ A Tabela 3 condensa esses erros e explicita suas características básicas.

Certos cuidados tiveram que ser tomados na injeção de erros de projeto. Apenas o primeiro nível de *cache* é dividido entre instruções e dados. Assim,

³Certos erros são detectados muito facilmente pelos *checkers*; outros nunca são exercitados pela plataforma de simulação. Eles não foram incluídos na comparação, pois não acrescentariam nenhum resultado interessante.

os níveis superiores não diferenciam blocos que contenham esses dois tipos de elementos. Se os erros forem aplicados a esmo aos blocos e mensagens do sistema de memória, eles podem provocar alterações ou reordenamentos nas instruções, que causariam um comportamento inesperado no programa ou até mesmo uma instabilidade na plataforma, podendo interromper a execução do teste. Para evitar isso, quando a plataforma detecta o trecho de inicialização dos endereços usados no programa (mencionado na Seção 4.1.1), esses endereços são armazenados, para que os erros sejam aplicados somente a operações que os referenciam.

4.2.2 Migração de *snooping* para diretórios

A plataforma tem dois sistemas de simulação de memória distintos. No sistema clássico, a coerência é implementada através de *snooping* e as *caches* são conectadas diretamente através de portas de entrada e saída. Quando o antigo M5 (BINKERT et al., 2006) foi mesclado com a plataforma GEMS de simulação de memória (MARTIN et al., 2005), ele se tornou o gem5 e adquiriu o novo sistema de simulação de memória Ruby (BINKERT et al., 2011). Nele, o protocolo de coerência e as interconexões são independentes (provendo uma enorme flexibilidade) e cada componente de memória possui *buffers* de entrada e saída em suas portas, aumentando a precisão da simulação. No entanto, ao executar testes com a coerência baseada em diretórios, a própria plataforma identificava *deadlocks* no protocolo de coerência⁴ e os *checkers* detectavam erros na plataforma, que deveria ser uma referência correta. Para contornar esses problemas técnicos, os componentes genéricos (comuns a todas as implementações de coerência) do modelo Ruby foram modificados para eliminar esses erros, chegando a uma plataforma estável, testada com sucesso com três níveis de *cache* (dois privativos) e um protocolo de coerência MESI baseado em diretórios. A interconexão usada possui uma topologia em estrela baseada em um comutador (*switch*) com roteamento simples.

4.2.3 Número de núcleos

A Tabela 4 ilustra, para *smartphones high-end* da Samsung, que o número de núcleos costuma dobrar a cada ano, havendo uma desaceleração no último ano. Entretanto, com a introdução da arquitetura big.LITTLE pela ARM

⁴Certos componentes do subsistema de memória enviavam mensagens e nunca obtinham uma resposta, impedindo que as operações de memória em execução terminassem e que a execução do programa avançasse.

Tabela 4: Crescimento do número de núcleos em *smartphones*

Produto	Lançamento	Arquitetura
Galaxy S	Junho de 2010	1 núcleo Cortex A8
Galaxy S II	Mai de 2011	2 núcleos Cortex A9
Galaxy S III	Mai de 2012	4 núcleos Cortex A9
Galaxy S4	Março de 2013	4 núcleos Cortex A15 e 4 núcleos Cortex A7 (big.LITTLE)
Galaxy S5	Abril de 2014	4 núcleos Cortex A15 e 4 núcleos Cortex A7 (big.LITTLE)

(ARM, 2013b), que divide os núcleos em dois *clusters* com processadores distintos, o número de núcleos de uma arquitetura CMP homogênea tende a ser metade do número de núcleos nominal. Baseando-se na taxa de crescimento do número de núcleos de processadores para celulares, é improvável que o número de núcleos por *cluster* seja maior do que 32 nos próximos 3 anos.

Apesar de os servidores demandarem um poder de processamento muito maior, costuma-se multiplicar sua capacidade através da utilização de diversos processadores em paralelo, cada um deles com múltiplos núcleos. Como o problema de verificação de modelos de memória se aplica aos processadores isolados, deve-se considerar o número de núcleos de cada um deles. Tomando-se como exemplo os processadores Opteron da AMD, não houve um crescimento significativo do número de núcleos nos últimos anos, com os processadores mais poderosos contendo 16 núcleos desde o ano de 2012 (AMD, 2012).

Diante dessa tendência, para simular cenários compatíveis com a evolução dos próximos três anos, foram realizados experimentos com números de processadores que variam de 2 a 32.

Em resumo, os parâmetros arquiteturais e o modelo de memória utilizado nos experimentos são compatíveis com a evolução esperada.

4.3 IMPLEMENTAÇÃO DE CHECKERS

A Tabela 5 resume as características principais dos *checkers* usados nos experimentos e lhes atribui acrônimos para facilitar a apresentação de resultados.

Os *checkers* escolhidos representam o estado da arte para cada uma das classes de verificadores dinâmicos analisados na Seção 3.2.

A implementação do *checker* baseado em emparelhamento estendido

Tabela 5: *Checkers* sob comparação

Acrônimo	Referência	Ideia-chave	Garantias
IBE	(HU et al., 2012)	Inferência (melhor esforço)	Nenhuma
IBT	(HU et al., 2012)	Inferência, backtracking	Plenas
EXM	(RAMBO et al., 2012)	Emparelhamento de grafos bipartidos	Plenas
SSB	(SHACHAM et al., 2008)	Única <i>scoreboard</i> relaxada	Nenhuma
MSB	(FREITAS et al., 2013)	Múltiplas <i>scoreboards</i> relaxadas	Plenas

(EXM) foi disponibilizada por seus autores para uso nos experimentos. Ele está descrito com detalhes em (RAMBO, 2012).

A implementação do *checker* baseado em múltiplas *scoreboards* (MSB) também foi disponibilizada por seus autores para uso nos experimentos. Ele está descrito com detalhes em (FREITAS, 2012).

A técnica baseada em *scoreboard* relaxada (SHACHAM et al., 2008) é uma ferramenta para construção de um *checker*, em que regras podem ser adicionadas gradativamente à medida que o projeto do sistema sob verificação avança. Diferente de um *golden model*, em que cada evento do sistema deve corresponder a apenas um valor possível, predeterminado, essas regras definem quando um valor é adicionado à *scoreboard* (a cada evento de escrita) e quando um valor é removido da *scoreboard* (a cada evento de leitura). Como as regras são muito flexíveis e dependem das características do sistema, os autores não as descrevem em detalhes, nem indicam precisamente os pontos que devem ser monitorados, dando apenas alguns exemplos.

Assim, teve-se que implementar uma versão própria (SSB) de um *checker* construído com aquela técnica, com regras distintas daquelas descritas em (SHACHAM et al., 2008). Para isso, foi implementado um conjunto básico de regras, capaz de expôr alguns dos erros de projeto adotados para os experimentos.

Dentre as três regras descritas em (SHACHAM et al., 2008), uma não se aplica ao modelo de memória usado, outra foi adaptada (pois foi concebida para o modelo TSO) e a outra foi implementada da forma descrita. Para se detectar todos os erros, teria que se implementar regras específicas para cada conjunto de erros semelhantes. A detecção de erros induzidos por barreiras de memória, por exemplo, demandaria uma série de estruturas de suporte à

scoreboard que não estão presentes na plataforma.⁵

Como XCHECK não está disponível em domínio público, teve-se que implementar uma versão desse *checker* baseada nos algoritmos descritos em (HU et al., 2012). Este *checker* admite dois modos de operação: sem (IBE) ou com (IBT) garantias de verificação.

A complexidade reportada para XCHECK em (HU et al., 2012) é $O(nC^p p^3)$. Entretanto, ele foi concebido para a verificação do processador Godson-3B, cujo modelo de memória se assemelha ao TSO. A utilização dessa técnica para modelos relaxados leva a uma maior complexidade. Para entender o porquê, será necessário distinguir duas noções: A noção de *fronteira* (GIBBONS; KORACH, 1994) refere-se a operações que estão em execução, mas ainda não terminaram. Uma *fronteira* contém uma operação para cada processador. A noção de *conjunto de fronteira* refere-se às operações de todos os processadores que já foram processadas. Ora, a análise de complexidade em (HU et al., 2012) baseia-se em uma prova anterior (GIBBONS; KORACH, 1994), que supõe um modelo de memória com consistência sequencial (SC). Neste caso, para cada *fronteira*, só há um conjunto de *fronteira* possível. No entanto, para um modelo de memória relaxado (como o que estamos utilizando), operações para endereços distintos podem ou não terem sido executadas quando se chega a uma *fronteira*. Portanto, foi necessário otimizar o algoritmo original de XCHECK para que sua avaliação experimental (ao resolver uma instância do problema de verificação envolvendo modelos relaxados) seja representativa da classe de *checkers* baseados em inferência.

As adaptações necessárias para produzir versões do XCHECK compatíveis com o modelo de memória utilizado nos experimentos são listadas a seguir:

- **Pré-processamento:** Os autores do XCHECK descrevem uma técnica para capturar os tempos de início e término das operações de tal forma que os tempos reais de início e término sempre estarão dentro do intervalo capturado. A forma como os tempos são capturados garante que operações consecutivas em ordem de programa sempre têm tempos em ordem crescente. Quanto menos precisos os tempos capturados, menos conhecimento se terá das ordens temporais entre as operações, que são o ponto-chave para a aceleração do *checker*. Como a plataforma usada é uma representação executável, ao invés de um *hardware* real, podemos ter uma precisão maior nos tempos das operações. No entanto, essa maior precisão captura também o efeito do reordenamento de instruções,

⁵Os autores disponibilizam *online* o código-fonte de uma versão do *checker* implementada em OpenVera (Synopsys Inc., s.d.). Entretanto, ela é destinada à verificação de sistemas com o modelo de memória Transaction Coherence and Consistency (TCC) (HAMMOND et al., 2004), sendo incompatível com o modelo Alpha adotado para os experimentos.

o que faz com que seus tempos não estejam em ordem crescente quando comparados com a ordem de programa. Isso dificulta certos aspectos do algoritmo. Para não comprometer o tempo de verificação, foi feito um pré-processamento (cujo tempo não é agregado aos tempos de simulação aqui reportados) em que são calculadas, para cada operação, todas as operações pertencentes ao seu intervalo temporal e os seus sucessores diretos em ordem temporal.

- **Operações candidatas:** Para cada operação da fronteira em que o algoritmo se encontra, existe um conjunto de operações candidatas a substituí-la, quando esta for retirada da fronteira. Para uma operação entrar nesse conjunto de candidatas, é necessário que todas as operações que a precedem em ordem de programa já tenham sido executadas previamente, ou estejam na fronteira. Isso garante que a operação candidata seja do mesmo processador que aquela que está na fronteira e evita que muitas operações inaptas sejam processadas, o que seria muito custoso computacionalmente.
- **Fronteiras contém apenas operações de escrita:** A navegação do algoritmo de *backtracking* é feita através das fronteiras. Como não há ordenamento especificado entre operações não conflitantes, torna-se desnecessário induzir combinações envolvendo operações de leitura. Assim, a navegação entre fronteiras se dá apenas pela ordem entre operações de escrita, e as arestas oriundas de operações de leitura anteriores são adicionadas sempre para uma operação de escrita que esteja saindo da fronteira (i.e. quando a operação de escrita é considerada executada).⁶
- **Execução de uma operação:** Para cada operação, há um contador de arestas incidentes (i.e. um contador de operações predecessoras). Para que uma operação possa ser executada, é necessário que esse contador seja zero (i.e. todas as operações anteriores já executaram). Quando uma operação é executada, todas as suas sucessoras diretas terão seu contador decrementado. Da mesma forma, quando é feito *backtracking* da execução de uma operação, o contador de suas sucessoras é incrementado.
- **Operações nulas:** Assim como no algoritmo original, é adicionada uma operação nula para cada processador. Elas são processadas como se fossem uma barreira de memória, sendo predecessoras de todas as operações do mesmo programa. Elas possuem também ordem temporal

⁶Arestas incidentes nas operações de leitura são diretamente inferidas pelo mapeamento de leituras para escritas.

com todas as outras operações, garantindo que serão as primeiras a executar. Além disso, foram adicionadas ordens artificiais entre elas (isso evita que o algoritmo tente reordenar as operações nulas).

- **Conjuntos de fronteira:** Cada vez que se chega a uma fronteira, o conjunto de fronteira é armazenado. Se, depois de um *backtracking*, aquela fronteira for atingida novamente com o mesmo conjunto de fronteira, ela é imediatamente considerada inválida e é feito *backtracking*. Se existisse apenas um endereço compartilhado no programa, todas as operações de escrita de um mesmo processador estariam ordenadas, de forma semelhante à consistência sequencial (SC). Neste caso, para todos os caminhos que levarem a uma mesma fronteira, os conjuntos de fronteira serão os mesmos, assim como ocorre no XCHECK original, e, provavelmente (GIBBONS; KORACH, 1994), não é necessário continuar seguindo este caminho. A complexidade, então, seria aquela reportada pelos autores (HU et al., 2012). Entretanto, como na prática há certamente mais que um endereço compartilhado, a complexidade reportada em (HU et al., 2012) não é válida para modelos relaxados. Apesar disso, os resultados experimentais mostraram que a otimização acima descrita reduz consideravelmente o tempo médio de análise mesmo nesse cenário (relaxado).

5 RESULTADOS EXPERIMENTAIS

Este capítulo resume os principais resultados obtidos da avaliação experimental comparativa de cinco *checkers* dinâmicos. Uma versão preliminar destes resultados, que foi publicada em trabalho apresentado em evento (HENSCHER; SANTOS, 2013), não será aqui reportada, uma vez que os resultados aqui apresentados envolvem um maior número de *checkers* e utilizam uma configuração experimental mais ampla. Portanto, os resultados aqui apresentados são em sua grande parte inéditos.¹ Não é do conhecimento do autor que haja na literatura uma avaliação comparativa tão ampla, envolvendo tantos *checkers* diferentes sob as mesmas condições experimentais, as quais foram justificadas no Capítulo 4.

Foram medidas as porcentagens de casos de teste que encontraram um dado erro². Como os casos de teste são gerados pseudo-aleatoriamente, tal porcentagem pode ser interpretada como a probabilidade de um caso de teste achar esse erro, i.e. a *eficácia* de um conjunto de casos de teste.

Para capturar a *eficiência* relativa de cada *checker*, também foram medidos os tempos associados a cada caso de teste. Para *checkers post-mortem* (EXM, IBE e IBT), o *esforço* de verificação inclui o tempo para executar o programa do caso de teste de forma a gerar *traces completos* a priori, além do tempo para realizar a análise desses *traces*. Para os *checkers on-the-fly* (SSB e MSB), o esforço de verificação é o tempo para executar o programa e a análise simultaneamente (os *traces* gerados não são completos se um erro for detectado antes do fim da simulação). Quando um erro é encontrado para um dado caso de teste, o tempo é medido até sua detecção. Entretanto, quando um caso de teste não expõe um erro, o tempo da análise completa é medido.

As Seções 5.1, 5.2 e 5.3 comparam quatro dos *checkers* selecionados (IBE, EXM, SSB e MSB) sob três pontos de vista distintos: a sensibilidade da eficácia ao tipo de erro, o impacto na eficiência e na eficácia com o crescimento do número de operações do caso de teste e o impacto na eficiência e na eficácia com o crescimento do número de processadores. Ao final, a Seção 5.4 compara todos os cinco *checkers* e justifica por que os resultados obtidos com o quinto *checker* selecionado (IBT) não foram mostrados nas seções anteriores.

¹ Eles serão submetidos para publicação em periódico como parte dos resultados reportados em artigo que está em preparação.

² Os erros, que foram injetados propositalmente na representação executável de referência, foram descritos na Seção 4.2.1.

5.1 SENSIBILIDADE AO TIPO DE ERRO

A Figura 5 mostra como a eficácia dos casos de teste varia entre os diferentes tipos de erros para três cenários diferentes de qualidade de verificação (correspondentes a diferentes valores da relação n/p). A figura também mostra (sob o nome “nulo”) a porcentagem de casos de teste indicando erro em uma plataforma correta, i.e. a porcentagem de falsos positivos.

Como esperado, alguns erros são mais difíceis de encontrar que outros (para um dado n/p na Figura 5, a dificuldade aumenta da esquerda para a direita) e a eficácia cresce com o tamanho dos casos de teste para a maior parte dos erros e dos *checkers* (para um dado *checker* na Figura 5, isso é capturado de (a) até (c)).

Os *checkers* IBE e SSB não oferecem garantias de verificação, porém IBE exibe uma eficácia competitiva em relação aos outros *checkers*. Entretanto, note que SSB conseguiu encontrar apenas os erros e_4 e e_8 para o conjunto de regras implementadas. Isso ilustra o preço pago por sua flexibilidade. Sua eficácia é muito dependente do repertório de regras.

Para entender por que a eficácia em encontrar alguns erros nem sempre aumenta conforme o tamanho dos casos de teste, considere os erros e_2 , e_3 e e_4 . Foi observado nos experimentos que nenhum deles é exposto quando o número de endereços compartilhados é menor que 4. Como 1/5 dos casos de teste gerados contém apenas 2 endereços compartilhados, isso explica porque a eficiência satura próxima de 80% para esses erros.

Apesar da inevitável variação conforme o tipo de erro, a eficácia do *checker* MSB é superior a todos os outros *checkers* para os três cenários de qualidade de verificação. Finalmente, note que uma porcentagem de 0% de erros foi medida para o cenário denominado “nulo”, o qual representa uma plataforma correta. Isso mostra que nenhum dos *checkers* indica erros numa plataforma correta, ou seja, nenhum deles sinaliza falsos positivos.

5.2 IMPACTO DO NÚMERO CRESCENTE DE OPERAÇÕES

Para capturar a eficácia global, foi calculada a porcentagem dos casos de teste em que são encontrados erros de qualquer tipo para todo o intervalo de valores de p , mas individualmente para cada cenário de qualidade de verificação (n/p), como é mostrado na Figura 6.

Como esperado, para todos os *checkers*, a eficácia média cresce com o tamanho normalizado dos casos de teste. Note que os *checkers* com garantias plenas (EXM e MSB) possuem uma eficácia semelhante, cuja diferença em relação à de IBE é de pelo menos 9% para todos os casos. Isso mostra como

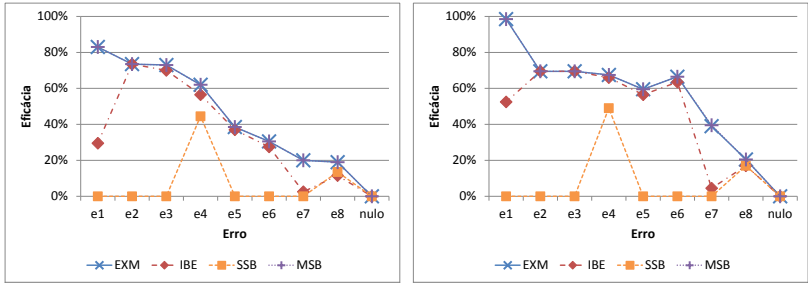
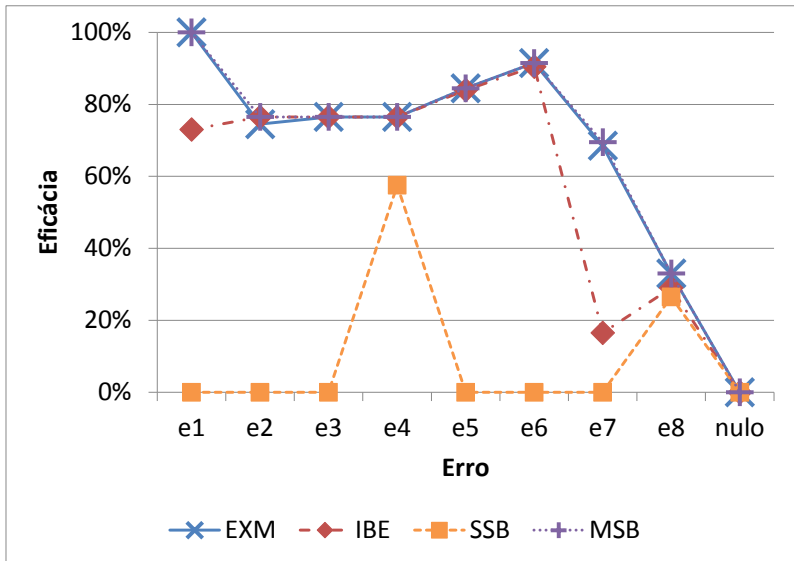
(a) $n/p = 125$ (b) $n/p = 500$ (c) $n/p = 2K$

Figura 5: Diferentes eficácias para erros distintos

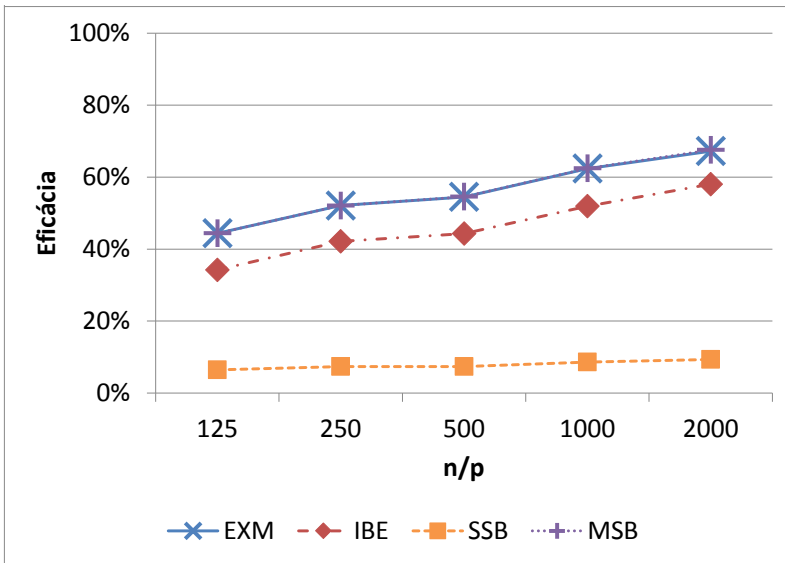


Figura 6: Eficácia para casos de teste com tamanho crescente

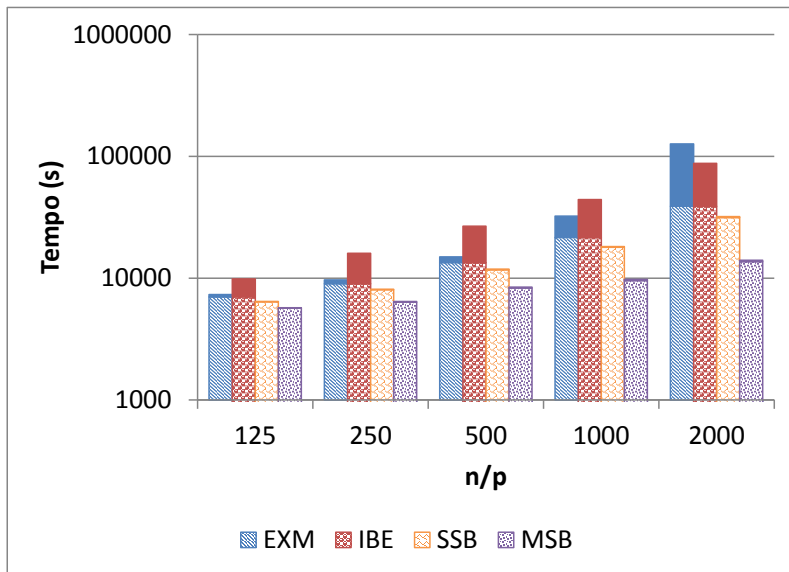


Figura 7: Eficiência para casos de teste com tamanho crescente

as garantias plenas ajudam na minimização do tamanho dos casos de teste. Por exemplo, os *checkers* MSB e EXM têm a mesma eficácia de 44% (com $n/p = 125$) que o *checker* IBE (com $n/p = 500$) com casos de teste 4 vezes menores! O checker SSB (que não possui garantias) obteve a pior eficácia, encontrando erros em menos do que 10% dos casos de teste para $n/p = 2000$.

Para capturar a eficiência global de todo o conjunto de testes e todo o conjunto de erros, foram somados os esforços individuais para um dado tamanho normalizado de caso de teste, conforme mostrado na Figura 7.

Na Figura 7, as barras foram divididas para distinguir o tempo gasto com geração de *traces* (parte inferior, texturizada) do tempo de análise (parte superior, com cor sólida). Observe que o MSB sempre leva ao menor esforço e à menor taxa de crescimento com o tamanho normalizado dos casos de teste. Em média, MSB é 1,7 vezes mais rápido que SSB (que também é um *checker on-the-fly*). Além disso, MSB é, em média, 4,3 vezes mais rápido que o EXM e 4,1 vezes mais eficiente que o IBE.

Note que o uso de um *checker on-the-fly* não apenas torna o tempo de análise insignificante, se comparado ao esforço de simulação, mas também reduz o próprio tempo de simulação, pois a análise é abortada assim que um

erro é detectado. Por exemplo, para $n/p = 1000$, MSB reduz o esforço de simulação em 2 vezes se comparado a qualquer *checker post-mortem*, enquanto SSB o reduz em 1,2 vezes.

5.3 IMPACTO DO NÚMERO CRESCENTE DE PROCESSADORES

Para capturar a eficácia dos casos de teste com o crescimento do número de processadores, foi computada a porcentagem de casos de teste que encontram erros (de qualquer tipo), para cada caso, como é mostrado na Figura 8 para três cenários de qualidade de verificação distintos.

Note que manter o número de operações por processador fixo evita a diminuição da eficácia com o crescimento do número de processadores, confirmando a hipótese da qualidade de verificação levantada na Seção 4.1.2. Para um n/p fixo, a eficácia aumenta para um número baixo de processadores e tende a saturar para números maiores. Isso pode ser explicado da seguinte forma: por um lado, se um erro for situado em um processador ou uma *cache* privativa, ele será replicado p vezes, tornando-o mais provável de ser exposto por pelo menos um dos processadores. Em contrapartida, como o número de mensagens e a complexidade do diretório crescem com o número de processadores, aumenta a probabilidade de um erro na manipulação das mensagens ou na atualização dos diretórios ocorrer na interconexão ou na *cache* compartilhada (dado que os diretórios residem nesta *cache*).

Para capturar a eficiência com o aumento do número de processadores, foram somados os esforços individuais para cada caso, conforme mostrado na Figura 9 para três cenários de qualidade de verificação distintos.

Note que, além de ser o *checker* mais rápido independentemente do número de processadores, MSB também exibe a segunda menor taxa de crescimento em relação a p , perdendo apenas para EXM. Para comparar a escalabilidade dos *checkers*, os valores usados na Figura 9c foram interpolados para gerar uma equação da forma $t = k(1 + r)^p$, onde t é o tempo total de verificação, k é uma constante, r é a taxa de crescimento e p é o número de processadores. O *checker* EXM leva à menor taxa de crescimento ($r = 0,043$), seguido por MSB ($r = 0,070$) e SSB ($r = 0,091$), os três significativamente menores que o valor obtido para IBE ($r = 0,229$). Apesar de ter a menor taxa de crescimento com o número de processadores, o *checker* EXM mostra-se inferior aos outros quando submetido a casos de teste com maior número de operações.

Assim, o uso pré-silício de um *checker on-the-fly* (tal como MSB) provê plenas garantias a uma taxa de “escalabilidade” pelo menos 3 vezes menor do que a de um *checker* baseado em inferências sem garantias de verificação (tal

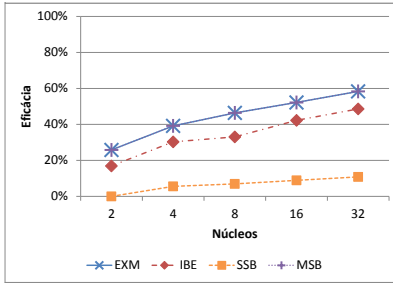
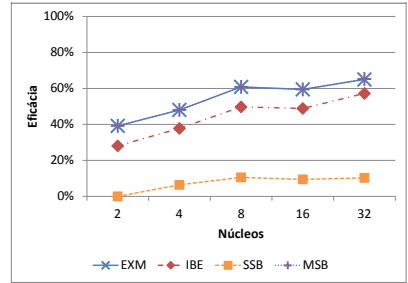
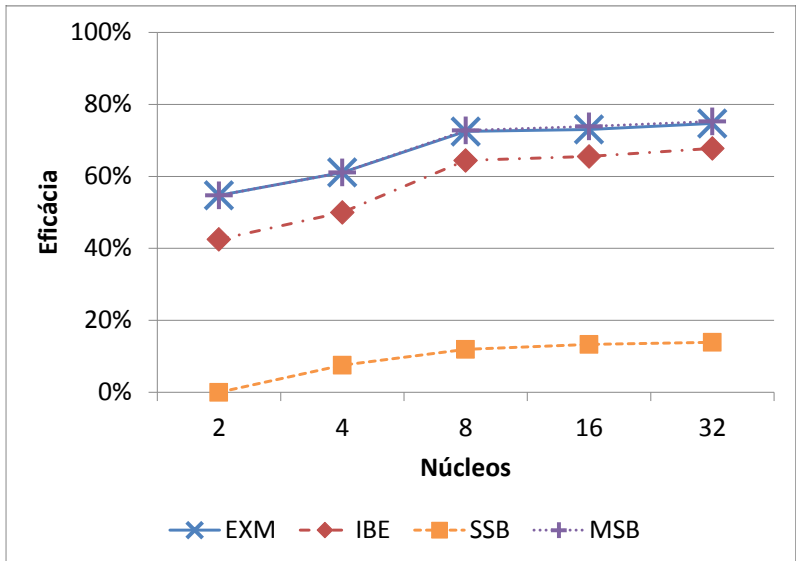
(a) $n/p = 125$ (b) $n/p = 500$ (c) $n/p = 2K$

Figura 8: Eficácia com o aumento do número de processadores

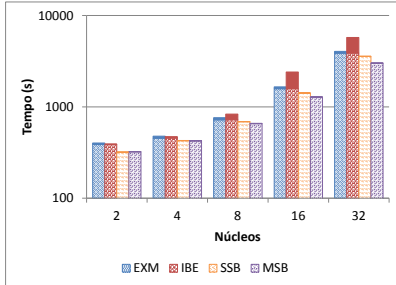
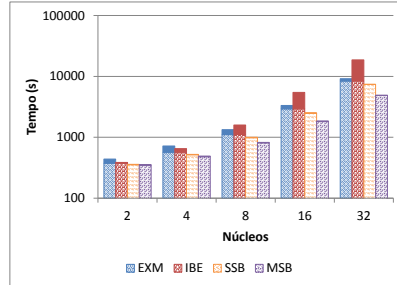
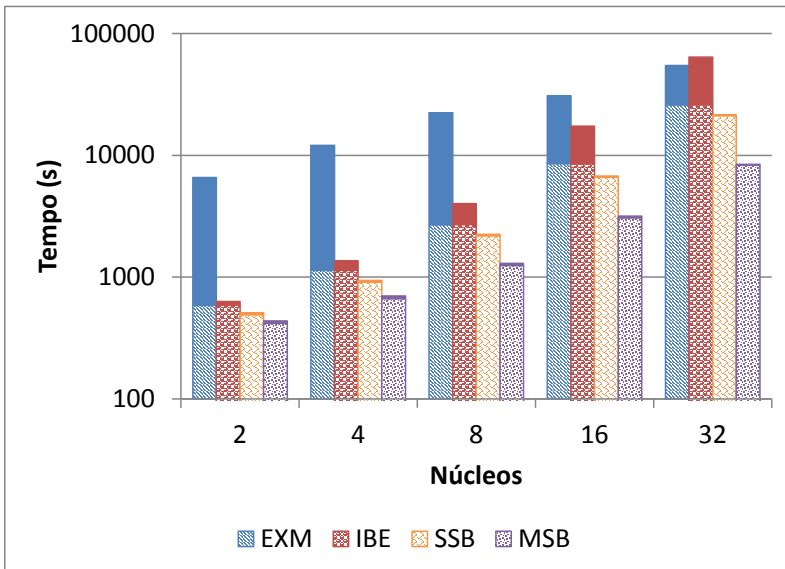
(a) $n/p = 125$ (b) $n/p = 500$ (c) $n/p = 2K$

Figura 9: Eficiência com o aumento do número de processadores

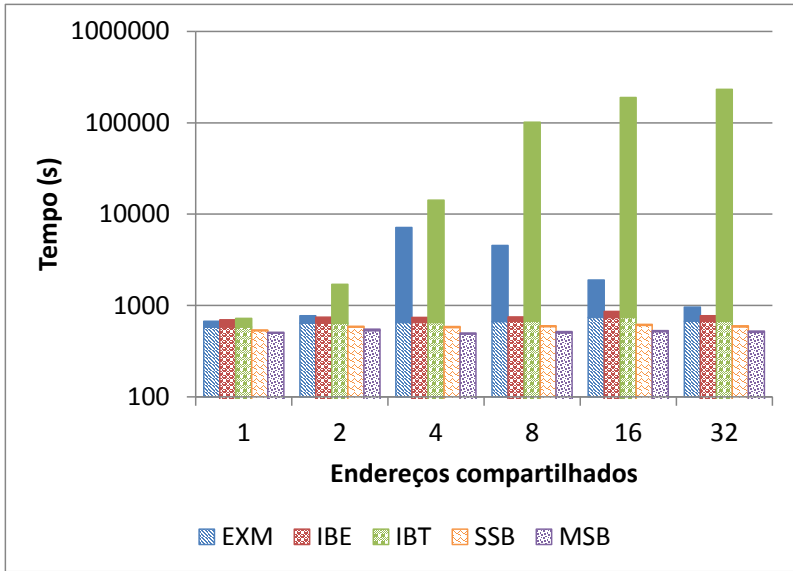


Figura 10: Eficiência com o número crescente de endereços

como IBE).³

5.4 IMPACTO DO NÚMERO CRESCENTE DE ENDEREÇOS

Um último experimento foi realizado para avaliar o impacto do número de endereços compartilhados. Os resultados desse experimento justificam por que a eficácia e eficiência do *checker* IBT não foi apresentada nas Seções 5.1 a 5.3. Para este experimento, apenas plataformas contendo quatro processadores foram avaliadas. Ainda assim, teve-se que impôr um limite de tempo para IBT encontrar os erros. Se ele não encontrasse um erro em um dado caso de teste em até duas horas, o caso de teste era interrompido e considerava-se que IBT não indicou um erro.

Na Figura 10, observa-se que IBT tem uma eficiência razoável quando o programa contém apenas um endereço compartilhado: um pouco acima de seus concorrentes e até mesmo de sua versão de melhor-esforço (IBE).

³Na verdade, resultados experimentais adicionais mostraram que MSB, por não se adaptar bem à equação exponencial utilizada, possui uma taxa de crescimento ainda menor do que a calculada.

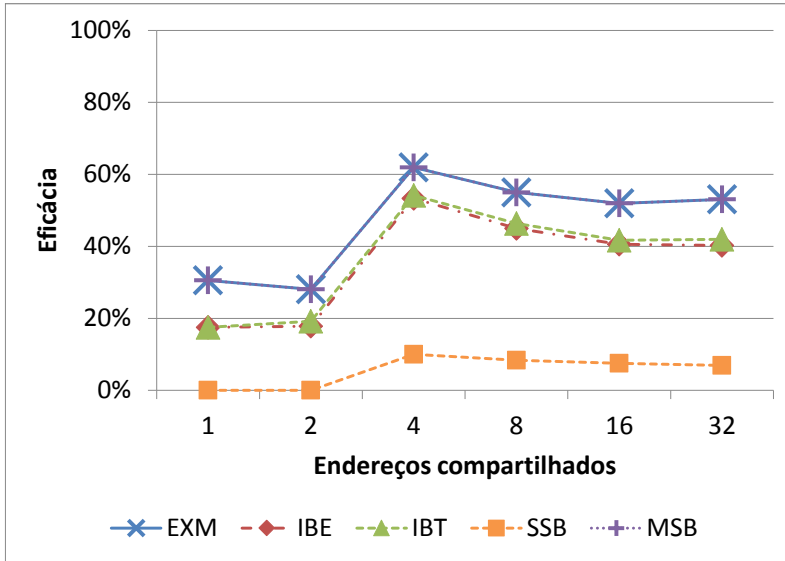


Figura 11: Eficácia com o número crescente de endereços

Porém, ao aumentar o número de endereços, seu tempo de análise cresce exponencialmente, até saturar devido ao limite imposto para os casos de teste.

Essa eficiência razoável para um único endereço deve-se ao fato de que estes programas degeneram o modelo de memória extremamente relaxado da plataforma de simulação em algo semelhante a SC e TSO, para os quais o *checker* foi projetado e possui garantias formais. Neste cenário extremo de verificação, todas as operações estariam ordenadas,⁴ pois referenciam o mesmo endereço.

O comportamento peculiar de EXM deve-se ao fato de que alguns dos erros só são expostos quando o programa contém 4 ou mais endereços compartilhados, ponto que marca o pico de seus tempos de análise, antes de começar a decrescer por causa da diminuição da frequência de *data-races* no programa.

Nota-se que o comportamento indesejável de IBT explica-se pelo uso de *backtracking* para prover essencialmente as mesmas garantias de verificação obtidas com MSB.

⁴Exceto pela ordem entre as operações de leitura.

Apesar das mesmas garantias, a eficácia de MSB é superior à de IBT, como ilustra a Figura 11. A eficácia de IBT é apenas marginalmente superior à de IBE. Os resultados indicam que, apesar de IBT realizar a análise em um tempo médio três ordens de grandeza superior a IBE, ele encontra erros em apenas 1,03 vezes mais casos de teste.

Portanto, este experimento mostra que *checkers* baseados em inferências usando *backtracking* são inadequados para o uso em modelos de memória relaxados, mesmo quando feitas otimizações (como as descritas na Seção 4.3). Por esta razão, os valores de IBT não foram mostrados nas Figuras 5 a 9.

6 PROPOSTA DE GENERALIZAÇÃO PARA *CHECKERS* ON-THE-FLY

6.1 MOTIVAÇÃO

Durante a análise da literatura (descrita no Capítulo 3) e avaliação experimental (descrita no Capítulo 4), encontrou-se uma limitação dos *checkers* dinâmicos reportados até o momento: eles são incapazes de tratar adequadamente modelos com fraca atomicidade de escrita, sem emitir diagnóstico de falso positivo. Esta limitação é agravada pelo fato de que, em arquiteturas recentemente lançadas (ARM, 2013a), os modelos de memória incorporam essa característica, uma vez que ela é inevitavelmente exposta com o aumento do número de processadores por chip, devido ao suporte de múltiplos níveis hierárquicos de consulta necessário (GHARACHORLOO, 1995), distâncias intra-chip crescentes (SCHUCHHARDT et al., 2013) e tráfego de coerência expressivo (KRISHNA et al., 2013).

Alguns dos *checkers* dinâmicos reportados, além de resolver instâncias dos Problemas 2 e 3 que supõem *forte* atomicidade de escrita, não utilizam observabilidade adequada para prover plenas garantias de verificação sob *fraca* atomicidade de escrita.

Por essa razão, este capítulo estende a observabilidade requerida (com a adição de um novo monitor) e propõe algoritmos mais gerais para construir *checkers* dinâmicos.

A Seção 6.2 ilustra, através de um exemplo simples, uma instância do problema de verificação para o cenário de fraca atomicidade de escrita. Em seguida, a Seção 6.3 justifica por que é preciso estender a observabilidade para prover plenas garantias de verificação, além de mostrar como fazê-lo com um número mínimo de monitores. A Seção 6.4 mostra esquematicamente a estrutura do *checker* generalizado em termos dos processos e rotinas que o constituem. Por fim, a Seção 6.5 apresenta e comenta a descrição formal dos algoritmos propostos para o *checker* generalizado. O status de progresso do trabalho de validação teórica é apresentado na Seção 6.6.

6.2 EXEMPLO ILUSTRATIVO

Quando o comportamento não-atômico das operações de escrita é exposto, mais eventos são observados para o mesmo programa, como mostra a Figura 12. Note que, nas Figuras 12c e 12d, cada *operação de escrita* foi separada em um *evento de escrita* por processador (esse é, por sinal, o mesmo

T_1	T_2
$(S_1)_a$	$(L_3)_a$
$(S_2)_a$	$(L_4)_a$

(a) programa

P_1	P_2
$S_1 = (W, a, 1)$	$L_3 = (R, a, 1)$
$S_2 = (W, a, 2)$	$L_4 = (R, a, 1)$

(b) *traces*

time	P_1	P_2
1	$S_1 : (W, a, 1)$	
2		$S_1 : (W, a, 1)$
3		$L_3 = (R, a, 1)$
4		$L_4 = (R, a, 1)$
5	$S_2 : (W, a, 2)$	
6		$S_2 : (W, a, 2)$

(c) um comportamento

time	P_1	P_2
1	$S_1 : (W, a, 1)$	
2		$S_1 : (W, a, 1)$
3		$L_3 = (R, a, 1)$
4	$S_2 : (W, a, 2)$	
5		$L_4 = (R, a, 1)$
6		$S_2 : (W, a, 2)$

(d) outro comportamento

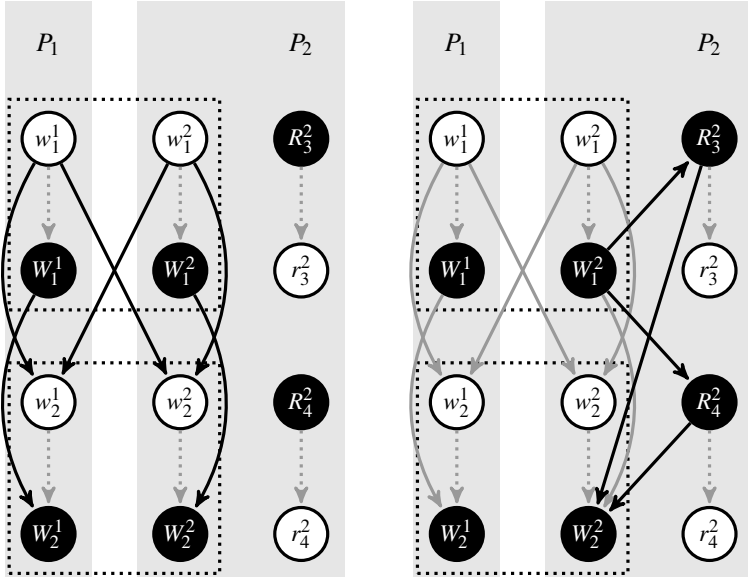
Figura 12: Um programa, seus *traces* e dois comportamentos não-atômicos

modelo adotado por algumas abordagens formais (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002)). O evento de escrita que acontece no mesmo processador em que a operação é emitida é destacado em negrito (um evento de escrita não destacado representa um evento de coerência).

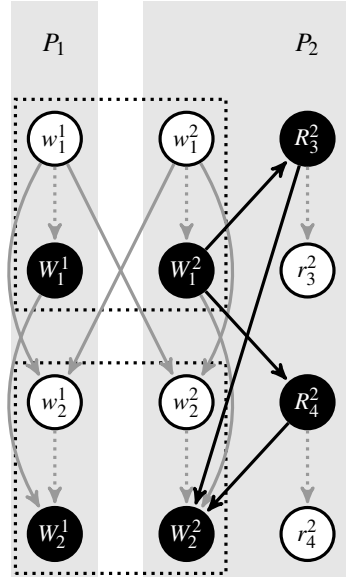
Infelizmente, essa modelagem é muito simplista para capturar alguns comportamentos válidos especificados por modelos de memória com fraca atomicidade de escrita. Por esse motivo, é empregada uma representação mais precisa para operações de escrita (GHARACHORLOO, 1995), onde cada evento de escrita é ainda dividido em eventos de consolidação e conclusão (resultando no dobro de eventos por escrita, se comparado ao modelo empregado em (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002)). Do ponto de vista de um processador que emite uma operação de escrita, um evento de *consolidação de escrita* representa a colocação de um valor em um *buffer* de saída; para os outros processadores, ele representa a colocação de uma mensagem de invalidação ou atualização nos seus *buffers* de entrada. Um evento de *conclusão de escrita* representa a real invalidação ou atualização de uma cópia na memória.

Diferente de (GHARACHORLOO, 1995) e (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002), emprega-se aqui uma representação mais precisa para operações de leitura, onde elas são divididas em eventos de consolidação e conclusão (para tratar corretamente a especulação de operações de leitura).¹ Um evento de *conclusão de leitura* representa de fato a leitura de

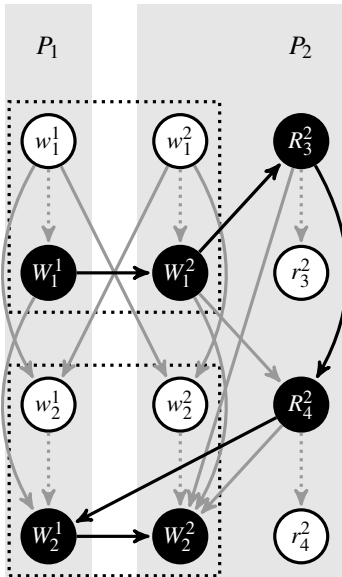
¹ Apesar desse refinamento não ser necessário para analisar questões de atomicidade de escrita, ele é introduzido aqui como uma prévia do modelo que está sendo construído para trabalhos futuros.



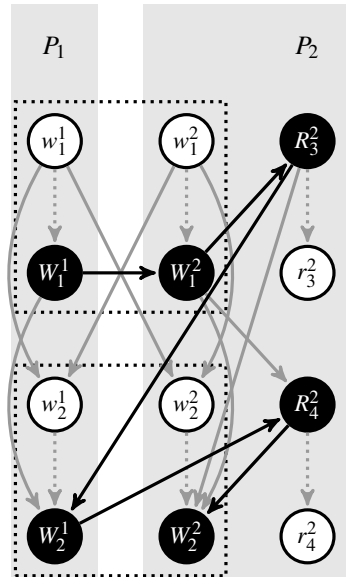
(a) ordem de programa



(b) ordem inferida



(c) uma ordem temporal válida



(d) uma ordem temporal inválida

Figura 13: Ordenamento de eventos sob fraca atomicidade de escrita

um valor da *cache* local do processador que emitiu a *operação de leitura*. Um evento de *consolidação de leitura* representa a transferência desse valor para o registrador destino da *operação de leitura*.

As Figuras 13a–d empregam tal representação generalizada quando se captura os ordenamentos de eventos correspondentes ao programa, os *traces* e as execuções mostrados nas Figuras 12a–d, respectivamente. Retângulos pontilhados delimitam todos os eventos induzidos por uma determinada operação de escrita. Usamos a notação pictórica de (GHARACHORLOO, 1995) para eventos de escrita e a estendemos para eventos de leitura: um vértice branco representa um evento de consolidação; um preto, representa um evento de conclusão. Arestas pontilhadas ilustram o fato que uma operação de escrita deve ser consolidada antes de concluir, enquanto uma operação de leitura deve concluir antes de ser consolidada.

Dada uma operação de escrita S_j , seus eventos de consolidação e conclusão em relação ao processador i são denotados por w_j^i e W_j^i , respectivamente. Dada uma operação de leitura L_j emitida pelo processador i , seus eventos de consolidação e conclusão são denotados por r_j^i e R_j^i , respectivamente.

Assumamos agora que a relaxação de atomicidade de escrita resulte do reconhecimento antecipado de pedidos de invalidação induzidos por cada operação de escrita (GHARACHORLOO, 1995). Para compelir a ordem de programa entre operações de escrita conflitantes, o processador deve aguardar pelos reconhecimentos de *todos* os eventos de consolidação induzidos pela primeira operação de escrita antes que *qualquer* evento de consolidação possa ser induzido pela segunda. Tal restrição é capturada na Figura 13a pelas arestas (w_1^1, w_2^1) , (w_1^1, w_2^2) , (w_1^2, w_2^1) , e (w_1^2, w_2^2) . Além disso, para coerência, as operações de escrita devem ser obrigadas a concluir na mesma ordem em todos os processadores, como é capturado pelas arestas (W_1^1, W_2^1) e (W_1^2, W_2^2) . Como nem mesmo operações de leitura para o mesmo endereço são ordenadas em modelos relaxados, não existe aresta (r_3^2, r_4^2) , i.e. pares de operações de leitura podem ser consolidados em qualquer ordem.

A Figura 13b ilustra as restrições de ordem inferidas dos *traces* na Figura 12b. Note que uma operação de leitura pode apenas observar um valor de uma operação de escrita que completou em relação ao processador que emitiu aquela operação de leitura. Em consequência, como ambos R_3^2 e R_4^2 observam o valor escrito por W_1^2 , infere-se que (W_1^2, R_3^2) e (W_1^2, R_4^2) . Como (W_1^1, W_2^2) vale, devido a restrições de coerência, mas nem R_3^2 nem R_4^2 observam o valor escrito por W_2^2 , também se infere que (R_3^2, W_2^2) e (R_4^2, W_2^2) valem. Assim, nenhum erro seria indicado por um *checker* baseado em *traces*.

A Figura 13c mostra a ordem linear de eventos observados correspondente às marcas temporais da Figura 12c. Como nenhum ciclo é formado, nenhum erro de projeto é indicado por um *checker* baseado em comportamen-

tos nesse caso.

A Figura 13d mostra a ordem de eventos correspondente às marcas temporais da Figura 12d. Note que, contrariamente ao que ocorre sob forte atomicidade de escrita para um comportamento similar (veja Figura 3d), um ciclo *não* é formado pelos motivos a seguir. Apesar de L_4 obter seu valor depois de S_2 ter concluído em relação a P_1 (i.e. (W_2^1, R_4^2)) e poder até ter sido consolidada em relação a P_2 (pois w_2^2 e R_4^2 não estão ordenados), L_4 não pode observar ainda o valor escrito por S_2 , que é definido localmente a posteriori (i.e. (R_4^2, W_2^2)).

Note que dividir as operações de escrita em eventos de consolidação e conclusão capturou o fato que operações de escrita (fracamente atômicas) podem atrasar sua conclusão em relação a outros processadores e que um erro não deve ser detectado nesse caso. Portanto, ao projetar um novo *checker* baseado em comportamentos, diagnósticos de falsos positivos podem ser evitados usando-se uma representação um pouco mais realista para operações de escrita, como a apresentada esquematicamente neste exemplo.

6.3 ARQUITETURA DA PLATAFORMA

Com o auxílio do diagrama esquemático mostrado na Seção 2.2, aqui reproduzido na Figura 14, ilustra-se como estender a observabilidade para tratar a fraca atomicidade de escrita, monitorando eventos de memória apenas no primeiro nível hierárquico. Foram adicionados dois monitores, representados pelos símbolos \odot e \otimes .

Uma operação de leitura conclui no processador em que ela é emitida quando uma resposta ao pedido de leitura chega ao seu *buffer* de entrada. O valor recebido pode ser consolidado quando ele sai desse *buffer* e chega ao ROB (a operação de leitura é consolidada ao atingir o início do ROB). Se uma operação de leitura L_j é emitida por P_i , um evento de *conclusão* de leitura $(R_j)^i$ e um evento de *consolidação* de leitura $(r_j)^i$ podem ser observados pelos monitores \ominus e \oplus , respectivamente.

Um evento de *consolidação* de escrita representa o tempo em que uma operação de escrita ou pedido de invalidação (ou atualização) é colocado em um *buffer* como uma mensagem para o subsistema de memória. Um evento de consolidação ocorre no início do ROB do processador que emitiu a operação de escrita e pode ser observado pelo monitor \oplus . Por outro lado, outros processadores recebem mensagens relacionadas a essa escrita (para fins de coerência). Nesse caso, os eventos de consolidação que representam as mensagens recebidas em *buffers* remotos podem ser observados pelo monitor \odot . Assuma que uma operação de escrita S_j é emitida por P_i . Como mostrado

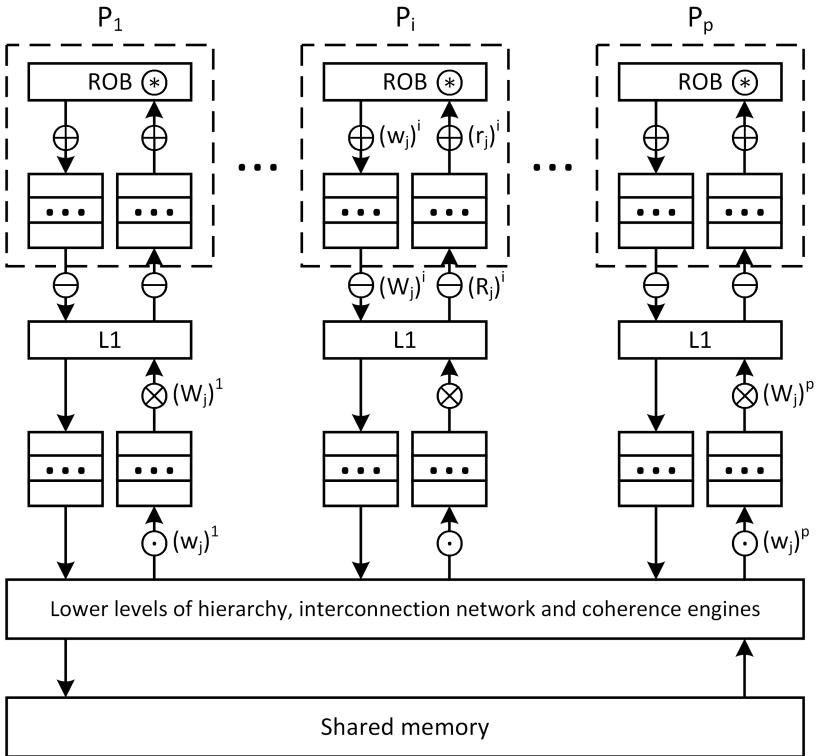


Figura 14: Um modelo genérico de sistema com memória compartilhada

na Figura 14, quando S_j atinge o início do ROB e é enviado ao subsistema de memória, um evento $(w_j)^i$ pode ser observado pelo monitor \oplus . Como resultado do mecanismo de coerência, outros eventos de consolidação $(w_j)^k$ (com $k \neq i$) podem ser observados pelos monitores \odot distribuídos pelo sistema de memória.

Um evento de *conclusão* representa o tempo em que a ação requisitada ocorre (i.e. quando um bloco da *cache* é escrito ou invalidado). Como a inserção de monitores dentro dos blocos da *cache* seria inadequada, o evento que *inicia* a real conclusão deve ser observado em seu lugar. A conclusão de S_j em relação a P_i , i. e. $(W_j)^i$, pode ser observada pelo monitor \ominus , enquanto sua conclusão em relação aos outros processadores, i.e. $(W_j)^k$ (com $k \neq i$) pode ser observada pelo monitor \otimes .²

A nova técnica de verificação proposta neste capítulo não amostra apenas eventos nos monitores \oplus , \ominus , e \otimes , mas também no monitor \otimes . Observando eventos com esses quatro monitores (apenas no primeiro nível da hierarquia de memória), acredita-se que podem ser dadas garantias plenas de verificação sob a fraca atomicidade de escrita.³

6.4 ESTRUTURA DO CHECKER E IDEIAS-BASE

A Figura 15 ilustra a comunicação entre os processos usados pelo *checker*. Uma seta sólida $A \rightarrow B$ indica que o processo A cria o processo B. Uma seta pontilhada $A \rightarrow B$ indica que o processo A envia mensagens para B.

Os monitores identificam as operações que passam por diversos pontos da plataforma de simulação e as repassam para os mecanismos de verificação, cada um responsável por verificar uma parte do modelo de memória. Os monitores também avisam aos mecanismos quando a simulação termina, para que eles verifiquem a condição final da execução, que é repassada para os processos que os criaram.

Os algoritmos cooperantes que constituem cada um dos mecanismos de verificação mostrados na Figura 15 são descritos na Seção 6.5.

Como cada evento observado na representação de projeto é uma tripla (op, a, v) , a distinção entre eventos de consolidação e conclusão só pode ser feita observando-os em pontos apropriados. Por isso, usamos monitores diferentes para distingui-los.

Note que, quando uma operação de leitura é (especulativamente) concluída, mas não é consolidada, a operação de leitura é cancelada (*squashed*),

²Para maior clareza, não se faz aqui nenhuma distinção entre eventos usados na modelagem e a cadeia de eventos reais que acontece fisicamente.

³Essas garantias estão sendo provadas para serem publicadas em um trabalho futuro.

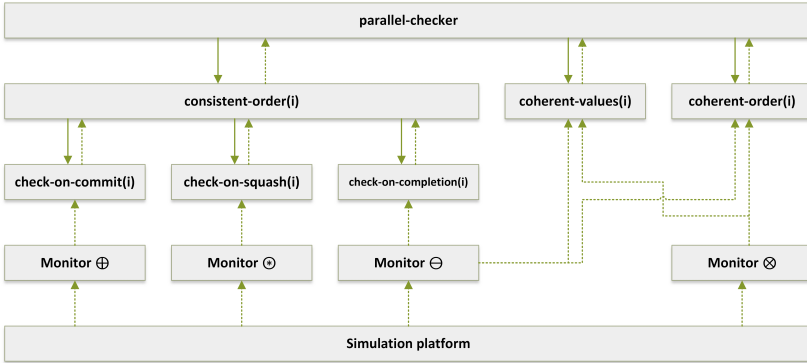


Figura 15: Estrutura do checker

antes de atingir o início do ROB. Para distinguir este comportamento correto de uma possível anomalia (i.e. para prevenir falsos positivos), observam-se não somente evento de consolidação e conclusão (através dos monitores \oplus e \ominus), mas também se observam eventos de cancelamento (através do monitor \otimes).

O checker proposto depende de uma noção fundamental, formalizada a seguir:

Definição 3 Dados dois eventos $e_j = (op_j, a_j, v_j)$ e $e_m = (op_m, a_m, v_m)$, diz-se que eles são equivalentes, escrito $e_j \equiv e_m$, sse $(op_j = op_m) \wedge (a_j = a_m) \wedge (v_j = v_m)$.

6.5 ALGORITMOS

O checker paralelo consiste de cinco mecanismos de verificação por processador, que são criados a partir da rotina principal descrita na Figura 16. Dado um processador i , seus mecanismos de verificação são processos *concorrentes* que aguardam eventos nos monitores \oplus , \ominus , \otimes e \otimes . Tais processos se comunicam através de *seqüências de eventos monitorados* (i.e. comportamentos observados). As seqüências denotadas por B_{\oplus}^i , B_{\ominus}^i e B_{\otimes}^i mantêm eventos observados pelos monitores \oplus , \ominus e \otimes , respectivamente. O checker também emprega uma tabela *hash*, denotada como B , para manter os comportamentos dos eventos de escrita para todos os endereços. Tais comportamentos são obtidos observando-se eventos com os monitores \ominus e \otimes . Cada entrada de B mantém a seqüência de todos os eventos de conclusão de escrita *globalmente* observados para um dado endereço. O mapeamento $h : A \mapsto \{1, 2, \dots, |A|\}$

```

1: for  $i \leftarrow 1, 2, \dots, p$  in parallel do
2:    $B_{\oplus}^i \leftarrow B_{\ominus}^i \leftarrow B_{\otimes}^i \leftarrow \emptyset$ 
3:   for  $j \leftarrow 1, 2, \dots, |A|$  do
4:      $B[j] \leftarrow \emptyset$ 
5:   for  $i \leftarrow 1, 2, \dots, p$  in parallel do
6:     for a given  $i$  in parallel do
7:       if  $\neg$  consistent-order( $i$ ) then
8:         return false
9:       if  $\neg$  coherent-values( $i$ ) then
10:        return false
11:      if  $\neg$  coherent-order( $i$ ) then
12:        return false
13:   return true

```

Figura 16: Rotina parallel-checker()

```

1: for a given  $i$  in parallel do
2:   check-on-commit( $i$ )
3:   check-on-squash( $i$ )
4:   check-on-completion( $i$ )
5:   return  $(B_{\oplus}^i = \emptyset) \wedge (B_{\ominus}^i = \emptyset)$ 

```

Figura 17: Rotina consistent-order(i)

denota a função *hash* empregada para acessar uma entrada da tabela (a mesma função será usada para outras tabelas *hash* definidas posteriormente). Para simplificar o pseudo-código, assume-se que B_{\oplus}^i , B_{\ominus}^i , B_{\otimes}^i , B e h têm escopo global. Além disso, emprega-se a notação de conjuntos para, implicitamente, representar sequências (elementos de conjuntos possuem índices que denotam suas posições na sequência).

Após todas as sequências terem sido inicializadas como vazias (linhas 1 a 4), a verificação de todos os processadores é iniciada em paralelo (linha 5). Dado um processador i , todos os cinco processos são iniciados concorrentemente (linha 6).⁴ O *checker* paralelo retorna falso assim que uma prova de inconsistência for detectada por qualquer um de seus processos.

A Figura 17 descreve a invocação concorrente dos três processos que verificam se as sequências de eventos de consolidação e de conclusão são consistentes (linhas 2 a 4). Ele retorna verdadeiro se todos os eventos de

⁴Três deles são iniciados na rotina consistent-order (Figura 17), invocada à linha 7. Os outros dois correspondem às linhas 9 e 11.

```

1:  $m \leftarrow 1$ 
2: repeat
3:   wait for  $e_m \leftarrow \text{event}(i, \oplus)$ 
4:    $B_{\oplus}^i \leftarrow B_{\oplus}^i \cup \{e_m\}$ 
5:   let  $e_m = (op, a, v)$ 
6:   if  $op = W$  then
7:      $F_{\ominus} \leftarrow \{e_k \in B_{\ominus}^i : e_k = (R, a, v) \wedge s_m \prec L_k\}$ 
8:     if  $F_{\ominus} \neq \emptyset$  then
9:       return false
10:  if  $op = R$  then
11:     $Q_{\ominus} \leftarrow \{e_q \in B_{\ominus}^i : e_q \equiv e_m\}$ 
12:    if  $Q_{\ominus} = \emptyset$  then
13:      return false
14:    else
15:       $j \leftarrow \min\{1 \leq q \leq |B_{\ominus}^i| : e_q \in Q_{\ominus}\}$ 
16:       $\text{match}(i, e_j)$ 
17:   $m \leftarrow m + 1$ 
18: until end-of-simulation
19: return true

```

Figura 18: Processo $\text{check-on-commit}(i)$

consolidação e conclusão tiverem sido emparelhados com equivalentes (linha 5), i.e., todos os eventos de consolidação e conclusão foram removidos das listas B_{\oplus}^i e B_{\ominus}^i .

Para descrever os processos concorrentes, a notação $\text{event}(i, m)$ denota o evento mais recente no processador i que foi observado por algum monitor $m \in \{\oplus, \ominus, \otimes\}$. Da mesma forma, a notação $\text{event}(i, \ominus, \otimes)$ denota o evento mais recente observado por um dos monitores \ominus ou \otimes .

O primeiro mecanismo de verificação é descrito na Figura 18. Quando um novo evento e_m é observado pelo monitor \oplus (linha 3), o processo o guarda na sequência de eventos consolidados (linha 4). Se o evento for de escrita, o processo verifica se há adiantamento (*bypassing*) adequado (linhas 5 a 9) da seguinte forma: se alguma operação de leitura foi concluída antes de uma operação de escrita conflitante que a precede na mesma *thread*, o processo retorna falso, pois isto é uma prova de que não houve adiantamento. Se o evento for de leitura, o processo tenta emparelhá-lo com um evento de leitura já observado previamente pelo monitor \ominus (linhas 10 a 16). Se nenhum evento equivalente for encontrado (linha 12), ele retorna falso, pois uma operação de leitura foi consolidada antes de ter sido completada, o que é uma prova de inconsistência.

```

1:  $m \leftarrow 1$ 
2: repeat
3:   wait for  $e_m \leftarrow \text{event}(i, \otimes)$ 
4:    $B_{\otimes}^i \leftarrow B_{\otimes}^i \cup \{e_m\}$ 
5:   let  $e_m = (op, a, v)$ 
6:   if  $op = R$  then
7:      $Q_{\ominus} \leftarrow \{e_q \in B_{\ominus}^i : e_q \equiv e_m\}$ 
8:     if  $Q_{\ominus} = \emptyset$  then
9:       return false
10:    else
11:       $j \leftarrow \min\{1 \leq q \leq |B_{\ominus}^i| : e_q \in Q_{\ominus}\}$ 
12:       $\text{match}(i, e_j)$ 
13:     $m \leftarrow m + 1$ 
14: until end-of-simulation
15: return true

```

Figura 19: Processo check-on-squash(i)

O segundo mecanismo de verificação, descrito na Figura 19, é similar, porém mais simples que o primeiro. Se um evento de cancelamento não possuir um evento de conclusão de leitura equivalente até o momento (linha 8), o processo retorna falso, pois uma operação de leitura foi cancelada antes de completar, o que é uma prova de inconsistência.

O terceiro mecanismo de verificação, descrito na Figura 20, também é algo similar aos anteriores. Se nenhum evento de consolidação equivalente puder ser encontrado para um evento de conclusão de escrita observado (linha 8), o processo retorna falso, pois uma operação de escrita concluiu antes de ser consolidada, o que é uma prova de inconsistência.

Todos os três mecanismos descritos até então invocam uma mesma rotina, descrita na Figura 21, que emparelha um evento de conclusão e_m com um evento de consolidação ou de cancelamento. Os eventos envolvidos são excluídos de exatamente duas sequências: o evento de conclusão é removido de B_{\ominus}^i (linha 10) e um evento equivalente é removido de B_{\oplus}^i (linha 6) ou de B_{\otimes}^i (linha 9). Primeiro, o algoritmo atribui a Q_{\oplus} os eventos de consolidação que são equivalentes a e_m (linha 1). Se pelo menos um evento equivalente for encontrado (linha 2), o primeiro da lista (e_j) é selecionado (linha 3). Se o evento e_m representa uma escrita concluída ou uma leitura concluída que foi consolidada (e, portanto, não cancelada), o evento de consolidação correspondente (e_j) é excluído de B_{\oplus}^i . Do contrário, e_m representa a conclusão de uma leitura que foi cancelada. Neste caso, o primeiro evento de cancelamento equivalente é excluído de B_{\otimes}^i . Finalmente, qualquer que seja o evento de

```

1:  $m \leftarrow 1$ 
2: repeat
3:   wait for  $e_m \leftarrow \text{event}(i, \ominus)$ 
4:    $B_{\ominus}^i \leftarrow B_{\ominus}^i \cup \{e_m\}$ 
5:   let  $e_m = (op, a, v)$ 
6:   if  $(op = W)$  then
7:      $Q_{\oplus} \leftarrow \{e_q \in B_{\oplus}^i : e_q \equiv e_m\}$ 
8:     if  $Q_{\oplus} = \emptyset$  then
9:       return false
10:    else
11:       $j \leftarrow \min\{1 \leq q \leq |B_{\oplus}^i| : e_q \in Q_{\oplus}\}$ 
12:       $\text{match}(i, e_j)$ 
13:     $m \leftarrow m + 1$ 
14: until end-of-simulation
15: return true

```

Figura 20: Processo check-on-completion(i)

```

1:  $Q_{\oplus} \leftarrow \{e_q \in B_{\oplus}^i : e_q \equiv e_m\}$ 
2: if  $Q_{\oplus} \neq \emptyset$  then
3:    $j \leftarrow \min\{1 \leq q \leq |B_{\oplus}^i| : e_q \in Q_{\oplus}\}$ 
4:   let  $e_m = (op, a, v)$ 
5:   if  $(op = W \vee (op = R \wedge Q_{\oplus} \neq \emptyset))$  then
6:      $B_{\oplus}^i \leftarrow B_{\oplus}^i - \{e_j\}$ 
7:   else
8:      $x \leftarrow \min\{1 \leq s \leq |B_{\otimes}^i| : e_s \in B_{\otimes}^i \wedge e_s \equiv e_m\}$ 
9:      $B_{\otimes}^i \leftarrow B_{\otimes}^i - \{e_x\}$ 
10:   $B_{\ominus}^i \leftarrow B_{\ominus}^i - \{e_m\}$ 

```

Figura 21: Rotina $\text{match}(i, e_m)$

```

1: for  $j \leftarrow 1, 2, \dots, |A|$  do
2:    $V^i[j] \leftarrow NIL$ 
3:  $m \leftarrow 1$ 
4: repeat
5:   wait for  $e_m \leftarrow \text{event}(i, \ominus, \otimes)$ 
6:   let  $e_m = (op, a, v)$ 
7:   if  $op = W$  then
8:      $V^i[h(a)] \leftarrow v$ 
9:      $m \leftarrow m + 1$ 
10:  if  $(op = R) \wedge (v \neq V^i[h(a)])$  then
11:    return false
12: until end-of-simulation
13: return true

```

Figura 22: Processo coherent-values(i)

conclusão (e_m), ele é sempre excluído da lista B_\ominus^i .

A Figura 22 descreve o quarto mecanismo de verificação. Ele usa uma tabela *hash* local, denotada como V^i . Cada entrada de V^i mantém um valor escrito por uma operação de escrita para um dado endereço. O algoritmo primeiramente inicializa todas as entradas da tabela (linha 2). Quando um novo evento é observado por \ominus ou \otimes , uma de duas ações é realizada. Quando o evento é de escrita, seu valor é armazenado na tabela *hash* (linha 8). Quando o evento é de leitura, o algoritmo verifica se o valor consumido equivale ao último valor produzido por uma operação de escrita para o mesmo endereço (linha 10) e retorna falso em caso contrário, pois isso é uma prova de incoerência.

Finalmente, a Figura 23 descreve o quinto mecanismo de verificação, que correlaciona globalmente os comportamentos locais dos eventos de escrita observados pelos monitores \ominus e \otimes . Ele emprega uma tabela *hash* local chamada de I^i , cujas entradas indicam, para cada endereço, a posição na sequência global dos eventos de escrita (B) em que se encontra o último evento de conclusão de escrita observado pelo processador i .

O algoritmo começa inicializando todas as entradas da tabela (linha 2) para apontar para a primeira posição na sequência global. Quando um novo evento de escrita e_m é observado (linhas 4 a 6), o algoritmo inicia no último evento observado pelo processador i (linha 7) e procura o primeiro evento equivalente a e_m que foi observado globalmente (linhas 8 e 9), i.e. dado um endereço a , ele faz uma busca na sequência $B[h(a)]$ de $I^i[h(a)]$ em diante. Se um evento equivalente for encontrado (linha 10), ele marca sua posição como a do último evento observado (linhas 11 a 12). Se tal evento equivalente não

```

1: for  $j \leftarrow 1, 2, \dots, |A|$  do
2:    $I^i[j] \leftarrow 1$ 
3: repeat
4:   wait for  $e_m \leftarrow \text{event}(i, \ominus, \otimes)$ 
5:   let  $e_m = (op, a, v)$ 
6:   if  $op = W$  then
7:      $x \leftarrow I^i[h(a)]$ 
8:     while  $(x < |B[h(a)]|) \wedge (e_x \in B[h(a)] \wedge e_x \neq e_m)$  do
9:        $x \leftarrow x + 1$ 
10:    if  $e_x \in B[h(a)] \wedge e_x \equiv e_m$  then
11:       $x \leftarrow x + 1$ 
12:       $I^i[h(a)] \leftarrow x$ 
13:    else
14:      if  $\exists e_y \in B[h(a)] : e_y \equiv e_m$  then
15:        return false
16:      else
17:         $x \leftarrow x + 1$ 
18:         $e_x \leftarrow e_m$ 
19:         $B[h(a)] \leftarrow B[h(a)] \cup \{e_x\}$ 
20:         $I^i[h(a)] \leftarrow x$ 
21: until end-of-simulation
22: return true

```

Figura 23: Processo coherent-order(i)

for encontrado *depois* da posição do último evento marcado, mas ele está de fato na sequência global (linha 14), ele deve ter aparecido na sequência global antes de $I^i[h(a)]$. Como essa incompatibilidade de ordenamentos é uma prova de incoerência, o algoritmo retorna falso. Se um evento equivalente certamente não está na sequência global, ele deve ser inserido pela primeira vez (linhas 17 to 20). O algoritmo retorna verdadeiro se nenhuma incompatibilidade de ordenamentos for encontrada.

6.6 STATUS DE VALIDAÇÃO E PERSPECTIVAS

Os algoritmos cooperantes descritos na seção anterior estão sendo validados teoricamente antes de serem avaliados experimentalmente.

Uma especificação formal para modelos de memória sob fraca atomicidade de escrita e máxima relaxação de ordem de programa já foi elaborada

pelo orientador e revisada pelo autor desta dissertação.

Já existem algumas provas teóricas contrastando a especificação formal do comportamento esperado com o comportamento observado pelo *checker* proposto, a serem publicadas posteriormente. Por exemplo, o algoritmo consistent-order (Figura 17) provadamente verifica todos os comportamentos especificados no escopo de um processador para um comportamento arbitrário observado.

Conforme avançam as provas dos demais algoritmos, espera-se provar plenas garantias de verificação (possivelmente com alguns ajustes nos algoritmos).

Os algoritmos aqui descritos resolvem instâncias do Problema 3 para modelos com fraca atomicidade de escrita. Contudo, há indícios de que eles podem ser generalizados para também tratar modelos com forte atomicidade de escrita. Isso também está sendo investigado.

A avaliação experimental dos algoritmos propostos será iniciada assim que a infraestrutura experimental necessária tiver sido preparada, como será discutido no próximo capítulo.

7 CONSIDERAÇÕES FINAIS

7.1 CONCLUSÕES

As tendências mostram que as arquiteturas de processadores estão caminhando para a relaxação total da ordem das operações, a não ser que a ordem de programa seja explicitamente imposta. Essa relaxação, em conjunto com o aumento do número de núcleos, dificulta sua verificação.

Mostrou-se que o reuso de *checkers* pós-silício para a verificação pré-silício é inadequado. A observabilidade proporcionada por representações de projeto do sistema permite a construção de *checkers* mais eficientes e escaláveis. Além disso, *checkers* que, provavelmente, possuem garantias plenas de verificação possuem uma eficácia significativamente superior a *checkers* sem garantias, permitindo que sejam analisados casos de teste menores para se encontrar a mesma quantidade de erros.

Encontrou-se também uma séria limitação de todos os *checkers* descritos na literatura até o momento. Eles não foram desenvolvidos para tratar a fraca atomicidade de escrita, uma importante característica dos modelos de memória mais relaxados, e que está presente em processadores já no mercado (ARM, 2012) (ARM, 2013a).

Essa análise foi levada em conta para se projetar os algoritmos de um novo *checker*, que utiliza a observabilidade das representações de projeto para tentar conseguir plenas garantias de verificação e bom desempenho, ao mesmo tempo que trata a fraca atomicidade de escrita.

7.2 TRABALHO EM ANDAMENTO

A plataforma de simulação está sendo adaptada para incorporar a fraca atomicidade de escrita. Isso permitirá comparar o *checker* proposto com os outros *checkers* aqui descritos, concebidos para a forte atomicidade de escrita, a fim de avaliar o impacto prático da generalização proposta. O adiantamento do valor das operações de escrita para as operações de leitura ainda precisa ser devidamente observado por algum dos monitores e tratado pelo *checker*.

A análise de complexidade dos algoritmos aqui propostos para o novo *checker* está em andamento, bem como as provas formais de garantias de verificação e sua avaliação experimental, que não foram parte do escopo original desta dissertação.

Os resultados da pesquisa em andamento serão descritos em artigo a

Tabela 6: Responsáveis pelas implementações dos *checkers* utilizados

Checker	Programadores	Referências
EXM	Eberle A. Rambo	(RAMBO, 2012) (RAMBO et al., 2012)
MSB	Leandro S. Freitas	(FREITAS, 2012) (FREITAS et al., 2013)
TSOTool	Gabriel G. Gava Olav P. Henschel	(HANGAL et al., 2004) (MANOVIT; HANGAL, 2005) (MANOVIT; HANGAL, 2006)
SSB	Eberle A. Rambo Gabriel G. Gava Olav P. Henschel	(SHACHAM et al., 2008)
IBE e IBT	Gabriel A. G. Andrade Olav P. Henschel	(CHEN et al., 2009) (HU et al., 2012)

ser submetido ao periódico *IEEE Transactions on Computers* (TC).

7.3 TRABALHOS FUTUROS

Apesar de quatro classes distintas de *checkers* terem sido comparadas para um mesmo modelo de memória e um mesmo conjunto de erros, resultados experimentais para modelos de memória distintos seriam úteis para avaliar até que ponto um *checker* depende do modelo de memória alvo.

Como *checkers on-the-fly* (como MSB) podem obter eficácia similar com casos de teste significativamente mais curtos do que *checkers* de melhor esforço e também podem ser 4 vezes mais rápidos do que o melhor *checker post-mortem* (EXM), há fortes indícios de que *checkers on-the-fly* (como MSB e o novo *checker* proposto) possam ser utilizados eficientemente durante a simulação de subsistemas de memória em RTL.¹ Assim, tais *checkers* poderiam ser usados em fases posteriores de projeto, tornando-os ainda mais atrativos para uso em ambientes industriais. Para verificar esses indícios, os experimentos aqui reportados deveriam ser repetidos para descrições RTL.

7.4 RECONHECIMENTOS

Diversos artefatos utilizados no desenvolvimento deste trabalho foram resultado de trabalhos anteriores. As Tabelas 6 e 7 listam os responsáveis por implementações de *checkers* e generalizações a partir de algoritmos anteriores.

¹Embora o uso de XCHECK para RTL tenha sido sugerido em (HU et al., 2012), esta dissertação mostrou que esta sugestão é inadequada.

Tabela 7: Autores da generalização proposta no Capítulo 6

Algoritmo	Autores	Baseado em
parallel-checker (Figura 16)	Luiz C. V. dos Santos	———
consistent-order (Figura 17)	Luiz C. V. dos Santos Olav P. Henschel	(FREITAS et al., 2013) Algoritmo 1
check-on-commit (Figura 18)		
check-on-squash (Figura 19)		
check-on-completion (Figura 20)	Luiz C. V. dos Santos Olav P. Henschel	(FREITAS et al., 2013) Algoritmo 2
match (Figura 21)		
coherent-values (Figura 22)	Luiz C. V. dos Santos	(RAMBO et al., 2012) Algoritmo 2
coherent-order (Figura 23)	Olav P. Henschel	———

Agradeço a todos estes, que contribuíram, direta ou indiretamente, para a viabilidade deste trabalho.

REFERÊNCIAS

- ABTS, D.; SCOTT, S.; LILJA, D. So many states, so little time: verifying memory coherence in the Cray X1. In: **Proceedings International Parallel and Distributed Processing Symposium**. IEEE Comput. Soc, 2003. p. 10. ISBN 0-7695-1926-1. ISSN 1530-2075.
- ADVE, S.; GHARACHORLOO, K. Shared memory consistency models: a tutorial. **Computer**, v. 29, n. 12, p. 66–76, 1996. ISSN 00189162.
- AMD. **AMD Opteron 6000 Series Platform**. 2012. Disponível em: <<http://www.amd.com/en-us/products/server/opteron/6000>>.
- ARM. **ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition**. 2012.
- ARM. **ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile**. 2013.
- ARM. **big.LITTLE Processing**. 2013. Disponível em: <<http://www.arm.com/products/processors/technologies/biglittleprocessing.php>>.
- ARVIND, A.; MAESSEN, J. Memory Model = Instruction Reordering + Store Atomicity. In: **33rd International Symposium on Computer Architecture (ISCA'06)**. IEEE, 2006. v. 34, n. 2, p. 29–40. ISBN 0-7695-2608-X.
- BINKERT, N. et al. The M5 Simulator: Modeling Networked Systems. **IEEE Micro**, v. 26, n. 4, p. 52–60, jul. 2006. ISSN 0272-1732.
- BINKERT, N. et al. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, ACM, v. 39, n. 2, p. 1, ago. 2011. ISSN 01635964.
- CHATTERJEE, P.; SIVARAJ, H.; GOPALAKRISHNAN, G. Shared memory consistency protocol verification against weak memory models: Refinement via model-checking. **Computer Aided Verification**, 2002.
- CHEN, Y. et al. Fast complete memory consistency verification. **Proc. of IEEE Int. Symp. on High Performance Computer Architecture (HPCA)**, n. 2007, p. 381–392, 2009.
- DEORIO, A.; WAGNER, I.; BERTACCO, V. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. **Proc. of IEEE Int. Symp. on High Performance Computer Architecture (HPCA)**, p. 405–416, 2009.

DEVADAS, S. Toward a Coherent Multicore Memory Model. **Computer**, IEEE Computer Society, v. 46, n. 10, p. 30–31, out. 2013. ISSN 0018-9162.

DUBOIS, M.; SCHEURICH, C.; BRIGGS, F. Memory access buffering in multiprocessors. **ACM SIGARCH Computer Architecture News**, ACM, v. 14, n. 2, p. 434–442, jun. 1986. ISSN 01635964.

FENSCH, C.; CINTRA, M. An OS-based alternative to full hardware coherence on tiled CMPs. In: **2008 IEEE 14th International Symposium on High Performance Computer Architecture**. IEEE, 2008. p. 355–366. ISBN 978-1-4244-2070-4. ISSN 1530-0897.

FREITAS, L. d. S. **Aceleradores e multiprocessadores em chip: o impacto da execução fora de ordem na verificação de funcionalidade e de consistência**. Tese (Mestrado) — Universidade Federal de Santa Catarina, 2012.

FREITAS, L. S. et al. On-the-fly verification of memory consistency with concurrent relaxed scoreboards. **Proc. of Design, Automation and Test in Europe (DATE)**, p. 631–636, 2013.

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

GHARACHORLOO, K.; GUPTA, A.; HENNESSY, J. Revision to "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors". 1993.

GHARACHORLOO, K. et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In: **Proceedings of The 17th Annual International Symposium on Computer Architecture**. IEEE Comput. Soc. Press, 1990. p. 15–26. ISBN 0-8186-2047-1.

GIBBONS, P. B.; KORACH, E. On testing cache-coherent shared memories. In: **Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures - SPAA '94**. New York, New York, USA: ACM Press, 1994. p. 177–188. ISBN 0897916719.

GIBBONS, P. B.; KORACH, E. Testing shared memories. **SIAM Journal on Computing**, v. 26, p. 1208–1244, 1997.

HAMMOND, L. et al. Transactional Memory Coherence and Consistency. **ACM SIGARCH Computer Architecture News**, ACM, v. 32, n. 2, p. 102, mar. 2004. ISSN 01635964.

HANGAL, S. et al. TSOtool. **ACM SIGARCH Computer Architecture News**, v. 32, n. 2, p. 114, mar. 2004. ISSN 01635964.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition: A Quantitative Approach**. Morgan Kaufmann, 2011. 856 p. ISBN 012383872X.

HENSCHEL, O. P.; SANTOS, L. C. V. dos. Pre-silicon verification of multiprocessor SoCs: The case for on-the-fly coherence/consistency checking. In: **2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)**. IEEE, 2013. p. 843–846. ISBN 978-1-4799-2452-3.

HENZINGER, T.; QADEER, S.; RAJAMANI, S. Verifying sequential consistency on shared-memory multiprocessor systems. **Computer Aided Verification**, 1999.

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

IBM. IBM System/370 Principles of Operation. **Publication Number GA22-7000-9, File Number S370-01**, 1983.

Intel Corporation. **Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes**. 2014.

KRISHNA, T. et al. Single-Cycle Multihop Asynchronous Repeated Traversal: A SMART Future for Reconfigurable On-Chip Networks. **Computer**, v. 46, n. 10, p. 48–55, out. 2013. ISSN 0018-9162.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, ACM, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 00010782.

LAMPORT, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. **IEEE Transactions on Computers**, IEEE, C-28, n. 9, p. 690–691, set. 1979. ISSN 0018-9340.

LENOSKI, D. et al. The directory-based cache coherence protocol for the DASH multiprocessor. In: **Proceedings of the 17th annual international symposium on Computer Architecture - ISCA '90**. New York, New York, USA: ACM Press, 1990. v. 18, n. 2SI, p. 148–159. ISBN 0897913663. ISSN 0163-5964.

MANOVIT, C.; HANGAL, S. Efficient algorithms for verifying memory consistency. **ACM symposium on Parallelism in algorithms**, p. 245–252, 2005.

MANOVIT, C.; HANGAL, S. Completely verifying memory consistency of test program executions. **The Twelfth International Symposium on High-Performance Computer Architecture**, p. 168–177, 2006.

MARTIN, M. M. K.; HILL, M. D.; SORIN, D. J. Why on-chip cache coherence is here to stay. **Communications of the ACM**, v. 55, n. 7, p. 78, jul. 2012. ISSN 00010782.

MARTIN, M. M. K. et al. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. **ACM SIGARCH Computer Architecture News**, ACM, v. 33, n. 4, p. 92, nov. 2005. ISSN 01635964.

MATEOSIAN, R. The PowerPC Architecture - A Specification of a New Family of RISC Processors [Micro Review]. **IEEE Micro**, v. 14, n. 5, p. 2, out. 1994. ISSN 0272-1732.

RAMBO, E. **Verificação de consistência de memória para sistemas integrados multiprocessados**. Tese (Mestrado) — Universidade Federal de Santa Catarina, 2012.

RAMBO, E. A.; HENSCHER, O. P.; SANTOS, L. C. V. dos. Automatic generation of memory consistency tests for chip multiprocessing. **18th IEEE International Conference on Electronics, Circuits and Systems (ICECS)**, 2011.

RAMBO, E. A. et al. On ESL verification of memory consistency for system-on-chip multiprocessing. **Proc. of Design, Automation and Test in Europe (DATE)**, p. 9–14, 2012.

ROY, A.; ZEISSET, S. Fast and generalized polynomial time memory consistency verification. **Proceedings of the 18th international conference on Computer Aided Verification**, p. 503–516, 2006.

SCHUCHHARDT, M. et al. The Impact of Dynamic Directories on Multicore Interconnects. **Computer**, v. 46, n. 10, p. 32–39, out. 2013. ISSN 0018-9162.

SHACHAM, O. et al. Verification of chip multiprocessor memory systems using a relaxed scoreboard. **Proc. of IEEE/ACM Int. Symp. on Microarchitecture (MICRO)**, Ieee, p. 294–305, nov. 2008.

SHIM, K. S. et al. Design tradeoffs for simplicity and efficient verification in the Execution Migration Machine. In: **2013 IEEE 31st International Conference on Computer Design (ICCD)**. IEEE, 2013. p. 145–153. ISBN 978-1-4799-2987-0.

SINDHU, P. S.; FRAILONG, J.-M.; CEKLEOV, M. **Formal Specification of Memory Models**. Boston, MA: Springer US, 1992. 25–41 p. ISBN 978-1-4613-6601-0.

SITES, R. **Alpha architecture reference manual**. Digital Press, 1992.

SPARC International Inc. **The SPARC Architecture Manual Version 8**. [s.n.], 1992. ISBN 0138250014.

Sun Microsystems Inc. **UltraSPARC Architecture 2005**. 2008. Disponível em: <<http://www.oracle.com/technetwork/systems/opensparc/t1-06-ua2005-d0-9-2-p-ext-1537734.html>>.

Synopsys Inc. **OpenVera**. s.d. Disponível em: <<http://www.open-vera.com/>>.